

Maven

1. Introducción

Maven es un Project Management Framework, esto es, un framework de gestión de proyectos de software.

Creada por Jason van Zyl en 2002, estuvo integrado inicialmente dentro del proyecto Jakarta, y ahora ya es un proyecto de nivel superior de la Apache Software Foundation.

Cada proyecto tiene la información para su ciclo de vida en el descriptor xml (por defecto el fichero pom.xml, basado en "project object model")

Con Maven vamos a poder:

- Compilar
- Empaquetar
- Generar documentación
- Pasar los test
- Preparar las builds

Maven estandariza un ciclo de vida para los proyectos, proporcionando un marco que permita la reutilización fácil de todos los componentes (artefactos) definidos bajo la estructura de Maven.

Los artefactos de Maven se ubican en repositorios, desde donde los proyectos Maven pueden accederlos.

2. Instalación

Se puede descargar la distribución desde [aquí](#)

No tiene instalación, simplemente descomprimir la distribución en la ubicación deseada.

Es recomendable definir la variable de entorno JAVA_HOME

```
JAVA_HOME="c:\java\jdk1.6.0_23"
```

Incluso incluir los ejecutables de Maven en el PATH, para que el comando **mvn** este accesible desde cualquier ubicación.

```
PATH = %PATH%; "c:\program files\maven-2.2.1\bin"
```

Todos los recursos que maneja Maven, son conocidos como **Artefactos** y se descargan desde un repositorio remoto al repositorio local, este repositorio local, se encuentra por defecto en

- **\$HOME/.m2** en unix/linux
- **C:/Users/{nombre-usuario}/.m2** en windows
- **~/.m2** en Mac

Se puede modificar dicha ubicación modificando el **settings.xml**, incluyendo

```
<localRepository>${ruta_nueva_repositorio_local}</localRepository>
```

3. Uso

Para emplear Maven, se ha de lanzar el comando mvn sobre un proyecto Maven, es decir un proyecto con un fichero **pom.xml** valido

```
mvn clean
```

Sobre el comando se pueden aplicar modificadores como **-U**, que fuerza la descarga de las dependencias.

```
mvn clean -U
```

4. Características

- Creación sencilla y ágil de un nuevo proyecto.
- Estandarización de la estructura de un proyecto. El proyecto se describe en su totalidad en el fichero pom.xml.
- Potente mecanismo de gestión de las dependencias, y la resolución de dependencias transitivas.
- Gestión simultánea de varios proyectos.
- Repositorio de librerías Open Source actualizado.
- Extensible, dispone de multitud de plugins y de la posibilidad de creación de otros que necesitemos.
- Acceso inmediato a nuevas funcionalidades requiriendo un mínimo esfuerzo.
- Integración con tareas ANT.
- Generación de diversos formatos de empaquetado de proyectos: WAR, EAR, JAR...
- Generación de un portal Web del proyecto. Incluyendo documentación del proyecto, informes del estado del proyecto, calidad del código.
- Gestión de releases y publicación. Integración con sistemas de gestión de versiones (como CVS o

SVN).

5. Arquetipos

El primer comando a conocer es **archetype**, ya que permite la creación de un proyecto (artefacto) Maven, siguiendo una plantilla, la cual habra que seleccionar, además de indicar otros parametros en una serie de pasos

- Elección del arquetipo.
- Versión del arquetipo.
- GroupId.
- ArtifactId.
- Version.
- Package.

```
mvn archetype:generate
```

Se puede indicar los valores concretos del arquetipo a emplear con los parametros

- **-DarchetypeGroupId**
- **-DarchetypeArtifactId**
- **-DarchetypeVersion**

Y los valores para el nuevo artefacto con

- **-DgroupId**
- **-DartifactId**

```
mvn archetype:generate -DarchetypeGroupId=com.curso.ecosistema  
-DarchetypeArtifactId=Miarquetipo -DarchetypeVersion=0.0.1-SNAPSHOT  
-DgroupId=com.curso.ecosistema.proyecto.con.arquetipo  
-DartifactId=ProyectoAPartirDeArquetipo -DarchetypeCatalog=local
```

El listado de arquetipos, se puede consultar [aqui](#)

Por ejemplo

- El 799, que es el por defecto, crea un proyecto java con una clase de código y de test de ejemplo.

Se pueden crear nuevos arquetipos, ya que no son mas que la foto de un proyecto en un momento concreto, para ello, se ha de generar el proyecto Maven, con todas aquellas configuraciones y codigos

deseados y se lanza el comando

```
mvn archetype:create-from-project
```



En las últimas versiones, el comando puede dar el error siguiente

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-archetype-  
plugin:2.4:create-from-project (default-cli) on project  
CreadoConArquetipo: Error configuring command-line. Reason: Maven  
executable not found at: D:\utilidades\apache-maven-3.3.9\bin\mvn.bat ->  
[Help 1]
```

Este error, se produce porque en las últimas versiones de Maven para Windows, se ha sustituido el fichero **mvn.bat**, por **mvn.cmd**, en realidad el contenido es equivalente, por lo que para evitar este error, simplemente habrá que copiar y pegar el fichero renombrando la copia.s

Una vez ejecutado el comando de creación del arquetipo basandose en el proyecto, en la carpeta **/target/generated-sources/archetype** se tiene el código fuente del arquetipo, la instalación del arquetipo será igual que para cualquier otro proyecto Maven, con los comandos **mvn install** o **mvn deploy** dependiendo de si es en el repositorio local o remoto/empresa.

El proyecto generado, tiene como características

- **packaging.** maven-archetype

Como porceso de la instalación del arquetipo en el repositorio local, se define el fichero **\.m2\archetype-catalog.xml**, siguiendo el siguiente esquema, en el que se incluyen todos los arquetipos disponibles de forma local.

```
<archetype-catalog
  xmlns="http://maven.apache.org/plugins/maven-archetype-plugin/archetype-
catalog/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-archetype-plugin/archetype-
catalog/1.0.0 http://maven.apache.org/xsd/archetype-catalog-1.0.0.xsd">

  <archetypes>
    <archetype>
      <groupId>com.curso.ecosistema</groupId>
      <artifactId>MiArquetipo</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <description>Mi Arquetipo</description>
    </archetype>
  </archetypes>
</archetype-catalog>
```

Una vez creado el catalogo, se puede acceder crear un artefacto nuevo basado en el arquetipo ejecutando

```
mvn archetype:generate -DarchetypeCatalog=local
```

Con este comando se listan los arquetipos definidos en el catalogo local.

6. Compatibilidad con Eclipse

El comando **eclipse**, permite generar dentro del proyecto Maven, los ficheros de configuración del IDE Eclipse, para que este reconozca el proyecto.

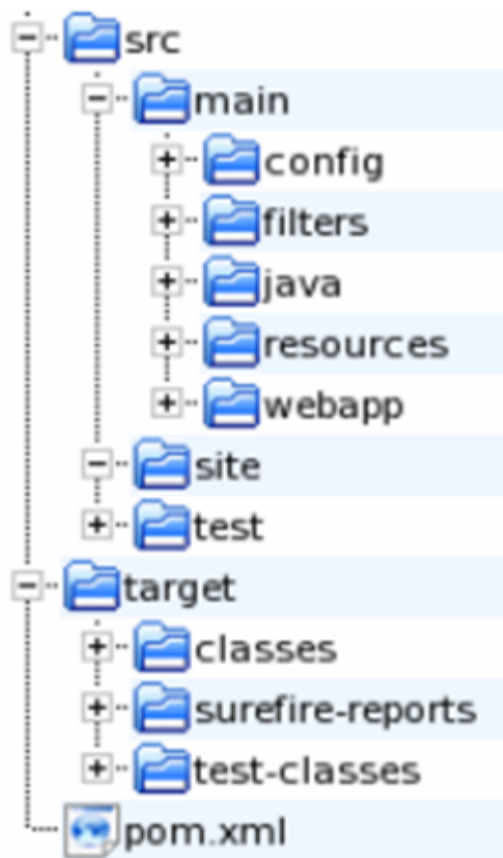
```
mvn eclipse:eclipse
```

Esta misma situación, a la inversa, se puede realizar desde el propio IDE, transformando un proyecto eclipse no Maven en un proyecto Maven, con la opción **Configure** → **Convert To Maven Project**.

Para que aparezca esta opción, se ha de tener instalado en Eclipse, un plugin de Maven, afortunadamente desde las últimas versiones de eclipse, este viene integrado por defecto.

7. Estructura del proyecto

Los proyectos de Maven, tienen una estructura por defecto, que puede ser cambiada a través del **pom.xml**. La estructura por defecto es la siguiente.



Algunas de las carpetas mas importantes que componen esta estructura son

- **src/main/java**: Código Fuente
- **src/main/resources**: Recursos no compilables necesarios en tiempo de ejecución,
- **src/main/webapp**: páginas, tags, etc.
- **src/main/webapp/WEB-INF**: Contiene el web.xml y otros ficheros de configuración.
- **src/test/java**: Código Fuente de pruebas.
- **src/test/resources**: Recursos no compilables necesarios para las pruebas.
- **src/site**: Carpeta con información para generar el sitio web HTML con la información del proyecto.
- **target**: Carpeta donde se guardan los resultados.

8. Configuración de un proyecto (pom.xml)

Se encuentra siempre en la raíz del proyecto y contienen toda información del proyecto.

Los proyectos tiene que definir obligatoriamente tres propiedades, estas son

- **artifactId**: Nombre del artefacto.
- **groupId**: texto que engloba a varios artefactos, puede ser la empresa que los genera o el proyecto en el que se enmarcan, suele adoptar forma de paquete java.

- **version:** Normalmente se establece un numero, aunque no es obligatorio.

A traves de estas tres características se puede hacer referencia a un artefacto Maven de forma univoca.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ejemplo.maven</groupId>
  <artifactId>HolaMundo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

Además se puede definir otras propiedades como

- **name:** Nombre del proyecto
- **packaging:** Formato de distribución del artefacto, por defecto es **jar**, aunque puede tomar como valores **ejb**, **ear**, **war** o **pom**.
- **description:** Una descripción del proyecto, empleada por Maven en el Sitio.
- **developers:** Personas que contribuyen al proyecto
- **license:** Tipo de licencias que afectan al proyecto
- **organization:** Empresa detras del proyecto
- **url:** Ubicación del Sitio en internet.
- **ciManagement:** Permite configurar como el servidor de Integración Continua (CI), comunica el estado de las tareas.
- **pluginRepositories:** Permite configurar nuevos repositorios para ampliar los plugin disponibles.
- **profiles:** Permiten definir perfiles de uso del Pom, a traves de la inclusion de variables en la configuración.
- **repositories:** Permite configurar nuevos repositorios para ampliar los artefactos disponibles.
- **build:** Seccion en la que se configura la construccion del proyecto.
- **dependencies:** Seccion donde se configuran las dependencias del proyecto con otros artefactos.
- **scm:** Permite configurar la ubicación del getor de versiones de codigo fuente, para realizar de forma automatica el etiquetado de release.
- **distributionManagement:** Permite configurar la ubicación del repositorio de artefactos maven, donde se instalarán las versiones del proyecto para su distribución.

9. Configuración del usuario (settings.xml)

Es un fichero que se encuentra ubicado en el directorio **.m2**, al mismo nivel que se encuentra por defecto el repositorio local.

En el se pueden configurar:

- **mirror**: Ubicaciones de repostorios, pueden ser los centrales de Maven indicando la propiedad **mirrorOf** con **central** o ***** para que el mirror lo sea de todos los servidores.

```
<mirrors>
  <mirror>
    <id>mirrorId</id>
    <mirrorOf>repositoryId</mirrorOf>
    <name>Human Readable name for this Mirror.</name>
    <url>http://my.repository.com/repo/path</url>
  </mirror>
</mirrors>
```

- **proxy**: Permite configurar como acceder a la red a Maven.

```
<proxies>
  <proxy>
    <id>optional</id>
    <active>true</active>
    <protocol>http</protocol>
    <username>proxyuser</username>
    <password>proxypass</password>
    <host>proxy.host.net</host>
    <port>80</port>
    <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
  </proxy>
</proxies>
```

10. Ciclo de Vida

Un ciclo de vida en Maven, esta compuesto por un conjunto de fases que se ejecutan de forma secuencial.

Existen tres ciclos de vida en maven.

- Default
- Clean
- Site

El ciclo de vida **Clean**, se compone de las siguientes fases.

- **pre-clean**: Se ejecutan los procesos necesarios antes de poder limpiar el proyecto.
- **clean**: Elimina todos los ficheros generados desde la anterior construcción, borra el contenido de `/target`
- **post-clean**: Se ejecutan los procesos necesarios para finalizar la limpieza del proyecto.

Se ejemplo de invocación sería con el comando

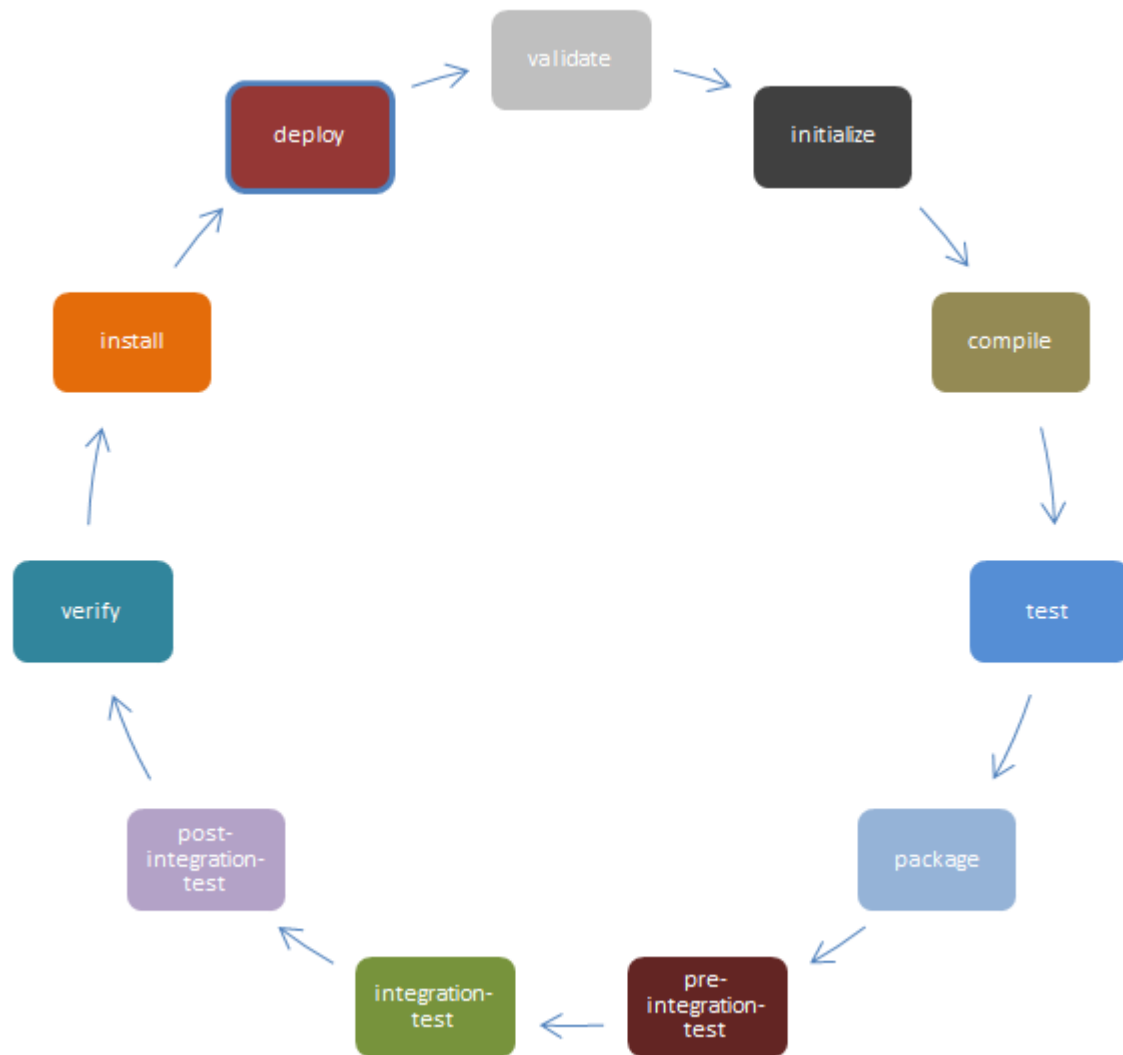
```
mvn clean
```

El ciclo de vida **Default**, se compone de las siguientes fases.

- **validate**: Valida que el proyecto es correcto y que esta toda la información necesaria.
- **initialize**: Inicializa el estado de construcción, creando propiedades y directorios.
- **generate-sources**: genera todos las fuentes de código a incluir en la compilación.
- **process-sources**: procesa los fuentes de código, aplicando filtros si los hubiera.
- **generate-resources**: genera recursos para incluir en el paquete.
- **process-resource**: Copia y procesa todos los recursos en el directorio destino, listos para empaquetarlos.
- **compile**: Compila el código fuente del proyecto.
- **process-classes**: post-procesa los ficheros generados en la compilación, para mejorarlos.
- **generate-test-sources**: genera los fuentes de código de test a incluir en la compilación.
- **process-test-sources**: procesa los fuentes de código de test, aplicando filtros si los hubiera.
- **generate-test-resources**: genera recursos para incluir en los test.
- **process-test-resources**: Copia y procesa todos los recursos en el directorio destino.
- **test-compile**: Compila el código fuente de los test del proyecto.
- **process-test-classes**: post-procesa los ficheros generados en la compilación de los test, para mejorarlos.
- **test**: ejecuta los test. El código de los test no se incluye en el paquete.
- **prepare-package**: prepara todo lo necesario antes de empaquetar.
- **package**: empaqueta el código compilado.
- **pre-integration-test**: realiza todo lo necesario para poder ejecutar los test de integración.
- **integration-test**: procesa y desliega si es necesario el paquete en el entorno de ejecución de los test de integración.
- **post-integration-test**: realiza todo lo necesario para limpiar el entorno de ejecución de los test de

integración.

- **verify**: verifica que el paquete es valido y cumple los criterios de calidad.
- **install**: instala el paquere en el repositorio local.
- **deploy**: instala el paquete en el repositorio remoto.



Se ejemplo de invocación sería con el comando

```
mvn deploy
```

El ciclo de vida **Site**, se compone de las siguientes fases.

- **pre-site**: execute processes needed prior to the actual project site generation
- **site**: generate the project's site documentation
- **post-site**: execute processes needed to finalize the site generation, and to prepare for site

deployment

- **site-deploy**: deploy the generated site documentation to the specified web server

Se ejemplo de invocación sería con el comando

```
mvn site
```

Se pueden ejecutar los ciclos enteros o de forma parcial hasta la fase deseada.

Se pueden incluir modificadores como

- **-U** → permite obligar a actualizar los artefactos del repositorio local, tanto **RELEASE** como **SNAPSHOT**

11. Dependencias

Representan aquellas librerías (jar) que el proyecto necesita en alguna fase del ciclo de vida.

```
<dependencies>
  <dependency>
    <groupId>commons-collections</groupId>
    <artifactId>commons-collections</artifactId>
    <version>3.2.1</version>
  </dependency>
</dependencies>
```

Se obtienen las dependencias de forma automática, bien del repositorio local si ya están, se han descargado previamente, o bien de un repositorio remoto, si nunca se han descargado.

Los proyectos que estan en desarrollo, pueden indicar esta situación en la version, añadiendo el sufijo **-SNAPSHOT**, que hará que Maven siempre compruebe el repositorio remoto, trabajando siempre con el jar mas nuevo.

Por efecto viene configurado un repositorio remoto unicamente, pudiendose añadir los que se quieran, por ejemplo un repositorio de empresa.

```
<repositories>
  <!-- Repositorio por defecto-->
  <repository>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <id>central</id>
    <name>Central Repository</name>
    <url>http://repo.maven.apache.org/maven2</url>
  </repository>
  <!-- Repositorio de empresa-->
  <repository>
    <id>repository-1</id>
    <url>http://repo.mycompany.com:8080/archiva/repository/internal/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

Las dependencias se resuelven de forma transitiva, esto quiere decir que si un proyecto depende de un segundo, y este a su vez de un tercero, el primer proyecto, depende automaticamente del tercero.

Cuando existen multiples dependencias, puede ocurrir que una dependencia de terceros, sea referenciada por otras dependencias en varias versiones, en este caso Maven, tambien gestiona con cual quedarse, el primer criterio que aplica es la profundidad en el arbol de dependencias, gana la dependencia que mas arriba esté, en segundo lugar aplica el orden de declaración de las dependencias en el **pom.xml**.

Tambien se pueden excluir dependencias transitivas, con motivo de la seleccion de la versión de la dependencia a emplear.

```
<dependencies>
  <dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>commons-dbcp</artifactId>
    <version>1.4</version>
    <exclusions>
      <exclusion>
        <artifactId>commons-pool</artifactId>
        <groupId>commons-pool</groupId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
```

Las dependencias pueden ser necesarias unicamente en una fase del ciclo de vida, para lo cual se permite la definicion del **scope** (ambito)

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Siendo los posibles valores del **scope**

- test: Solo disponible para las fases relacionadas con los test
- compile: Disponible en todo momento (es el por defecto)
- provided: O la JDK o el contendor proporcionan dicha dependencia.
- runtime: Disponible para ejecución o test, pero no para compilación.
- system: Similar a provided
- import: Permite importar configuraciones de proyectos de tipo pom. Solo se emplean sobre dependencias definidas en **dependencyManagement** con **type** pom.

En ocasiones se pueden tener librerias no desarrolladas con Maven, que para su inclusión en otros proyectos, seria recomendable trasladar al mundo Maven, pero solo se dispone de los binarios, para ello, se puede hacer

```
mvn install:install-file
  -DgroupId=<grupo>
  -DartifactId=<artefacto>
  -Dversion=<versión>
  -Dpackaging=jar
  -Dfile=<ruta al archivo>
```

Un ejemplo para publicar en el repo local los drivers de Oracle, seria

```
mvn install:install-file -Dfile=ojdbc6.jar -DgroupId=oracle -DartifactId=ojdbc -Dversion=6.0.0 -Dpackaging=jar
```

Existen unas palabras reservadas, para obtener las últimas versiones de una artefacto en un repositorio

- **LATEST**: Retorna la última dependencia, sea esta RELEASE o SNAPSHOT.
- **RELEASE**: Retorna la última dependencia en estado RELEASE.

Se puede indicar la version de las dependencias como rangos, siendo

- [,] → inclusive.
- (,) → exclusivo.

Incluire la versión anterior a la 1.2

```
<dependency>
  <groupId>commons-logging</groupId>
  <artifactId>commons-logging</artifactId>
  <version>[1.0,1.2)</version>
</dependency>
```

Si los extremos quedan vacios, indica que es desde el minimo o hasta el maximo.

Incluirá la última version del artefacto.

```
<dependency>
  <groupId>commons-logging</groupId>
  <artifactId>commons-logging</artifactId>
  <version>[1.0,)</version>
</dependency>
```

Se puede controlar que dependencias son incluidas en el classpath, aunque estas sean referenciadas por otras relaciones, así como definir la versión deseada de una dependencia.

11.1. Gestion de Dependencias

Dado que en grandes proyectos modulares con muchas dependencias, resolver la colision entre versiones de dependencias puede ser una tarea compleja, Maven ofrece la posibilidad de trasladar esta complejidad a un unico sitio, el proyecto padre **pom**, y simplificar de esta forma el uso de las dependencias en los proyectos hijos.

La etiqueta **dependencyManagement** permite definir un preconfiguracion de las versiones de las dependencias, que los proyectos hijos pueden heredar.

En el proyecto padre, se establece toda la logica de versionado, exclusiones ...

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Y en los modulos hijos, se aprovecha dicha configuracion

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
  </dependency>
</dependencies>
```

Si la definicion se realiza en la etiqueta **dependencyManagement**, no implica que sea una dependencia, solo es la configuracion de la dependencia, para que sea dependencia del modulo, debe aparecer en **dependencies**

12. Herencia del pom

Se trata de establecer una relación entre varios proyectos, para obtener dos ventajas

- Unificar configuraciones repetidas en el proyecto padre.
- Manejar conjuntamente todos los proyectos hijos.

La idea es generar un poryecto **padre**, que será de tipo **pom**, y asociar a el los proyectos **hijos** como

modulos, con lo que se consigue que los comandos ejecutados sobre el padre, afecten también a los hijos.

```
<packaging>pom</packaging>
<modules>
  <module>proyecto_hijo</module>
</modules>
```

Y luego indicar en los proyectos **hijos** que tienen un proyecto **padre**, con lo que se consigue que las configuraciones del padre, sean heredadas por los hijos.

```
<parent>
  <groupId>paquete_inicial</groupId>
  <artifactId>proyecto_padre</groupId>
  <version>1.0-SNAPSHOT</version>
</parent>
```

Maven supone que los proyectos **hijos** (módulos), estarán dentro de la carpeta del proyecto **padre**, es decir en el mismo path que el **pom.xml** del **padre**, pero puede no ser así, para establecer dicha consideración, se debe configurar el parámetro **<parent><relativePath>** haciendo referencia a la ubicación del proyecto padre.

13. Profiles

Permiten definir conjuntos de variables a emplear en el **pom.xml**, seleccionando en cada momento que conjunto de variables emplear, así se puede reutilizar la misma configuración de Maven (el mismo pom.xml) para distintos entornos (desarrollo, pruebas, producción, ...).

La idea es definir el **profile** con una serie de configuraciones particulares para cada caso

En los **profile** se pueden definir

- **activation**. Permite definir condiciones por las cuales el perfil se activa. Las condiciones pueden ser
 - **activeByDefault**. Activa por defecto el profile con sus características.
 - **file**. Activa el profile si existe o no un fichero o directorio.

```
<activation>
  <file>
    <missing>target/generated-sources/axistools/wsdl2java/org/apache/maven</missing>
  </file>
</activation>
```

- **jdk.** Activa un profile si se emplea una version de la **jdk** particular, se pueden definir rangos.

```
<activation>
  <jdk>[1.3,1.6)</jdk>
</activation>
```

- **os.** Activa el profile si se esta en un Sistema Operativo dado.

```
<activation>
  <os>
    <name>Windows XP</name>
    <family>Windows</family>
    <arch>x86</arch>
    <version>5.1.2600</version>
  </os>
</activation>
```

- **property.** Activa el profile si existe una propiedad particular.

```
<activation>
  <property>
    <name>debug</name>
    <value>true</value>
  </property>
</activation>
```

Las propiedades se pueden pasar por linea de comandos

```
mvn install -Ddebug=true
```

- **properties.** Permite definir propiedades a emplear en la configuración de **Maven** cuando se emplee este *profile.

```
<profiles>
  <profile>
    <id>build-java-8</id>
    <properties>
      <java.version>1.8</java.version>
    </properties>
  </profile>
</profiles>
```

De definirse propiedades (variables), estas se referenciarán desde el resto del **pom.xml**, como por ejemplo

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>${java.version}</source>
        <target>${java.version}</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- **repositories**
- **pluginRepositories**
- **dependencies**
- **plugins**
- **modules**
- **reporting**
- **dependencyManagement**
- **distributionManagement**

Para seleccionar que perfil emplear.

```
mvn install -P build-java-8
```

Y para desactivar un **profile**, se emplea el caracter !

```
mvn install -P !build-java-8
```

Los perfiles pueden ser definidos bien en el **pom.xml** o bien en el **settings.xml** tanto a nivel del usuario o a nivel global de la distribución.

14. Gestion de Recursos

En las fases de **generate-source-resource** y **generate-test-resources**, se procesan los ficheros de

recursos del proyecto y de los test respectivamente, esto significa que se van a copiar a la carpeta donde se han puesto los compilados, para construir el classpath de los test por un lado y para su paquetización por otro

Se puede configurar la ruta donde encontrar los recursos de proyecto que no se han de compilar, para ello se tiene la etiqueta **resources**

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

De forma analoga se tiene la etiqueta **testresources** para los test.

A mayores, se pueden instrumentalizar todos los ficheros de texto incluidos en dicho directorio, empleando la sintaxis de las propiedades del pom **\${}**, para ello se ha de activar el filtrado

```
<properties>
  <mensaje.error>Mensaje desde el pom.xml</mensaje.error>
</properties>
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

Así se puede tener un fichero properties en el directorio **src/main/resources** parametrizado, que tome como valor de sus propiedades, constantes definidas en el pom.

```
mensaje=${mensaje.error}
```

Habr  que tener cuidado con la lectura del recurso, dado que no ser  en la carpeta de fuentes, sino en la de compilaci n.

```
Properties properties = new Properties();
//Carga del fichero de properties que se encuentra en la raiz del classpath
properties.load(this.getClass().getClassLoader().getResourceAsStream("propiedades.properties"));
```

15. Plugins

Permiten extender la funcionalidad de Maven, ya que son los que definen los **Goal** que se ejecutan en las distintas **Phases**.

Se puede indicar una configuración personalizada a través de la etiqueta **configuration**, la cual dependerá de cada uno de los **Plugin**

Los **Goal** definidos por los **Plugin**, se pueden

- Asociar a las **Phases** con lo que serán ejecutados al invocar dicha **Phase** como parte de un ciclo de vida
- O no asociar a ninguna **Phase**, con lo que solo se ejecutarán cuando se ejecute de forma explícita.

[Aquí](#) un listado de plugins.

16. Plugins Maven

16.1. Maven Compiler Plugin

Permite indicar que version de Java se va a emplear para la compilacion del codigo fuente

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

16.2. Maven War Plugin

Permite crear un War a partir del código del proyecto, se emplea siempre en Aplicaciones Web

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.2</version>
      <configuration>
        <warSourceDirectory>WebContent</warSourceDirectory>
        <webXml>WebContent\WEB-INF\web.xml</webXml>
      </configuration>
    </plugin>
  </plugins>
</build>
```

16.3. Maven Surefire Plugin

Permite ejecutar pruebas unitarias.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.7.1</version>
      <configuration>
        <excludes>
          <exclude>**/integracion/*.java</exclude>
        </excludes>
        <includes>
          <include>**/unitarias/*.java</include>
        </includes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

16.4. Maven Failsafe Plugin

Permite ejecutar pruebas de integración.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>2.8</version>
      <configuration>
        <excludes>
          <exclude>**/unitarias/*.java</exclude>
        </excludes>
        <includes>
          <include>**/integracion/*.java</include>
        </includes>
      </configuration>
      <executions>
        <execution>
          <id>pasar test integracion</id>
          <phase>integration-test</phase>
          <goals>
            <goal>integration-test</goal>
          </goals>
        </execution>
        <execution>
          <id>validar pruebas integracion</id>
          <phase>verify</phase>
          <goals>
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

17. Plugins Mojohaus

17.1. SQL Maven Plugin

Permite ejecutar sentencias SQL, utiles para preparar el entorno ante test de integración

```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>sql-maven-plugin</artifactId>
  <version>1.5</version>
  <dependencies>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derbyclient</artifactId>
      <version>10.11.1.1</version>
    </dependency>
  </dependencies>
  <configuration>
    <username>admin</username>
    <password>admin</password>
    <url>jdbc:derby://localhost:1527/parejas;create=true</url>
    <driver>org.apache.derby.jdbc.ClientDriver</driver>
    <settingsKey>derbyDB</settingsKey>
    <skip>${maven.test.skip}</skip>
  </configuration>
  <executions>
    <execution>
      <id>default-cli</id>
      <goals>
        <goal>execute</goal>
      </goals>
      <configuration>
        <autocommit>true</autocommit>
        <srcFiles>
          <srcFile>sql/create.sql</srcFile>
        </srcFiles>
        <onError>continue</onError>
      </configuration>
    </execution>
  </executions>
</plugin>

```

Mención especial al parametro de configuración **settingsKey**, que permite definir el **id** que se tomará como **server.id**, para no tener el user`password definido en el **pom.xml**, estando definidos en el **settings.xml** incluso encriptados.

Otro parametro interesante es **skip** que permite saltarse la ejecución de este **goal** de formar parte de un ciclo de vida.

A parte de definirse los **script** con ficheros, tambien pueden definirse directamente en el **pom.xml** con **sqlCommand**


```
<sqlCommand>drop database yourdb</sqlCommand>
```

Más información [aquí](#)

17.2. Exec Maven Plugin

Plugin que permite ejecutar comandos del sistema operativo.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.2</version>
  <configuration>
    <executable>java</executable>
    <arguments>
      <argument>-Dmyproperty=myvalue</argument>
      <argument>-classpath</argument>
      <!-- automatically creates the classpath using all project dependencies,
           also adding the project build directory -->
      <classpath/>
      <argument>com.example.Main</argument>
    </arguments>
  </configuration>
</plugin>
```



Ojo, porque para la ejecución de **.bat** puede no ser valido, ya que mantiene a la espera el proceso Maven hasta que el bat acabe.

Mas información [aquí](#)

18. Plugins Codehaus

18.1. Maven Cargo Plugin

Permite operar con distintos servidores (jetty, tomcat, Glassfish, JBoss, ...), en distintas modalidades (installed, remote, embeded, ...).

La página con la documentación del plugin [aquí](#)

Hay varias formas de configurar el plugin veamos dos

- Configuración para despliegue en remoto, donde se empleará el goal **deployer-deploy**.

```

<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.4.18</version>
  <configuration>
    <wait>true</wait>
    <container>
      <!-- Servidor a emplear, en este caso Tomcat 7 -->
      <containerId>tomcat7x</containerId>
      <!-- Tipo de servidor, en este caso un servidor remoto -->
      <type>remote</type>
    </container>
    <configuration>
      <type>runtime</type>
      <!-- Dado que se va a desplegar sobre un servidor ya arrancado y operativo,
se emplea la
funcionalidad de despliegue de war del propio servidor, a traves de su
consola, para lo que
hay que indicar los siguientes datos -->
      <properties>
        <cargo.remote.username>admin</cargo.remote.username>
        <cargo.remote.password>admin</cargo.remote.password>
        <cargo.servlet.port>8081</cargo.servlet.port>
      </properties>
    </configuration>
    <!-- Se define que se quiere desplegar, en este caso el propio proyecto, se puede
indicar de dos
formas, por groupId y artifactId, o por la ubicación del recurso-->
    <deployables>
      <deployable>
        <groupId>com.curso.ecosistema</groupId>
        <artifactId>ManejoPluginCargo</artifactId>
        <type>war</type>
        <!-- Se puede indicar la URL con la que validar el despliegue -->
        <pingURL>http://localhost:8081/ManejoPluginCargo/index.html</pingURL>
        <!--<location>${project.build.directory}/ManejoPluginCargo-0.0.1-
SNAPSHOT.war</location>-->
      </deployable>
    </deployables>
  </configuration>
</plugin>

```

- Configuración que permite arrancar, desplegar y parar un servidor en local. Con esta configuración, se pueden emplear los goals **start**, **stop** y **run**, este último permite arrancar el servidor y mantener en espera la ejecución.

```

<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.4.18</version>
  <configuration>
    <wait>true</wait>
    <container>
      <containerId>tomcat7x</containerId>
      <!-- Ubicacion del contenedor -->
      <home>D:\utilidades\apache-tomcat-7.0.67</home>
      <type>installed</type>
    </container>
    <configuration>
      <type>existing</type>
      <!-- Ubicacion a partir de la cual se creará la configuración de despliegue-->
    </configuration>
  </configuration>
  <properties>
    <cargo.servlet.port>8081</cargo.servlet.port>
  </properties>
</configuration>
<deployables>
  <deployable>
    <groupId>com.curso.ecosistema</groupId>
    <artifactId>ManejoPluginCargo</artifactId>
    <type>war</type>
    <pingURL>http://localhost:8081/ManejoPluginCargo/index.html</pingURL>
  </deployable>
</deployables>
</configuration>
</plugin>

```

19. Plugins de otros orígenes

19.1. Liquibase Maven Plugin

Maven dispone de un Plugin que permite integrar el lanzamiento de las tareas de **Liquibase** en el ciclo de vida del proyecto.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.liquibase</groupId>
      <artifactId>liquibase-maven-plugin</artifactId>
      <version>2.0.5</version>
      <configuration>
        <propertyFile>
src/main/resources/liquibase/liquibase.properties</propertyFile>
        <changeLogFile>src/main/resources/liquibase/master.xml</changeLogFile>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>update</goal>
          </goals>
          <phase>pre-integration-test</phase>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

El plugin se basa en la librería **Liquibase**, que permite automatizar tareas sobre motores de base de datos, la documentación oficial se puede encontrar [aquí](#)

A este plugin se le han de indicar

- Las propiedades de conexión, que en el ejemplo quedan definidas por el fichero **liquibase.properties**.

```

#liquibase.properties
driver: org.apache.derby.jdbc.ClientDriver
url: jdbc:derby://localhost:1527/Liquibase;create=true
username: root
password: root

```

- El fichero de ChangeLog, o lo que es lo mismo las tareas (script) a realizar sobre la base de datos, que en el ejemplo queda definida por **master.xml**.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog/1.9" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://www.liquibase.org/xml/ns/dbchangelog/1.9
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-1.9.xsd">
  <preConditions>
    <dbms type="mysql" />
    <runningAs username="root" />
  </preConditions>
  <changeSet author="root" id="1">
    <createTable tableName="Libros">
      <column name="titulo" type="VARCHAR(255)" />
      <column name="isbn" type="INT">
        <constraints nullable="false" primaryKey="true" />
      </column>
    </createTable>
  </changeSet>
</databaseChangeLog>
```

En este caso se está asociando a la fase de **pre-integration-test** la ejecución del goal **update** del plugin de liquibase que lanza la configuración.

19.1.1. Etiquetas Changelog

Algunas de las etiquetas que se pueden incluir en el fichero de **changelog** son

- `<databaseChangeLog>`: Etiqueta raíz.
- `<preConditions>`: Comprueba las condiciones definidas, si alguna falla se lanza error.
- `<changeSet>`: Define los cambios que se realizan en la base de datos. Se identifica por dos atributos, el **id** y el **author**.
- `<createTable>`: Permite crear nueva tabla en base de datos.

Existen mas etiquetas que permiten realizar las tareas mas habituales de gestión en la base de datos como `dropTable`, `dropColumn`, ... pero quizás las mas interesante sea `<sqlFile>`, que nos permite ejecutar un script SQL dentro del proceso de liquibase, por lo que no tenemos porque conocer el lenguaje propio de liquibase, solo el SQL.

```
<sqlFile path="/liquibase/master.sql" />
```

20. Repositorios de empresa

Un repositorio de empresa de Maven, es un repositorio de artefactos Maven privado, es decir con

securización, que habitualmente se emplea en un entorno empresarial para compartir dentro de la empresa y no con todo el mundo los artefactos que se van generando.

Existen varios repositorios

- Archiva. (Gratuito)
- Nexus. (De pago)
- Artifactory. (De pago barato)

Para la explicación vamos a emplear Archiva, para ello descargamos la versión stand-alone de [aquí](#)

Se descarga un zip, en cuyo interior encontramos un fichero **bin/archiva.bat**. Para arrancar se ejecuta el comando

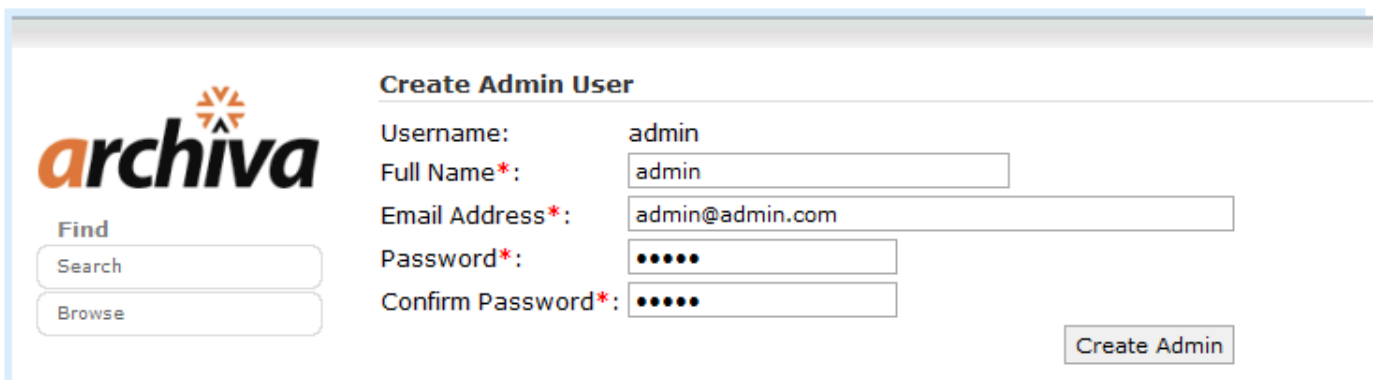
```
archiva console
```

Este comando arranca un jetty en el puerto 8080, publicando la interfaz sobre el repositorio, para acceder a la consola de administración

```
http://localhost:8080
```

Tanto el puerto (8080), como la base de datos, que por defecto es derby, se pueden modificar directamente en el fichero **conf/jetty.xml**

Una vez se accede a Archiva, se ha de dar de alta un administrador.



The screenshot shows the Archiva web interface. On the left is the Archiva logo and a search bar with 'Find', 'Search', and 'Browse' buttons. On the right is the 'Create Admin User' form with the following fields:

Create Admin User	
Username:	admin
Full Name*:	admin
Email Address*:	admin@admin.com
Password*:	•••••
Confirm Password*:	•••••

A 'Create Admin' button is located at the bottom right of the form.

Con este usuario habrá que generar los usuarios con permisos de despliegue, es decir los que tengan un rol **Repository Manager**.

Archiva

Welcome admin EDIT DETAILS LOGOUT Quick Search

ARTIFACTS
Search
Browse
Upload Artifact
ADMINISTRATION
Repository Groups
Repositories
Proxy Connectors
ProxyConnector Rules
Network Proxies
Repository Scanning
Runtime Configuration
System Status
UI Configuration
Reports
USERS
Manage
Roles
Users Runtime Configuration
DOCUMENTATION
REST Api
User Documentation

Users List

Users Edit

Edit Edit Roles

Username
Full Name
Password
Confirm Password
Email Address
Validated ☒
Locked ☐
Change password required ☐
Save Cancel

Effective Roles

- Registered User
- Repository Manager - internal
- Repository Manager - snapshots
- Repository Observer - internal
- Repository Observer - snapshots

Archiva

Welcome admin EDIT DETAILS LOGOUT Quick Search

ARTIFACTS
Search
Browse
Upload Artifact
ADMINISTRATION
Repository Groups
Repositories
Proxy Connectors
ProxyConnector Rules
Network Proxies
Repository Scanning
Runtime Configuration
System Status
UI Configuration
Reports
USERS
Manage
Roles
Users Runtime Configuration
DOCUMENTATION
REST Api
User Documentation

Users List

Users Edit

Edit Edit Roles

System

Roles that apply system-wide, across all of the applications

- ☐ Guest
- ☒ Registered User
- ☐ System Administrator
- ☐ User Administrator

Archiva

- ☐ Archiva Guest
- ☐ Archiva System Administrator
- ☐ Archiva User Administrator
- ☐ Global Repository Manager
- ☐ Global Repository Observer

Repository ManagerRepository Observer

snapshots ☒ ☒
internal ☒ ☒
Update

Una vez definidos los repositorios y los usuarios, se ha de configurar el proyecto para que emplee el repositorio de empresa.

Lo primero será definir el repositorio como nuevo repositorio para las descargas, para ello

```

<repositories>
  <repository>
    <id>archiva.internal</id>
    <url>http://localhost:8080/repository/internal/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>archiva.snapshots</id>
    <url>http://localhost:8080/repository/snapshots/</url>
    <releases>
      <enabled>>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>

```

La anterior configuración, no es necesaria si se desea que sea el repositorio de empresa el único repositorio accesible, es decir deshabilitar el acceso al repositorio **Maven Central**, para conseguirlo se ha de incluir el siguiente **Mirror** en el fichero **/.m2/settings.xml**, en lugar de la configuración anterior.

```

<mirror>
  <id>archiva.internal</id>
  <name>archiva.internal</name>
  <url>http://Victor-Portatil:8080/repository/internal</url>
  <mirrorOf>*</mirrorOf>
</mirror>

```



Los Mirror, permiten establecer repositorios alternativos a repositorios ya existentes, por si el repositorio configurado, no tuviese buena cobertura o simplemente no fuese accesible en algún caso.

De establecer * como valor de **mirrorOf**, se está anulando otras configuraciones de repositorios

El siguiente paso será definir los repositorios, como receptores de nuevas versiones, para ello a través de la etiqueta **<distributionManagement>**


```

<distributionManagement>
  <repository>
    <uniqueVersion>>false</uniqueVersion>
    <id>releases</id>
    <name>Releases</name>
    <url>http://Victor-Portatil:8080/repository/internal</url>
    <layout>default</layout>
  </repository>
  <snapshotRepository>
    <uniqueVersion>true</uniqueVersion>
    <id>snapshots</id>
    <name>Snapshots</name>
    <url>http://Victor-Portatil:8080/repository/snapshots</url>
    <layout>default</layout>
  </snapshotRepository>
</distributionManagement>

```

A tener en cuenta que en ambos dos casos anteriores, el identificador empleado para los repositorios es el mismo, esto es debido a la necesidad de identificación en el repositorio de empresa para subidas y descargas, dado que el acceso al servidor será restringido, con lo que se ha de configurar el **usuario/password** a emplear con dicho servidor, el lugar mas adecuado para configurar esto, dado que no va asociado al proyecto sino a usuario que maneja el proyecto, es el fichero **.m2\settings.xml**, aquí a través de un identificador del servidor, se asociará a dicho servidor el **usuario/password** a emplear.

```

<servers>
  <server>
    <id>archiva.internal</id>
    <username>despliegue</username>
    <password>despliegue1</password>
  </server>
  <server>
    <id>archiva.snapshots</id>
    <username>despliegue</username>
    <password>despliegue1</password>
  </server>
</servers>

```

21. Encriptado de Contraseñas

Para que Maven pueda realizar alguna de sus funcionalidades, se precisa que se autentique, casos como el despliegue de un nuevo artefacto en el repositorio de empresa de maven, o la descarga o subida de código al scm, para estos casos, se sugiere siempre que estas contraseñas, no vayan en el

pom.xml, sino en el **settings.xml** del usuario, para que nadie pueda acceder a dichas contraseñas, pero aun así, siempre es interesante poder encriptar esas contraseñas, para esto, **Maven** proporciona unos comandos **encrypt-master-password** y **encrypt-password**.

La idea, es generar primeramente una clave maestra, con el comando **encrypt-master-password**

```
mvn --encrypt-master-password <palabra maestra>
```

Esta sentencia mostrará una clave alfanumerica,

```
{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}
```

Esta clave se almacenará en **/.m2/settings-security.xml**, siguiendo la estructura

```
<settingsSecurity>
  <master>{jSMOWnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}</master>
</settingsSecurity>
```

Una vez generado el fichero **/.m2/settings-security.xml**, se procederá a generar los password encriptados a emplear por la aplicación

```
mvn --encrypt-password <password>
```

Las claves generadas se incluirán dentro del fichero **/.m2/settings.xml**, en la seccion de **servers**, solo en esta sección serán descriptadas.

```
<settings>
  <servers>
    <server>
      <id>identificador de servidor</id>
      <username>nombre de usuario</username>
      <password>{COQLCE6DU6GtcS5P=}</password>
    </server>
  </servers>
</settings>
```

Como caso especial, si se quiere emplear la encriptacion de password para el servidor **SCM**, y dado que la asociación entre servidor y clave, se hace a traves del **id**, y la etiqueta **scm** del **pom.xml** no acepta **id**, se ha de emplear el parametro **project.scm.id**, quedando el **/.m2/settings.xml**, como sigue

```
<settings>
  <servers>
    <server>
      <id>my-scm-server</id>
      <username>nombre de usuario</username>
      <password>{COQLCE6DU6GtcS5P=}</password>
    </server>
  </servers>
</settings>
```

Y el **pom.xml**

```
<project>
  <properties>
    <project.scm.id>my-scm-server</project.scm.id>
  </properties>
</project>
```