

MAVEN-2

MAVEN.

Personalizar información del proyecto

Configuración del entorno Maven

Dependencias

Ámbito

Dependencias propietarias

Uso de properties

Repositorio de Maven

Herencia y agregación

Creación de proyectos

Compiling code

Testing code

Packaging y deploying de una aplicación

MAVEN. Personalizar información del proyecto

Antes de empezar a escribir código, vamos a personalizar la información del proyecto.

Queremos añadir la siguiente información:

- Licencia del proyecto
- La organización
- Relación de desarrolladores vinculados al proyecto

Esta es toda la información estándar que se puede esperar ver en la mayoría de los proyectos.

MAVEN. Personalizar información del proyecto

Licencia del proyecto:

```
<licenses>
  <license>
    <name>Ulpgc</name>
    <url>http://...</url>
    <distribution>repo</distribution>
    <comments>Licencia
      de
      educacion</comments>
  </license>
</licenses>
```

MAVEN. Personalizar información del proyecto

Organización:

```
<organization>  
  <name>ULPGC</name>  
  <url>http://www.ulpgc.es</url>  
</organization>
```

MAVEN. Personalizar información del proyecto

Desarrolladores:

<developers>

<developer>

<name>Enrique Ramon</name>

<email>ejrbalma@gmail.com</email>

<url>http://www.ulpgc.es</url>

<roles>

<role>developer</role>

</roles>

<timezone>-1</timezone>

</developer>

<developer>

<name>Luisa Rodriguez</name>

<email>lr@gmail.com</email>

<url>http://www.ulpgc.es</url>

<roles>

<role>developer</role>

</roles>

<timezone>-1</timezone>

</developer>

</developers>

MAVEN. Configuración del entorno de Maven

El archivo `setting.xml` permite definir determinados elementos de configuración para trabajar con Maven.

En este archivo se Incluye cosas que no deben ser distribuidos con el archivo `pom.xml`, tales como la identidad desarrollador, junto con los ajustes locales, al igual que la información de proxy.

La ubicación predeterminada para el archivo de configuración es `~ / .m2 / settings.xml`.

Hay que crearlo expresamente.

Información a configurar:

Ubicación del Repositorio local

MAVEN. Configuración del entorno de Maven

Información a configurar:

- **localRepository**, ubicación del repositorio local.
- **interactiveMode**, Si Maven debe tratar de interactuar con el usuario para la entrada.
- **Offline**, Si Maven debe operar en modo offline todo el tiempo.
- **usePluginRegistry**, Si Maven debe utilizar el archivo plugin-registry.xml para gestionar versiones del plugin.
- Información de los proxy.
- Configuración de servidores específicos.
- Configuración de Repositorios espejos.
- La configuración de los perfiles de construcción de proyectos para ajustar con los parámetros de entorno.
- Lista de perfiles de construcción activados manualmente, especificados en el orden en que deben aplicarse.
- Lista de groupIds para buscar un plugin cuando no se proporciona explícitamente que groupId plugin.

MAVEN. Configuración del entorno de Maven

Configuración del entorno de Maven:

- Cuando utilizamos un proxy, para la conexión con internet.
- Usar un repositorio espejo en la organización
- Para cambiar la localización del repositorio local.
 <settings>
 <localRepository>c:\maven\repository\</localRepository>
 </settings>

Dependencias en Maven

Usando la herramienta de Dependencias Transitivas, Maven no solo localizará las librerías que se declaren, sino todas de las librerías que necesiten las dependencias que se han declarado.

Se declaran en la sección `<dependencies>` de pom.xml hemos de declarar la librería que tenemos que necesitamos compilar, test o run nuestra aplicación.

Las librerías serán recuperadas de los repositorios remotos o locales y cacheados en nuestra máquina local.

```
<dependencies>
  <dependency>
    <groupId>org.jfree</groupId>
    <artifactId>jfreechart</artifactId>
    <version>1.0.19</version>
  </dependency>
</dependencies>
```

Dependencias en Maven

Una librería puede tener diferentes versiones de librerías con el mismo número de versión por ejemplo misma versión pero para dos jdk distintos:

```
testing-5.1-jdk14.jar  
testing-5.1-jdk15.jar
```

Cuando declaramos la dependencia hemos de indicar exactamente qué versión necesitamos, para ello Maven proporciona el elemento *<classifier>*

```
<dependencies>  
  <dependency>  
    <groupId>org.testing</groupId>  
    <artifactId>testing</artifactId>  
    <version>5.1</version>  
    <classifier>jdk15</classifier>  
    <scope>test</scope>  
  </dependency>  
</dependencies>
```

Dependencias en Maven

Maven es flexible con el número de las versiones y podemos usar la notación de intervalos de la teoría de conjuntos.

```
(1,4)
[1,4]
[1,4)
[2,)
[1,5),(5,10]
```

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>[3.0,)</version>
  </dependency>
</dependencies>
```

Se requiere la última versión pero como mínimo la versión 3.0

```
<dependencies>
  <dependency>
    <groupId>commons-collection</groupId>
    <artifactId>commons-collection</artifactId>
    <version>[2.0,3.0)</version>
  </dependency>
</dependencies>
```

Ámbito de las dependencias

Scope de la dependencias, no siempre es necesario incluir todas las dependencias en el despliegue de la aplicación. Algunos Jar solo se necesitan para los test Unitarios, mientras que otros serán proporcionados en el tiempo de ejecución. Usando la técnica llamada de **dependency scoping** Maven permite usar los Jar cuando realmente se necesitan y excluirlos cuando no se necesitan.

Ámbito de las dependencias

compile: es la que tenemos por defecto si no especificamos scope. Indica que la dependencia es necesaria para compilar. La dependencia además se propaga en los proyectos dependientes.

provided: Es como la anterior, pero esperas que el contenedor ya tenga esa librería. Un claro ejemplo es cuando desplegamos en un servidor de aplicaciones, que por defecto, tiene bastantes librerías que utilizaremos en el proyecto, así que no necesitamos desplegar la dependencia.

runtime: La dependencia es necesaria en tiempo de ejecución pero no es necesaria para compilar.

Ámbito de las dependencias

test: La dependencia solo es para testing que es una de las fases de compilación con Maven. JUnit es un claro ejemplo de esto.

system: Es como provided pero hay que incluir la dependencia explícitamente. Maven no buscará este artefacto en tu repositorio local. Habrá que especificar la ruta de la dependencia mediante la etiqueta *<systemPath>*

Dependencias en Maven. Manejo dependencias propietarias

Por razones comerciales y derechos de propiedad no todas las librerías se encuentran en repositorios públicos.

Ejemplo:

el JDBC Driver de Oracle. Es público pero no está incluido en un repositorio Maven.

La librería Java Transaction API (JTA) requerida por Hibernate. Requiere que aceptes la licencia antes de poder descargarla.

Si necesitamos librerías propietarias, hemos de bajarlas manualmente a nuestro repositorio local.

Dependencias en Maven. Manejo dependencias propietarias

Como hacerlo:

Descargar el fichero Jar JDBC Driver de Oracle.

Es importante anotar exactamente la versión, a veces no es simple deducible del nombre del archivo. El número de versión lo utilizaremos para identificar el Jar en el repositorio.

La declaración de dependencia debería ser algo parecido a:

```
<dependency>
  <groupId>oracle</groupId>
  <artifactId>oracle-jdbc</artifactId>
  <version>10.1.0.2.0</version>
  <scope>runtime</scope>
</dependency>
```

Dependencias en Maven. Manejo dependencias propietarias

```
<dependency>
  <groupId>oracle</groupId>
  <artifactId>oracle-jdbc</artifactId>
  <version>10.1.0.2.0</version>
  <scope>runtime</scope>
</dependency>
```

Hemos de copiar el fichero Jar en el lugar adecuado en nuestro repositorio.

En nuestro Repositorio Local usando mvn install:install-file

```
$mvn install:install-file -Dfile = ojdbc7.jar \
  -DgroupId=oracle \
  -DartifactId=oracle-jdbc \
  -Dversion=12.1.0.2 \
  -Dpackaging=jar
```

Dependencias en Maven. Uso de properties

Refactorig las dependencias usando Properties.

En grandes proyecto incluso con los beneficios de las dependencias transitivas podemos acabar con un numero elevado de dependencias.

A veces es habitual declarar una clave con el número de versión, la forma de hacerlos es usando properties que se pueden definir el pom.xml:

- Son constantes que se pueden utilizar posteriormente.
- Son fáciles de mantener.
- Se suelen ubicar o bien al principio del fichero o al final del pom.xml

Dependencias en Maven. Uso de properties

Ejemplo:

Supongamos que estamos desarrollando una aplicación web usando **JavaServerPages** y JSTL API.

```
<properties>  
  <servlet-api.version>2.4</servlet-api.version>  
  <jstl.version>1.1.2</jstl.version>  
</properties>
```

Estas **properties** pueden ser utilizadas para declarar de forma más flexible dependencias.

Dependencias en Maven. Uso de properties

```
<properties>
  <servlet-api.version>2.4</servlet-api.version>
  <jstl.version>1.1.2</jstl.version>
</properties>
....
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>${servlet-api.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>${jstl.version}</version>
</dependency>
<dependency>
  <groupId>>taglibs</groupId>
  <artifactId>standard</artifactId>
  <version>${jstl.version}</version>
</dependency>
```

Dependencias en Maven. El repositorio de Maven.

Buscar Dependencias en con el repositorio de Maven

Acceder al repositorio de Maven. [Entrar](#)

Podemos estando en el repositorio:

- Ver el listado de las dependencias de una librería particular.
- Ver la última versión subida al repositorio
- Navegar por la estructura del repositorio.

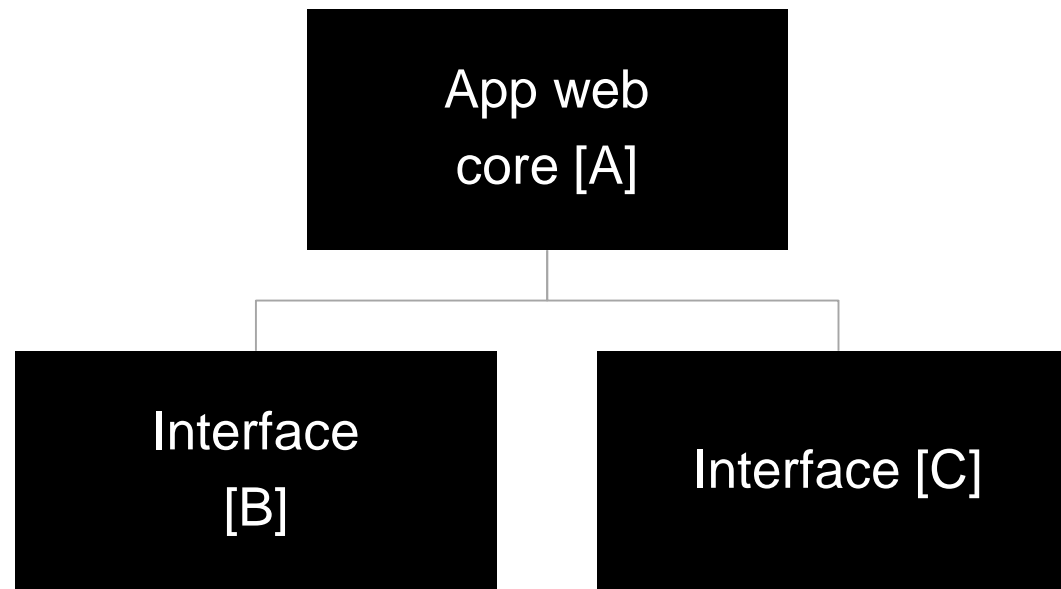
Dependencias en Maven. Herencia y Agregación

Maven fomenta escribir proyectos como un conjunto de pequeños y flexibles módulos, más que como un gran proyecto de código monolítico.

Las dependencias es una forma que permite crear relaciones entre módulos de un proyecto. Y la herencia es otro.

La herencia nos permite definir propiedades y valores que luego será heredados por los módulos hijos.

Dependencias en Maven. Herencia y Agregación



Dependencias en Maven. Herencia y Agregación

Los elementos de POM que son fusionado son los siguientes:

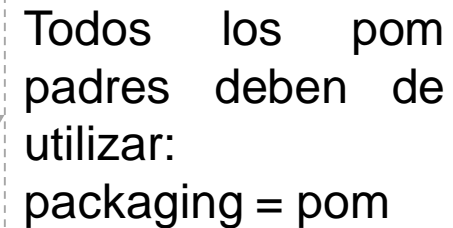
- Dependencias
- Desarrolladores y contribuidores
- Lista de plug-ins (incluyendo los informes)
- Plug-in de ejecución con
- Plug-in de configuración
- Recursos

Dependencias en Maven. Herencia y Agregación

Veamos como deberíamos implementar esta estructura

En el root-parent-pom.xml, que en el fondo es solo un pom.xml como otro cualquiera.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.company</groupId>
  <artifactId>myapp</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Killer application</name>
</project>
```

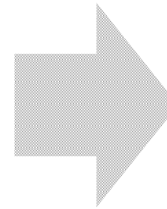


Todos los pom
padres deben de
utilizar:
packaging = pom

Dependencias en Maven. Herencia y Agregación

Cada uno de los hijos debe de declarar un elemento <parent>

```
<project>
<parent>
  <groupId>com.company</groupId>
  <artifactId>myapp</artifactId>
  <version>1.0</version>
</parent>
<modelVersion>4.0.0</modelVersion>
<artifactId>calculator-core</artifactId>
....
</project>
```



No hemos de definir:
<groupId>
<version>

Dependencias en Maven. Herencia y Agregación

El **parent** pom es un excelente lugar para definir **properties** y configuraciones de la construcción del proyecto.

Un uso típico es definir las opciones de compilación en un lugar central. Por ejemplo podemos poner que deseamos compilar según la versión java 1.5 y queremos que esto se herede a todos los módulos hijos y sin necesidad de realizar una configuración especial en cada uno de los módulos hijos.

Dependencias en Maven. Herencia y Agregación

```
<properties>
  <java-api.version>1.5</java-api.version>
</build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>${java-api.version}</source>
        <target>${java-api.version}</target>
      </configuration>
    </plugin>
  </plugins>
</build>
</properties>
```

Dependencias en Maven. Herencia y Agregación

En este nivel también podemos definir las dependencias.

```
<properties>
  <junit.version>4.4</junit.version>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Dependencias en Maven. Herencia y Agregación

Todos los proyectos hijos tendrán disponibles estas dependencias sin tener que incluirlas. También es una excelente forma de asegurarte que todos los proyectos hijos tienen la misma versión que determinadas APIs.

Otro uso que se le suele dar al pom padre es que incluya la relación de Report y su configuración, para que todos los proyectos incluyan los mismo informes:

```

                                <reporting>
                                    <plugins>
                                        <plugin>
<artifactId>maven-surefire-report-plugin</artifactId>
                                        </plugin>
                                </plugin>
                                <artifactId>maven-checkstyle-plugin</artifactId>
                                    </plugin>
                                </plugins>
                                </reporting>
```

Dependencias en Maven. Herencia y Agregación

También podemos incluir los sub-proyectos como módulos. Esto es conocido como agregación y nos permite construir todos los proyectos hijos de una vez desde el directorio del proyecto padre.

```
        <modules>
        <module>myapp-core</module>
        <module>myapp-war</module>
        <module>mayapp-porlet</module>
</modules>
```

Cuando compilamos desde el proyecto padre, todos los proyectos hijos serán compilados.

Archetypes. Creación de proyectos.

A pesar de que Maven define su estructura de directorios, puede ser pesado tener que crear la estructura de directorios a mano cada vez que iniciamos un proyecto. Para simplificar el proceso de iniciar un proyecto, Maven proporciona el plug-in ***archetype***

El ***archetype*** por defecto genera un proyecto Maven produce un proyecto JAR, otros artifact está disponibles para otro tipo de proyectos incluyendo aplicaciones web, para generar plug-in de Maven y otros.

Veamos algunos ejemplos:

Archetypes. Creación de proyectos.

Por ejemplo creación de una tienda On-Line. Siguiendo las buenas prácticas de Maven vamos a dividir el proyecto en varios módulos:

Nuestro modulo de Backend lo llamaremos ShopCoreApi

```
$ mvn archetype:generate -DgroupId=com.ejrbalma.shop -  
DartifactId=ShopCoreApi -DpackageName=com.ejrbalma.shop
```

Nos habrá creado el proyecto en un subdirectorío con el mismo nombre que el artifactID, en este caso ShopCoreApi.

Es habitual que el packageName = groupId

Nos posicionamos en el directorio ShopCoreApi y ejecutamos

```
$ mvn package
```

Archetypes. Creación de proyectos.

Vamos a utilizar otro artifactId para crear un proyecto War.

```
$ mvn archetype:generate -DgroupId=com.ejrbalma.shop -  
DartifactId=ShopWeb -DarchetypeArtifactId=maven-archetype-webapp
```

Es archetype creará un proyecto War.

Archetypes. Creación de proyectos.

Otro archetype muy utilizado es maven-archetype-site que se utiliza para crear un web-site template para un proyecto existente, incluyendo una completa estructura bilingüe con ejemplos XDoc, APT y FAQ's. Este archetype es el único que tenemos que ejecutar sobre un proyecto existente. Aunque no proporciona fuentes de código basados en características, tales como informes de test unitarios, informes de checkstyle, es un buen punto de partida para añadir contenido a un web-site

```
$ mvn archetype:generate -DgroupId=com.ejrbalma.shop -  
DartifactId=ShopCoreApi -DarchetypeArtifactId=maven-archetype-site
```

```
$ cd ShopCoreApi  
$ mvn site
```

Archetypes. Creación de proyectos.

Como no podía ser de otra forma, no solo existen archetype de Maven, también existe de terceros para otro tipo de aplicaciones web y web stack, tales como Struts, Spring, JSF, Hibernate,... Una lista de ellas la podemos encontrar en la web Codehaus.

Por ejemplo el archetype appfuse-basic-spring creará un completo prototipo de aplicación web basando en Hibernate, Spring y JSF.

```
$mvn      archetype:generate      -DgroupId=com.ejrbalma.jpt      -  
DartifactId=shopfront      -DarchetypeArtifactId=appfuse-basic-jsf      -  
DarchetypeGroupId=org.appfuse.archetypes
```

Archetypes. Creación de proyectos.

Esto creará una aplicación web ejecutable, también es un ejemplo de trabajo de un verdadero pom.xml. Intentará conectarse con una base de datos local MySQL (usando el usuario root sin contraseña).

```
$ cd shopfront  
$ mvn jetty:run
```

Compiling Code

Compilar el código en Maven es fácil, simplemente tenemos que ejecutar:

```
$mvn compile
```

Antes de compilar Maven:

- Verifica que todas las dependencias han sido bajadas.
- Buscará cualquiera que todavía no esté.
- Generará cualquier código fuente o recursos del proyecto que necesite generarse
- Instanciará variables y recursos y ficheros de configuración,

Compiling Code

Y todo esto lo hará Maven de forma automática y transparente como una parte normal del ciclo de vida, sin necesidad de una configuración especial.

Para asegurarnos que no existe ningún objeto obsoleto podemos ejecutar:

```
$ mvn clean
```


Compiling Code

A veces se necesita compilar ciertos proyectos con versiones anteriores de Java.

Para ello configuramos el plugin del compilador utilizando source y target, por ejemplo si queremos compilar para la versión Java 4.

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <configuration>
      <source>1.4</source>
      <target>1.4</target>
    </configuration>
  </plugin>
</plugins>
```

Testing Code

Los test unitarios es una parte importante de las modernas metodologías de desarrollo y ella juegan un rol importante en el ciclo de vida de un proyecto maven. Por defecto Maven no permitirá el package o el despliegue de la aplicación a menos que todos los test sean pasado satisfactoriamente. Maven reconoce Junit 3.x y Junit 4 y TestNG unit test. Incluye Maven una estructura de directorio propio para ello src/test.

Para ejecutar los test:

```
$ mvn test
```

Si es necesario compilará primero antes de ejecutar los test.

Testing Code

El detalle del resultado de los test es generado en un fichero xml que se ubica en target/surefire-reports, se puede generar un fichero HTML con el resultado de los test, para ello:

```
$mvn surefire-report:report
```

El HTML será generado en un fichero llamado target/site/surefire-report.html

Otro aspecto importante de los test unitario es Test Coverage, el cual asegura que porcentaje de código está cubierto por los test.

El report de los test está disponible en target/site/cobertureindex.Html

Testing Code

Los test unitarios es una parte importante de las modernas metodologías de desarrollo y ella juegan un rol importante en el ciclo de vida de un proyecto maven. Por defecto Maven no permitirá el package o el despliegue de la aplicación a menos que todos los test sean pasado satisfactoriamente. Maven reconoce Junit 3.x y Junit 4 y TestNG unit test. Incluye Maven una estructura de directorio propio para ello src/test.

Para ejecutar los test:

```
$ mvn test
```

Si es necesario compilará primero antes de ejecutar los test.

Packaging y Deploying una Aplicación

Uno de los principios fundamentales de Maven es que proyecto genera uno y solo un artifact principal. El tipo de artifact generado por un proyecto Maven es definido en <packaging> sección en el archivo POM. Los principales tipo de packaging son:

- jar
- war
- ear

Y un típico ejemplo sería:

```
<groupId>org.sonatype.mavenbook.custom</groupId>  
  <artifactId>simple-weather</artifactId>  
    <packaging>war</packaging>  
      <version>1.0</version>  
        <name>simple-weather</name>  
<url>http://maven.apache.org</url>
```

Packaging y Deploying una Aplicación

El tipo de packaging determinará exactamente como el proyecto es agrupado:

- Las clases compiladas se colocan en la raíz de archivo JAR
- Y En el subdirectorio de la web-Info/classes para archivos War

El siguiente paso es instalar y/o desplegar la aplicación. El comando `install` genera y despliega el artifact del proyecto al repositorio local de la máquina.

```
$ mvn install
```

Packaging y Deploying una Aplicación

El comando `deploy` generará u desplegará el artifact a un servidor remoto vía uno de los protocolos soportados (SSH2, SFTP, FTP y SSH) o simplemente a un sistema de ficheros local.

`$mvn deploy`

La aplicación será desplegada un repositorio remoto definido en `<distributionManagement>` sección del archivo POM. Veamos un ejemplo:

```
        <distributionManagement>
            <repository>
                <id>company.repository</id>
                <name>Enterprise          Maven          Repository</name>
                <url>scp://repo.ejrbalma.com/maven</url>
            </repository>
        </distributionManagement>
```

Packaging y Deploying una Aplicación

Podemos utilizar una ubicación local, para ello:

```
<distributionManagement>
  <repository>
    <id>company.repository</id>
    <name>Enterprise          Maven          Repository</name>
    <url>file:///d:/maven/repo</url>
  </repository>
</distributionManagement>
```


Packaging y Deploying una Aplicación

Si es necesario incluir nombre y contraseña cuando necesitamos copiar en un servidor remoto. Necesitamos incluir esta información en el archivo setting.xml

```
<settings>
```

```
...
```

```
  <servers>
```

```
    <server>
```

```
      <id>company.repository</id>
```

```
      <name>ejrb</name>
```

```
      <password>tigger</password>
```

```
    </server>
```

```
  </servers>
```

```
</settings>
```