

Unix Maven Plug-in

Trygve Laugstøl

Unix Maven Plug-in

Trygve Laugstøl

Publication date \$Id: handbook.xml 14315 2011-07-08 16:08:05Z trygvis \$

Acknowledgements

Trygve Laugstøl would like to thank the following persons and companies in no particular order for their effort in helping with the development of the Unix Maven Plugin

- Erik Drolshammer, Objectware for providing me with RPM related use cases and being helpful with testing new features.
- Conax for providing use cases and allowing me to work on the plugin while on the clock.
- Arktekk for sponsoring me to work on the plugin.

Preface	iv
I. Guide	5
1. Introduction	6
1.1. Scope	6
1.2. About the plugin	6
1.3. How the Plugin Work	6
1.3.1. Meta Data Generation	6
1.3.2. Assembly Operation Execution	7
1.4. About the Alternatives to the Unix Maven Plugin	7
2. Concepts	8
2.1. Operation Modes	8
2.2. Package Formats	8
2.3. Platform	8
2.4. Package File System and File System Objects	8
3. Default Values	9
3.1. Default Assembly Operations	9
II. Example of Usage	10
Introduction to the Examples	xi
4. Single, Standalone Application	12
4.1. The Deb version	12
4.1.1. pom.xml for the deb version	12
4.1.2. Resulting Deb	13
4.2. The Pkg version	13
4.2.1. pom.xml for the pkg version	13
4.2.2. Resulting PKG	14
4.3. The RPM version	16
4.3.1. pom.xml for RPM version	16
4.3.2. Resulting RPM	17
4.4. The Zip version	17
4.4.1. pom.xml for the zip version	17
4.4.2. Resulting ZIP	18
5. Other	19
III. Reference	20
6. Unix Package Definition	21
6.1. Deb	22
6.2. Pkg	22
6.3. Rpm	22
6.4. Version Calculation	22
6.4.1. Version Calculation for Snapshot Versions	22
6.4.2. Version Calculation for Release Versions	23
6.5. Mapping to Native Formats	23
6.5.1. Deb Specific Mappings	24
6.5.2. Pkg Specific Mappings	24
6.5.3. Rpm Specific Mappings	24
7. Action Scripts	25
7.1. Action Scripts in Primary Artifact Mode	25
7.2. Action Scripts in Attached Artifact Mode	25
8. Operating Requirements	26
9. Assembly Operations	27
9.1. Common Settings and Behaviour	27
9.1.1. Artifact Naming and Identification	27
9.1.2. File renaming	28
9.1.3. Includes and Excludes	28
9.1.4. File Attributes	28
9.2. Copy File	29
9.3. Copy Artifact	29
9.4. Make Directory	30
9.5. Set Attributes	30

9.6. Symlink	31
9.7. Copy Directory, Extract Artifact and Extract File	31
9.7.1. Copy Directory	32
9.7.2. Extract Artifact	32
9.7.3. Extract File	32
10. Creating Native Package Repositories	33
10.1. Creating Debian/APT Repositories	33
10.2. Creating RPM/Yum Repositories	33
10.3. Creating pkg-get/pkgutil Repositories	33
11. Troubleshooting	34
11.1. Debugging	34
IV. Useful Tips For Making Useful Packages	35
12. Package Command Reference	36
12.1. dpkg Commands	36
12.1.1. Listing All Files In a Package	36
12.2. rpm Commands	36
12.2.1. List All Files In a Package	36
12.2.2. List All Files In a Package With File Permissions	36
12.2.3. Listing All Available Software Groups	36
12.3. zip Commands	37
13. Snippets for Scripts	38
13.1. RedHat/Fedora	38
13.1.1. Making your server start on boot	38
V. Version History and Upgrading	39
Preface to Part IV	xl
14. 1.0-alpha-4	41
14.1. Change Log	41
14.2. Upgrading from 1.0-alpha-3	41

Preface

This handbook is the complete documentation of the Unix Maven Plugin. It has four major parts:

- Part I - Guide: An introduction and overview over the plugin. This part is the first you should read if you're looking into using the plugin or is getting started.
- Part II - Examples: This part contains a set of practical examples showing different aspects of the plugin.
- Part III - Reference: This is the reference manual for the plugin. Documents all available options and assembly operations.
- Part IV - Package Command Reference: A list of commands useful for looking into and debugging assembled packages.
- Part V - Version History and Upgrading: This part contains a list of all major features added in the different versions and explanations on how to upgrade between the different versions.

The book is under continuous development and is written in parallel with the plugin. It will only document the current version until a stable release is done. See Part V, “Version History and Upgrading” on how to upgrade from one version to the next version.

Currently there are chapters and sections that serve as placeholders and there are warnings which serve as TODO entries. These will be filled in and resolved before a final 1.0 release.

Part I. Guide

Chapter 1. Introduction

1.1. Scope

Warning

What does the plugin cover, what does it not cover. Contrast in particular with the Assembly plugin

1.2. About the plugin

The Unix Maven Plugin is meant to fill in the gap between creating platform independent applications and reality. It make it possible to create Java application with your standard Java development stack, but still be able to install the applications using the native tools that your system administrators already know.

As it automates yet another step in the development chain¹, it also give a new set of possibilities that can be taken advantage of. In particular the ability to store production ready binaries in a repository and the ability to install, remove and upgrade the software in an uniform and consistent way across an entire company.

Both of these abilities is similar to what Maven did for Java developers with the repository concept with Maven 1 and the high focus on build unification in Maven 2.

1.3. How the Plugin Work

This section is a high level explanation of how the plugin work.

The plugin goes through two major phases:

- Meta data generation: Each package format contain some form of meta data that it uses when the package is being installed or removed. The plugin will pick up as much meta data as possible from the Maven POM as possible, but there are certain options that might have to be filled in, depending on the package format.
- Assembly operation execution: For each package that is to be created, a virtual package file system is created with pointers to the original file or which file in and archive the file came from. The assembled file system will be used when the physical package is created.

1.3.1. Meta Data Generation

The first thing that happens is that the plugin starts to gather all the information it needs to build the package. It has main sources it uses:

- The general information in the POM
- The `<configuration>` of the plugin is used to set up defaults and contain, amongst other settings, a shared list of assembly operations
- The per-package configuration `<package>` entry

¹Development chain here refers to all the step that's between writing the code in an IDE to having it run in production or a production-like environment.

1.3.2. Assembly Operation Execution

The virtual package file system is just like a normal file system. Files can be added, and their attributes can be modified by any operation. As in a normal file system, if the a file is written to the file system twice, the last one will overwrite the first one.

As the file system is just a virtual file system with pointers to the original file or files in an archive only the last file will actually be used when creating the physical package.

When an existing file object is used as a source, as much as possible of the existing attributes will be copied from the object.

After all the assembly operations has been executed, the physical package file will be created.

See also Section 2.4, “Package File System and File System Objects”.

1.4. About the Alternatives to the Unix Maven Plugin

Note

This information is current as of May 2009.

The Unix Maven Plugin contains functionality that is similar to at least four existing plugins: The Deb Maven plugin, the RPM Maven Plugin, the Solaris Maven plugin and the Maven Assembly plugin.

The Unix plugin is loosely based on the source code of the Deb, Solaris and Rpm Maven plugins and was meant as a unified and complete replacement for all three plugins. The plugins supported different areas of functionality and the Unix plugin has implemented all features for all formats where applicable. The Unix plugin is already more complete and easier to use than the Deb and Solaris plugins. The RPM plugin still has features that the Unix plugin does not support yet.

One major feature of the Unix Maven plugin is the support for assembling the files to be included into the package, including file attributes. The Assembly plugin is cumbersome to use and gives little to no control over the file attributes that are requested. If the Unix plugin was to require a pre-assembled file system before executing the process would be significantly slower as all the files would have to be written to disk in an intermediate area. The Unix Maven plugin optimizes this as much as possible and will in many cases read the files directly from inside the Maven repository or from archives in a repository. This gives significant speed improvements when large binaries are created.

Chapter 2. Concepts

2.1. Operation Modes

The plugin can operate in two major modes, depending on how it is configured:

Primary artifact mode	The plugin is running in primary mode when the project has set a <packaging> to one of the packaging that the plugin support. The plugin will install an artifact as the primary artifact into the repository
Attached artifact mode	The plugin is running in attached mode when the project is executing the one of the package-FOO-attached goals. The plugin will install a package in addition to the primary artifact which will be delivered by another plugin.

2.2. Package Formats

A package format is a specific file format that a platform supports. The plugin currently support three formats; deb, pkg and rpm.

2.3. Platform

A platform is a specific operating system, for example Debian or Solaris. Each platform has its own native package format while some platforms support multiple formats.

2.4. Package File System and File System Objects

A package file system is a virtual file system created while the plugin executes all the assembly operations. It consist of file objects that represent regular files, directories and symbolic links.

Each file system object has a set of attributes similar to a normal Unix file system:

User and group	These two attributes control the user and group that own the file.
Mode	The read, write and execute permissions of the file object.
Tags	Each file object can have a set of tags which are plain text labels that each package format can use as an extension method.

Example 2.1. Example of file mode specifications

- 0644 becomes `-rw-r--r--`
- 0744 becomes `-rwxr--r--`

Note

Not all file objects support all attributes.

Chapter 3. Default Values

3.1. Default Assembly Operations

Warning

Show a completely expanded `<configuration>` section with defaults

Part II. Example of Usage

Introduction to the Examples

Warning

Explain the general format of each example, which concepts are used. Other plugins used.
Different formats.

Chapter 4. Single, Standalone Application

Warning

Missing appassembler snippet

This is an example on how a simple, standalone application can be packaged. The example is available in all four formats.

Keywords:

- Primary Artifact Mode
- <copyArtifact>
- <extractArtifact>
- <symlink>

4.1. The Deb version

4.1.1. pom.xml for the deb version

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo.unix.example</groupId>
  <artifactId>basic</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>deb</packaging>
  <name>Hudson</name>
  <repositories>
    <repository>
      <id>java.net</id>
      <url>http://download.java.net/maven/2</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>org.jvnet.hudson.main</groupId>
      <artifactId>hudson-war</artifactId>
      <version>1.255</version>
      <type>war</type>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>unix-maven-plugin</artifactId>
        <version>1.0-alpha-5</version>
        <extensions>true</extensions>
        <configuration>
          <contact>Acme</contact>
          <contactEmail>support@acme.org</contactEmail>
          <deb>
            <section>devel</section>
          </deb>
          <assembly>
            <copyArtifact>
              <artifact>org.jvnet.hudson.main:hudson-war:war</artifact>
              <toFile>/opt/hudson/hudson.war</toFile>
              <attributes>
```

```

        <user>hudson</user>
        <group>hudson</group>
        <mode>0666</mode>
    </attributes>
</copyArtifact>
<extractArtifact>
    <artifact>org.jvnet.hudson.main:hudson-war:war</artifact>
    <to>/opt/hudson/doc</to>
    <includes>
        <include>*/license.txt</include>
    </includes>
    <pattern>*/(*.txt)</pattern>
    <replacement>$1</replacement>
</extractArtifact>
<symlink>
    <path>/var/log/hudson</path>
    <value>/var/opt/hudson/log</value>
</symlink>
</assembly>
</configuration>
</plugin>
</plugins>
</build>
</project>

```

4.1.2. Resulting Deb

After running **mvn install** on the project, the package is packaged into `target/basic-1.0-SNAPSHOT.deb`.

The package will contain these files:

```

$ dpkg-deb -c target/*.deb
drwxr-xr-x root/root      0 2011-07-08 17:10 ./
drwxr-xr-x root/root      0 2011-07-08 17:10 ./var/
drwxr-xr-x root/root      0 2011-07-08 17:10 ./var/log/
drwxr-xr-x root/root      0 2011-07-08 17:10 ./opt/
drwxr-xr-x root/root      0 2011-07-08 17:10 ./opt/hudson/
drwxr-xr-x root/root      0 2011-07-08 17:10 ./opt/hudson/doc/
-rw-r--r-- root/root    1544 2008-10-02 02:07 ./opt/hudson/doc/dc-license.txt
-rw-r--r-- root/root      49 2008-10-02 02:07 ./opt/hudson/doc/atom-license.txt
-rw-r--r-- root/root 20623413 2011-04-27 10:41 ./opt/hudson/hudson.war
lrwxrwxrwx root/root      0 2011-07-08 17:10 ./var/log/hudson -> /var/opt/hudson/log

```

The generated control file:

```

$ dpkg-deb -f target/*.deb
Package: basic
Section: devel
Priority: standard
Maintainer: Acme
Version: 1.0-20110708.151032
Architecture: all
Description: Hudson Solaris Package

```

4.2. The Pkg version

4.2.1. pom.xml for the pkg version

```

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo.unix.example</groupId>
  <artifactId>basic</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pkg</packaging>
  <name>Hudson</name>
  <repositories>
    <repository>
      <id>java.net</id>
      <url>http://download.java.net/maven/2</url>
    </repository>
  </repositories>
</project>

```

```

</repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.jvnet.hudson.main</groupId>
    <artifactId>hudson-war</artifactId>
    <version>1.255</version>
    <type>war</type>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>unix-maven-plugin</artifactId>
      <version>1.0-alpha-5</version>
      <extensions>true</extensions>
      <configuration>
        <contact>Acme</contact>
        <contactEmail>support@acme.org</contactEmail>
        <assembly>
          <copyArtifact>
            <artifact>org.jvnet.hudson.main:hudson-war:war</artifact>
            <toFile>/opt/hudson/hudson.war</toFile>
            <attributes>
              <user>hudson</user>
              <group>hudson</group>
              <mode>0666</mode>
            </attributes>
          </copyArtifact>
          <extractArtifact>
            <artifact>org.jvnet.hudson.main:hudson-war:war</artifact>
            <to>/opt/hudson/doc</to>
            <includes>
              <include>*/license.txt</include>
            </includes>
            <pattern>*/(*.txt)</pattern>
            <replacement>${1}</replacement>
          </extractArtifact>
          <symlink>
            <path>/var/log/hudson</path>
            <value>/var/opt/hudson/log</value>
          </symlink>
        </assembly>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

4.2.2. Resulting PKG

After running `mvn install` on the pkg version of the project, the package is packaged into `target/basic-pkg-1.0.pkg`.

The generated meta data will look like this:

```

$ pkginfo -l -d target/basic-*.pkg
  PKGINST: basic
    NAME: Hudson
  CATEGORY: application
    ARCH: all
  VERSION: 1.0-20090513.193125
  PSTAMP: 20090513.193125
    EMAIL: support@acme.org
  STATUS: spooled
    FILES:      11 spooled pathnames
              5 directories
              2 setuid/setgid executables
              2 package information files
              40286 blocks used (approx)

```

Notice that the SNAPSHOT part of the version string has been replaced with the a timestamp.

To see verify the paths and their attributes, run this:

```
$ pkgchk -l -d target/basic-*.pkg all
Checking uninstalled stream format package <basic> fromn <../target/basic-1.0-
SNAPSHOT.pkg>
## Checking control scripts.
## Checking package objects.
Pathname: /opt
Type: directory
Expected mode: 1777777777
Expected owner: ?
Expected group: ?
Current status: installed

Pathname: /opt/hudson
Type: directory
Expected mode: 0755
Expected owner: nobody
Expected group: nogroup
Current status: installed

Pathname: /opt/hudson/doc
Type: directory
Expected mode: 0755
Expected owner: nobody
Expected group: nogroup
Current status: installed

Pathname: /opt/hudson/doc/atom-license.txt
Type: regular file
Expected mode: 0644
Expected owner: nobody
Expected group: nogroup
Expected file size (bytes): 49
Expected sum(1) of contents: 4473
Expected last modification: Oct 02 02:07:36 2008
Current status: installed

Pathname: /opt/hudson/doc/dc-license.txt
Type: regular file
Expected mode: 0644
Expected owner: nobody
Expected group: nogroup
Expected file size (bytes): 1544
Expected sum(1) of contents: 59072
Expected last modification: Oct 02 02:07:36 2008
Current status: installed

Pathname: /opt/hudson/hudson.war
Type: regular file
Expected mode: 0666
Expected owner: hudson
Expected group: hudson
Expected file size (bytes): 20623413
Expected sum(1) of contents: 3301
Expected last modification: Oct 24 23:08:16 2008
Current status: installed

Pathname: /var
Type: directory
Expected mode: 1777777777
Expected owner: ?
Expected group: ?
Current status: installed

Pathname: /var/log
Type: directory
Expected mode: 0755
Expected owner: nobody
Expected group: nogroup
Current status: installed

Pathname: /var/log/hudson
Type: symbolic link
Source of link: /var/opt/hudson/log
Current status: installed
```

```
Pathname: pkginfo
Type: installation file
Expected file size (bytes): 161
Expected sum(1) of contents: 12122
Expected last modification: May 13 21:31:26 2009
```

```
## Checking is complete.
```

4.3. The RPM version

4.3.1. pom.xml for RPM version

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo.unix.example</groupId>
  <artifactId>basic</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pkg</packaging>
  <name>Hudson</name>
  <repositories>
    <repository>
      <id>java.net</id>
      <url>http://download.java.net/maven/2</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>org.jvnet.hudson.main</groupId>
      <artifactId>hudson-war</artifactId>
      <version>1.255</version>
      <type>war</type>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>unix-maven-plugin</artifactId>
        <version>1.0-alpha-5</version>
        <extensions>true</extensions>
        <configuration>
          <contact>Acme</contact>
          <contactEmail>support@acme.org</contactEmail>
          <assembly>
            <copyArtifact>
              <artifact>org.jvnet.hudson.main:hudson-war:war</artifact>
              <toFile>/opt/hudson/hudson.war</toFile>
              <attributes>
                <user>hudson</user>
                <group>hudson</group>
                <mode>0666</mode>
              </attributes>
            </copyArtifact>
            <extractArtifact>
              <artifact>org.jvnet.hudson.main:hudson-war:war</artifact>
              <to>/opt/hudson/doc</to>
              <includes>
                <include>*/license.txt</include>
              </includes>
              <pattern>*/(*.txt)</pattern>
              <replacement>${1}</replacement>
            </extractArtifact>
            <symlink>
              <path>/var/log/hudson</path>
              <value>/var/opt/hudson/log</value>
            </symlink>
          </assembly>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

4.3.2. Resulting RPM

After running `mvn install` on the rpm version of the project, the package is packaged into `target/basic-pkg-1.0.rpm`.

The package will contain these files:

```
$ rpm -q -l -p target/basic-*.rpm
/opt
/opt/hudson
/opt/hudson/doc
/opt/hudson/doc/atom-license.txt
/opt/hudson/doc/dc-license.txt
/opt/hudson/hudson.war
/var
/var/log
/var/log/hudson
```

4.4. The Zip version

4.4.1. `pom.xml` for the zip version

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo.unix.example</groupId>
  <artifactId>basic</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pkg</packaging>
  <name>Hudson</name>
  <repositories>
    <repository>
      <id>java.net</id>
      <url>http://download.java.net/maven/2</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>org.jvnet.hudson.main</groupId>
      <artifactId>hudson-war</artifactId>
      <version>1.255</version>
      <type>war</type>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>unix-maven-plugin</artifactId>
        <version>1.0-alpha-5</version>
        <extensions>true</extensions>
        <configuration>
          <contact>Acme</contact>
          <contactEmail>support@acme.org</contactEmail>
        </configuration>
        <assembly>
          <copyArtifact>
            <artifact>org.jvnet.hudson.main:hudson-war:war</artifact>
            <toFile>/opt/hudson/hudson.war</toFile>
            <attributes>
              <user>hudson</user>
              <group>hudson</group>
              <mode>0666</mode>
            </attributes>
          </copyArtifact>
          <extractArtifact>
            <artifact>org.jvnet.hudson.main:hudson-war:war</artifact>
            <to>/opt/hudson/doc</to>
            <includes>
              <include>*/*/license.txt</include>
            </includes>
          </extractArtifact>
        </assembly>
      </plugin>
    </plugins>
  </build>
</project>
```

```
    </includes>
    <pattern>.*/(.*.txt)</pattern>
    <replacement>$1</replacement>
  </extractArtifact>
  <symlink>
    <path>/var/log/hudson</path>
    <value>/var/opt/hudson/log</value>
  </symlink>
</assembly>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

4.4.2. Resulting ZIP

After running **mvn install** on the rpm version of the project, the package is packaged into target/basic-pkg-1.0.rpm.

The zip will contain these files:

```
$ unzip -l target/basic-*.zip
```

Chapter 5. Other

Other examples to write

Package Attached to a WAR

Show a normal "hello world" web application (generated from the webapp archetype perhaps) which creates an attached package

Show a complete standalone application with appassembler and docx plugin to generate man pages

Useful to show how to interact with other plugins. Should also show of default values if possible

Package for multiple platforms

Show how the src/unix/files and script directories are picked

Part III. Reference

Chapter 6. Unix Package Definition

When the plugin is collecting all relevant data for a package it collects all the information in a generic object called "package parameters". This object is used as the basis when the package is created. See Section 6.5, "Mapping to Native Formats" on how the package parameters map to the format specific fields.

Note

- Parameters that reference `mojo` refers to parameters configured in a `<configuration>` block.
- When multiple values are available, the first one is selected.

Field	Required	Source(s)	Description
<code>groupId</code>	Y	<code>project.groupId</code>	
<code>artifactId</code>	Y	<code>project.artifactId</code>	
<code>version</code>	N/A	Calculated	This field is calculated from different sources. See Section 6.4, "Version Calculation" for details.
<code>revision</code>	N	<code>mojo.revision</code>	This field is calculated from different sources unless specified. See Section 6.4, "Version Calculation" for details.
<code>id</code>	N	<code>package.id</code>	The default value is the lower case version of the artifact id.
<code>name</code>	N	1. <code>package.name</code> 2. <code>mojo.name</code> 3. <code>project.name</code>	Maven will supply a default name if none is given.
<code>description</code>	N	1. <code>package.description</code> 2. <code>mojo.description</code> 3. <code>project.description</code>	
<code>contact</code>	N	<code>mojo.contact</code>	
<code>contactEmail</code>	N	<code>mojo.contactEmail</code>	
<code>license</code>	N	<code>project.licenses</code>	The first license listed in the POM will be used.
<code>architecture</code>	N	1. <code>mojo.architecture</code> 2. <code>format.architecture</code>	A default value indicating that the package is architecture independent is used if applicable.

Warning

Particular formats might add additional requirements.

Tip

For more information see the class `org.codehaus.mojo.unix.PackageParameters`.

In addition to the generic object, additional information is collected for each type of package.

6.1. Deb

Field	Required	Source(s)	Description
priority	Y	deb.priority	
section	Y	deb.section	

For the allowed values for the priority field, see `priorities` [<http://www.debian.org/doc/debian-policy/ch-archive.html#s-priorities>] and `sections` [<http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections>] sections in the Debian Policy Manual.

6.2. Pkg

The `pkg` format does not have any specific settings. All settings is derived from the generic settings.

6.3. Rpm

Additional requirements for the RPM format:

- At least one license is specified in the POM.

Field	Required	Source(s)	Description
group	Y	rpm.group	

6.4. Version Calculation

Tip

This information is derived from `org.codehaus.mojo.unix.maven.core.VersionTest` [<http://mojo.codehaus.org/unix/xref-test/org/codehaus/mojo/unix/maven/core/VersionTest.html>].

When the plugin is creating a unix package it also create a version object which consist of the following fields:

Version	The "main" part of the version configured in the POM. The main part is everything before the last dash in the version string.
Revision	The "rest" part of the version configured in the POM.
Timestamp	The timestamp of the build. This is the same timestamp as the one used for installing the artifacts in the repository.

The overall strategy for SNAPSHOT versions is to make the version field in the package as specific as possible. This means that the build timestamp will be appended to the version string where applicable.

6.4.1. Version Calculation for Snapshot Versions

Version	Revision	Format	Result
1.2-SNAPSHOT	Not set	Deb	1.2-20090423095107
		Pkg	1.2-20090423095107
		Rpm	1.2_20090423095107, rev: 1
	3	Deb	1.2-3-20090423095107

Version	Revision	Format	Result
		Pkg	1.2-3-20090423095107
		Rpm	1.2_20090423095107, rev: 3
1.2-3-SNAPSHOT	Not set	Deb	1.2-3-20090423095107
		Pkg	1.2-3-20090423095107
		Rpm	1.2_20090423095107, rev: 3
	3	Deb	1.2-3-3-20090423095107
		Pkg	1.2-3-3-20090423095107
		Pkg	1.2_3_20090423095107, rev: 3

6.4.2. Version Calculation for Release Versions

Version	Revision	Format	Result
1.2	Not set	Deb	1.2
		Pkg	1.2
		Rpm	1.2, rev: 1
	3	Deb	1.2
		Pkg	1.2
		Rpm	1.2, rev: 3
1.2-3	Not set	Deb	1.2-3
		Pkg	1.2-3
		Rpm	1.2, rev: 3
	3	Deb	1.2-3-3
		Pkg	1.2-3-3
		Pkg	1.2_3, rev: 3

6.5. Mapping to Native Formats

This table describe the relationship between the fields in the pom.xml and the fields in the generated package.

In addition to the generic fields, some formats has additional mappings.

Table 6.1. Mapping between the generic definition and the specific format

Unix Package	Deb	Pkg	Rpm
version	Version	VERS	Version
revision			Release
id	Package	PKG	Name
name	Description	NAME	Summary
description		DESC	Description
contact	Maintainer		
contactEmail		EMAIL	
license			License
architecture	Architecture	ARCH	

6.5.1. Deb Specific Mappings

The `Architecture` field will default to `all` if not set.

6.5.2. Pkg Specific Mappings

The `PSTAMP` field in `pkginfo` is set to `version.timestamp`.

6.5.3. Rpm Specific Mappings

Chapter 7. Action Scripts

All the native packaging formats support running programs when the packages are installed and removed. The plugin has automatic support for picking up scripts from the source folders.

The plugin will automatically pick up scripts from the `src/main/unix/scripts` directory. If a build is generating more than one artifact, it is possible to share common parts of the script between the packages based on the package id. The plugin uses two different strategies for naming the scripts based on the operation mode as described below.

7.1. Action Scripts in Primary Artifact Mode

In the primary artifact mode the scripts are named using the naming convention of the current format:

Table 7.1. Naming convention for action scripts per format

Action	Deb	Pkg	Rpm
Before installation	preinst	preinstall	pre
After installation	postinst	postinstall	post
Before removal	prerm	preremove	preun
After removal	postrm	postremove	postun

Using the "after installation" script for the Deb format as an example, the plugin will generate a `postinst` file based on the concatenation of these two files:

- `src/main/unix/scripts/postinst`
- `src/main/unix/scripts/postinst-<package id>`

If one of the files are missing it will be skipped.

7.2. Action Scripts in Attached Artifact Mode

In the attached artifact mode the scripts are named after both the package id and the format of the current package. The action script files uses a generic naming convention.

Table 7.2. Naming convention for generic action scripts

Action	Generic
Before installation	pre-install
After installation	post-install
Before removal	pre-remove
After removal	post-remove

Again using the "after installation" for the deb format as an example, the plugin will generate a `postinst` file based on the concatenation of these files:

- `src/main/unix/scripts/post-install`
- `src/main/unix/scripts/post-install-<package id>`
- `src/main/unix/scripts/post-install-<package id>-deb`

If one of the files are missing it will be skipped.

Chapter 8. Operating Requirements

Warning

Document the required binaries

Chapter 9. Assembly Operations

Table 9.1. Summary of assembly operations

Copy file	Copies a single file
Copy directory	Copies a directory structure
Copy artifact	Copies an artifact from the repository
Extract file	Extracts a single file
Extract artifact	Extracts an artifact from the repository
Make directory	Creates one or more directories
Set attributes	Sets file attributes on a file set
Symlink	Creates a symlink

9.1. Common Settings and Behaviour

All `to` elements refer to a path *within* the package.

Warning

Explain "file object" somewhere

9.1.1. Artifact Naming and Identification

The plugin has operations that can use artifacts from the Maven repository directly. In order to do this the plugin requires that you have a dependency on the artifact that you want to use. This is required for proper ordering in the Maven reactor.

When referring to an artifact from an assembly operation the normal `groupId:artifactId[:classifier][:type]` syntax is used. Note that the version is not specified, the version specified as a dependency will be used. If a type is not specified, a default value of `jar` is used.

Example 9.1. Artifact naming

```
com.acme:myapp          <dependency>
                        <groupId>com.acme</groupId>
                        <artifactId>myapp</artifactId>
                        <version>...</version>
                        </dependency>

com.acme:myapp:tar.gz   <dependency>
                        <groupId>com.acme</groupId>
                        <artifactId>myapp</artifactId>
                        <version>...</version>
                        <type>tar.gz</type>
                        </dependency>

com.acme:myapp:slave:tar.gz <dependency>
                        <groupId>com.acme</groupId>
                        <artifactId>myapp</artifactId>
                        <version>...</version>
                        <classifier>slave</classifier>
                        <type>tar.gz</type>
                        </dependency>
```

Tip

If the plugin can't find the artifact that's referred to it will list all available artifacts.

9.1.2. File renaming

All operations that involves moving a set of files around support renaming the files before putting them in the package.

The renaming process is controlled by two attributes:

`pattern` A regular expression that selects the files to be renamed. The regular expression may contain groups to pick out parts of the string.

The syntax used is the standard Java syntax. See the reference documentation on Pattern [<http://java.sun.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html>].

`replacement` The new name of the file. May contain references to groups matched in the `pattern`.

Warning

Document the effect on the matched files vs those that doesn't match

Example 9.2. Removing the first directory by renaming the files

```
<extract-artifact>
  <artifact>org.mortbay.jetty:jetty-assembly:zip</artifact>
  <to>/opt/jetty</to>
  <pattern>/jetty- $\{jetty.version\}$ (.*)</pattern>
  <replacement> $\$1$ </replacement>
</extract-artifact>
```

This example will match all files in the `jetty-assembly` artifact, put the entire path except the first part into group number one. The replacement value will be the value of the first group.

9.1.3. Includes and Excludes

Warning

Explain the difference between:

- `*.foo`
- `**/*.foo`

Warning

The order of includes vs excludes

Warning

Explain the effect of having a `basedir`-like parameter set when calculating matches. (the `include/exclude` expressions can't contain the `basedir` part). Applies only to set attributes for `now`.

9.1.4. File Attributes

All file objects has a set of attributes. Note: not all attributes applies to all object types.

Note

Some assembly operations support both per-directory and per-file attributes. If so the outer tag will be named `<fileAttributes>` and `<directoryAttributes>`, but they contain exactly the same set of elements.

Table 9.2. Available attributes

user	Specifies the user of the owns the file
group	Specifies the group that owns the file
mode	Specifies the read/write/executable bits on the file. The value has to be in octal notation

Mode examples:

- 0644 becomes `-rw-r--r--`
- 0744 becomes `-rwxr--r--`

See Section 2.4, “Package File System and File System Objects” for more details.

Example 9.3. Example usage of `<attributes>`

```
<attributes>
  <user>myapp</user>
  <group>myapp</group>
  <mode>0644</mode>
</attributes>
```

9.2. Copy File

Purpose: to copy a single file from the file system.

Table 9.3. Supported parameters for `<copyFile>`

file	The file to copy
toFile or toDir	The destination file or directory. Only one of the two parameters may be specified. If <code>toDir</code> is used, a file will be created in the specified directory with the same name as the source file.
attributes	The attributes to set on the copied file. See Section 9.1.4, “File Attributes”.

Example 9.4. Example usage of `<copyFile>`

```
<copyFile>
  <file>src/main/native/myapp.so</file>
  <toFile>/opt/myapp/myapp.so</toFile>
  <attributes>
    <user>myapp</user>
    <group>myapp</group>
    <mode>0644</mode>
  </attributes>
</copyFile>
```

9.3. Copy Artifact

Purpose: to copy an artifact from a Maven repository.

See also: Copy File

Table 9.4. Supported parameters for <copyArtifact>

artifact	The artifact to copy. See Section 9.1.1, “Artifact Naming and Identification” on how to identify the artifact to copy.
toFile or toDir	The destination file or directory. Only one of the two parameters may be specified. If toDir is used, a file will be created in the specified directory with the same name as the source file.
attributes	The attributes to set on the copied file. See Section 9.1.4, “File Attributes”.

Example 9.5. Example usage of <copyArtifact>

```
<copyArtifact>
  <artifact>org.jvnet.hudson.main:hudson-war:war</artifact>
  <toFile>/opt/hudson/hudson.war</toFile>
  <attributes>
    <user>hudson</user>
    <group>hudson</group>
    <mode>0644</mode>
  </attributes>
</copyArtifact>
```

9.4. Make Directory

Purpose: to create one or more directories

Table 9.5. Supported parameters <makeDirectory>

path or paths	The directory or directories to create. Only one of these parameters can be specified.
attributes	The attributes to set on the copied file. See Section 9.1.4, “File Attributes”.

Example 9.6. Example usage of <mkdirs>

```
<mkdirs>
  <paths>
    <path>/var/opt/jetty</path>
    <path>/var/opt/jetty/cache</path>
    <path>/var/opt/jetty/log</path>
  </paths>
  <attributes>
    <user>jetty</user>
    <group>jetty</group>
  </attributes>
</mkdirs>
```

9.5. Set Attributes

Purpose: to change one or more attributes on a set of files

Table 9.6. Supported parameters <setAttributes>

basedir	The base directory when applying. Default: the root of the package.
---------	---

fileAttributes	The attributes to set on the matched <i>files</i> . See Section 9.1.4, “File Attributes”.
directoryAttributes	The attributes to set on the matched <i>directories</i> . See Section 9.1.4, “File Attributes”.
includes and excludes	Selects which files to include into the package. See Section 9.1.3, “Includes and Excludes”.

Example 9.7. Example usage of <setAttributes>

```
<setAttributes>
  <basedir>/usr/share/hello/bin</basedir>
  <fileAttributes>
    <user>foo</user>
    <group>bar</group>
    <mode>0755</mode>
  </fileAttributes>
</setAttributes>
```

9.6. Symlink

Purpose: to create symbolic links.

Table 9.7. Supported parameters <symlink>

path	The file object to create in the package
value	The value of the link

Example 9.8. Example usage of <symlink>

```
<symlink>
  <path>/var/log/myapp</path>
  <value>/var/opt/myapp/log</value>
</symlink>
```

This will create a symbolink link under /var/log/myapp that points to /var/opt/myapp/log.

9.7. Copy Directory, Extract Artifact and Extract File

These operation work in a similar fasion and share these attributes:

Table 9.8. Common parameters for the copy directory, extract artifact and extract file operations

to	The base destination directory
includes and excludes	Selects which files to include into the package. See Section 9.1.3, “Includes and Excludes”.
pattern and replacement	Controls renaming of the files. See Section 9.1.2, “File renaming”.
fileAttributes	The attributes to set on the matched <i>files</i> . See Section 9.1.4, “File Attributes”.
directoryAttributes	The attributes to set on the matched <i>directories</i> . See Section 9.1.4, “File Attributes”.

9.7.1. Copy Directory

Purpose: to copy a directory structure.

Table 9.9. Additional parameters for <copyDirectory>

from	The directory to copy from.
------	-----------------------------

Example 9.9. Example usage of <copyDirectory>

```
<copyDirectory>
  <from>target/appassembler</from>
  <to>/usr/share/hello</to>
</copyDirectory>
```

9.7.2. Extract Artifact

Purpose: to extract an artifact from the repository into the package.

Table 9.10. Additional parameters for <extractArtifact>

artifact	The artifact to copy. See Section 9.1.1, “Artifact Naming and Identification” on how to identify the artifact to copy.
----------	--

Warning

document the supported archive types. (at least zip, jar and war are supported).

Example 9.10. Example usage of <extractArtifact>

```
<extractArtifact>
  <artifact>org.mortbay.jetty:jetty-assembly:zip</artifact>
  <to>/opt/jetty</to>
</extractArtifact>
```

9.7.3. Extract File

Purpose: to extract a file from the file system into the package.

Warning

document the supported archive types. (at least zip, jar and war are supported).

Table 9.11. Additional parameters for <extractDile>

archive	The path to an archive to extract. See Section 9.1.1, “Artifact Naming and Identification” on how to identify the artifact to copy.
---------	---

Example 9.11. Example usage of <extractArtifact>

```
<extractFile>
  <archive>src/main/extras.zip</archive>
  <to>/opt/share/myapp/extras</to>
</extractFile>
```

Chapter 10. Creating Native Package Repositories

10.1. Creating Debian/APT Repositories

10.2. Creating RPM/Yum Repositories

10.3. Creating pkg-get/pkgutil Repositories

Chapter 11. Troubleshooting

11.1. Debugging

The plugin has its own mechanism to be more verbose to make it easier to just debug the plugin. By running Maven with the **-Dmaven.unix.debug=true** flag you will get a lot of extra debugging information. In particular you will get information about all the assembly operations it has collected per package and all output from any external command it will use while building the package.

Part IV. Useful Tips For Making Useful Packages

Chapter 12. Package Command Reference

This part describes a set of commands that are useful for looking into and disassembling packages.

12.1. `dpkg` Commands

12.1.1. Listing All Files In a Package

Example 12.1. Output of `rpm -q -l -p`

```
$ rpm -q -l -p target/basic-*.rpm
/opt
/opt/hudson
/opt/hudson/doc
/opt/hudson/doc/atom-license.txt
/opt/hudson/doc/dc-license.txt
/opt/hudson/hudson.war
/var
/var/log
/var/log/hudson
```

12.2. `rpm` Commands

12.2.1. List All Files In a Package

Example 12.2. Output of `rpm -q -l -p`

```
$ rpm -q -l -p target/basic-*.rpm
/opt
/opt/hudson
/opt/hudson/doc
/opt/hudson/doc/atom-license.txt
/opt/hudson/doc/dc-license.txt
/opt/hudson/hudson.war
/var
/var/log
/var/log/hudson
```

12.2.2. List All Files In a Package With File Permissions

Example 12.3. Output of `rpm -q -l -p -v`

```
$ rpm -q -v -l -p target/basic-*.rpm
drwxr-xr-x 2 nobody nogroup 0 May 13 21:27 /opt
drwxr-xr-x 2 nobody nogroup 0 May 13 21:27 /opt/hudson
drwxr-xr-x 2 nobody nogroup 0 May 13 21:27 /opt/hudson/doc
-rw-r--r-- 1 nobody nogroup 49 Oct 2 2008 /opt/hudson/doc/atom-
license.txt
-rw-r--r-- 1 nobody nogroup 1544 Oct 2 2008 /opt/hudson/doc/dc-license.txt
-rw-rw-rw- 1 hudson hudson 20623413 Oct 24 2008 /opt/hudson/hudson.war
drwxr-xr-x 2 nobody nogroup 0 May 13 21:27 /var
drwxr-xr-x 2 nobody nogroup 0 May 13 21:27 /var/log
lrwxrwxrwx 1 nobody nogroup 19 May 13 21:27 /var/log/hudson -> /var/opt/
hudson/log
```

12.2.3. Listing All Available Software Groups

This is useful select an appropriate category when configuring the Group setting for RPMs.

Example 12.4. Listing all available software groups

```
$ cat /usr/share/doc/rpm-*/GROUPS
Amusements/Games
Amusements/Graphics
Applications/Archiving
Applications/Communications
Applications/Databases
Applications/Editors
Applications/Emulators
Applications/Engineering
Applications/File
Applications/Internet
Applications/Multimedia
Applications/Productivity
Applications/Publishing
Applications/System
Applications/Text
Development/Debuggers
Development/Languages
Development/Libraries
Development/System
Development/Tools
Documentation
System Environment/Base
System Environment/Daemons
System Environment/Kernel
System Environment/Libraries
System Environment/Shells
User Interface/Desktops
User Interface/X
User Interface/X Hardware Support
```

12.3. zip Commands

Note that there are no standard for the utilities used to handle zip files. However, most platforms support this command.

Example 12.5. Listing all files in a ZIP file

```
$ unzip -l target/basic-*.zip
Archive: target/basic-1.0-SNAPSHOT.zip
  Length      Date    Time    Name
-----
         0  04-07-11 13:33  ./opt/
         0  04-07-11 13:33  ./opt/hudson/
         0  04-07-11 13:33  ./opt/hudson/doc/
        49  10-02-08 02:07  ./opt/hudson/doc/atom-license.txt
       1544  10-02-08 02:07  ./opt/hudson/doc/dc-license.txt
    20623413  09-30-10 10:12  ./opt/hudson/hudson.war
-----
    20625006                          6 files
```

Chapter 13. Snippets for Scripts

13.1. RedHat/Fedora

13.1.1. Making your server start on boot

If you want your application to start when the machine starts you can use the `chkconfig`. Your script has to be `chkconfig-compatible` and installed under `/etc/init.d/foo`. The script has to include this line:

```
# chkconfig: 345 10 90
```

This will make the system start the application in runlevels 3, 4 and 6 with the priority 10 and 90 at shutdown and start, respectively. To install such a script, simply run

```
chkconfig --add foo
```

Similarly, if you want to remove it when the package is uninstalled simply run:

```
chkconfig --del foo
```

13.1.1.1. See Also

- http://docs.fedoraproject.org/en-US/Fedora/12/html/Deployment_Guide/s1-services-chkconfig.html
- <http://linux.die.net/man/8/chkconfig>

Part V. Version History and Upgrading

Preface to Part IV

This part document the major features that was introduced in the different versions and instructions on how to upgrade between the different alpha and beta releases.

Once the plugin stabilizes expect the upgrade instructions to be removed as the pre releases will be unsupported.

Chapter 14. 1.0-alpha-4

14.1. Change Log

Major features added:

- Support for creating "zip" archives
- Added support for classes for PKG
- %doc and %config for RPM
- Lots of documentation, including a PDF

JIRA issues fixed:

- MUNIX-18: Simplify the script mechanism
- MUNIX-17: Document how versions and revisions are calculated
- MUNIX-16: Document how the scripts directories work
- MUNIX-15: Improve error message when an artifact is not found
- MUNIX-12: Add support for creating zip files
- MUNIX-6: set-attributes doesn't pick up paths that aren't explicitly created with mkdirs
- MUNIX-4: Add support for the %doc directive for RPM
- MUNIX-3: Add support for the %config directive for RPM
- MUNIX-1: the "release" portion of the rpm does not have to be an int

14.2. Upgrading from 1.0-alpha-3

The `<softwareGroup>` element within the RPM configuration tag (`<rpm>`) has been renamed to `<group>` to match the field in the RPM spec file more closely.

The tag `<id>` has been renamed to `<classifier>` to be closer to the existing Maven nomenclature.

All examples use hyphenated elements instead of camel casing to be more consistent with how Maven POMs normally are written. This will not break any builds as Maven interpret both version the same way. For example `<fileAttributes>` is now used instead of `<file-attributes>`.