Tutorial Introducción a Maven 3

Por Erick Camacho <@ecamacho> <ecamacho at javahispano.org>

Contenido

Introducción	2
¿Qué es Maven?	2
Requisitos.	2
Instalación de Maven 3.	2
Nuestra aplicación	4
Creación del proyecto de tipo POM	4
Compilación de un proyecto Maven	8
El ciclo de vida de los proyectos maven	9
Creación de proyecto de tipo Jar	10
1. Definir que se usará java 5 para compilar.	13
2. Agregar una dependencia.	14
3. Importar el proyecto en Eclipse.	16
Creación de proyecto de tipo War	21
Dependencias Transitivas.	25
Ejecución de una aplicación web	27
Mejorando tu aplicación - técnicas avanzadas	31
Centralizando configuración en el pom padre.	31
Uso de rangos para control de versiones	34
Versiones Snapshot	35
Añadir otros repositorios.	37
Conclusión	39

Introducción

Este tutorial sirve como un quickstart para el uso de Maven 2 y Maven 3. El objetivo central es que al finalizarlo seas capaz de crear una aplicación web con Java usando esta herramienta y entiendas los conceptos básicos utilizados por ella.

¿Qué es Maven?

Maven es una herramienta open source para administrar proyectos de software. Por administrar, nos referimos a gestionar el ciclo de vida desde la creación de un proyecto en un lenguaje dado, hasta la generación de un binario que pueda distribuirse con el proyecto.

Maven nació dentro de la fundación Apache para complementar a Ant, la herramienta de compilación más usada en el mundo Java. Si no la conoces, piensa en Make para C. Esto es, Ant permite crear scripts (usando XML) que indican cómo compilar un proyecto Java y generar un binario.

Maven complementa esta tarea brindándonos una estructura consistente de proyectos (todos los proyectos Maven tienen por default los mismos directorios) y herramientas necesarias actualmente para la complejidad de los proyectos de software: gestión avanzada de dependencias, informes sobre testing automáticos y extensibilidad vía plugins.

Por detrás, Maven sigue confiando en Ant para manejar la compilación e incluso puedes usar las tareas de Ant dentro de Maven. Así que no es en sí un sustituto de esta herramienta, sino un complemento.

Requisitos.

Antes de iniciar este tutorial deberás tener instalado:

- JDK de Java versión 5 ó superior.
- Eclipse Galileo o superior

Instalación de Maven 3.

Vamos a comenzar, lo primero que tienes que hacer es instalarte la herramienta. Para ello basta con que cumplas los siguientes pasos:

Descarga del sitio de Maven (http://maven.apache.org/download.html) la última versión, a la hora de escribir este documento, es la 3.0. Maven se distribuye en un archivo (zip o tar.gz) que contiene la distribución binaria. Descomprime este archivo en cualquier ubicación de tu disco duro local. Para este tutorial, lo haré en /Users/erick/javahispano/tutorial_maven.

Maven se ejecuta con el comando mvn (mvn.bat o mvn.sh) que se encuentra dentro de la carpeta bin de la carpeta que descomprimiste. Para poder usarlo, debes de configurar ciertas variables de ambiente

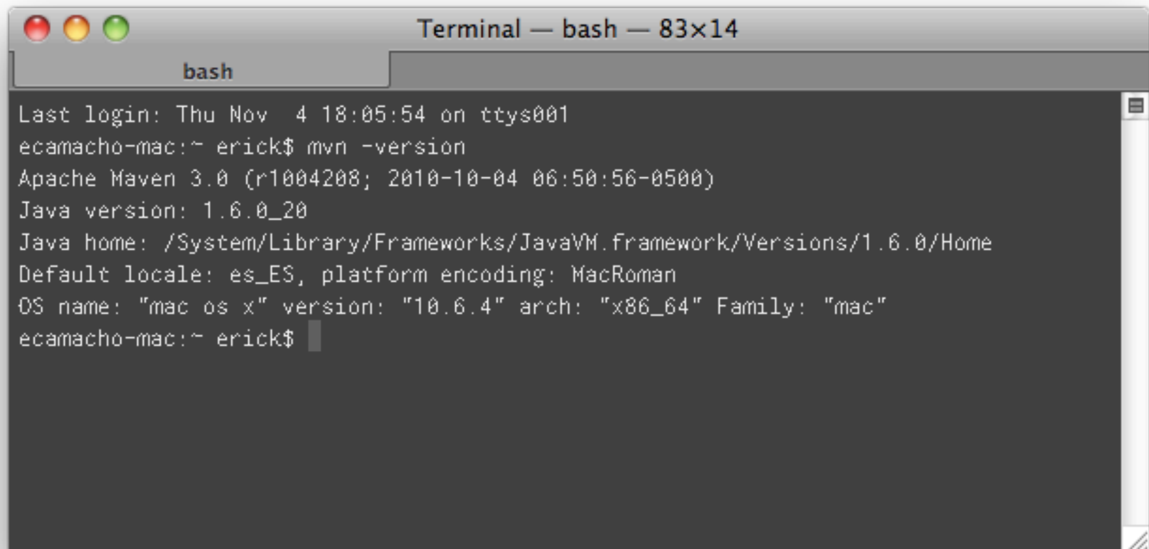
JAVA_HOME. Debe contener el path a tu instalación del JDK. En Windows, normalmente es en C:\Archivos de Programa\Java\jdk1.X. En Ubuntu es en /usr/lib/jvm/java-6-sun y en MacOSX en $(/usr/libexec/java_home).

PATH. Agrega al path la ruta al directorio bin que se encuentra donde descomprimiste Maven. Por ejemplo en mi caso es /Users/erick/javahispano/apache-maven-3.0/bin

Abre una nueva consola de comandos y ejecuta el comando

mvn -version

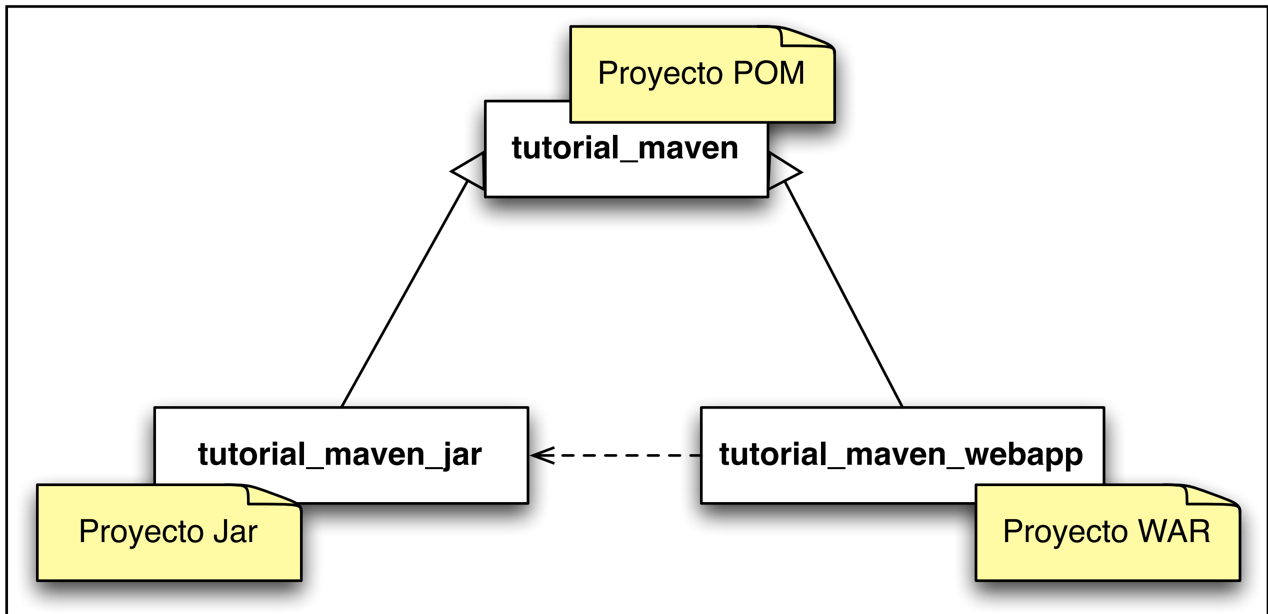
Si configuraste las variables de ambiente correctamente, deberás ver la información de la versión de maven.



Nuestra aplicación

Ya está todo listo para empezar a usar Maven. Maven promueve la **modularización** de los proyectos. Típicamente, en lugar de tener un proyecto monolítico, tendremos un módulo padre y varios módulos hijos.

En nuestro caso, crearemos una aplicación web. La estructura de nuestro proyecto será:



En esta gráfica, estoy usando notación UML para denotar las relaciones . En otras palabras, crearemos un proyecto de tipo POM (ya hablaremos sobre qué son estos proyectos) y dos proyectos hijos: tutorial_maven_jar que es un proyecto de tipo Jar (el binario que genera es un archivo jar) y un proyecto de tipo War (el binario que genera es un .war) que tiene como dependencia al jar.

Creación del proyecto de tipo POM

POM son las siglas de "**Project Object Model**" (Modelo de Objetos de Proyecto), ya hemos dicho que Maven es una herramienta de administración de proyectos. Un POM no es más que la abstracción usada por Maven para definir dichos proyectos, como tal contiene los atributos de estos y las instrucciones para construirlo.

Un Proyecto en Maven se define mediante un archivo POM, que es un archivo llamado **pom.xml** con una etiqueta inicial de <project>. En dicho archivo se definen cosas como las instrucciones para compilar el proyecto, las librerías necesarias, etc. Ya veremos más adelante su estructura y cómo editarlos.

Ahora bien, ¿qué es un proyecto de tipo POM?. En Maven, la ejecución de un archivo POM siempre genera un "**artefacto**". Este artefacto puede ser cualquier cosa: un archivo jar, un swf de flash, un archivo zip o el mismo archivo pom. ¿Para que quieres un archivo pom que te genere el mismo archivo pom? La respuesta es para organización de tu proyecto. Como ya dijimos, Maven trabaja modularizando tus proyectos. De esta forma tendremos varios módulos que conforman un sólo proyecto. Para denotar esta relación en Maven, se crea un proyecto **padre** de tipo POM (esto es que no genera un binario en sí) y los módulos se definen como otros archivos pom que heredan del primero.

Más aún, esta organización sirve para centralizar en el pom padre las variables (como el nombre del proyecto o el número de versión), las dependencias, los repositorios, etc. que son comunes a los módulos, eliminando duplicidad de código.

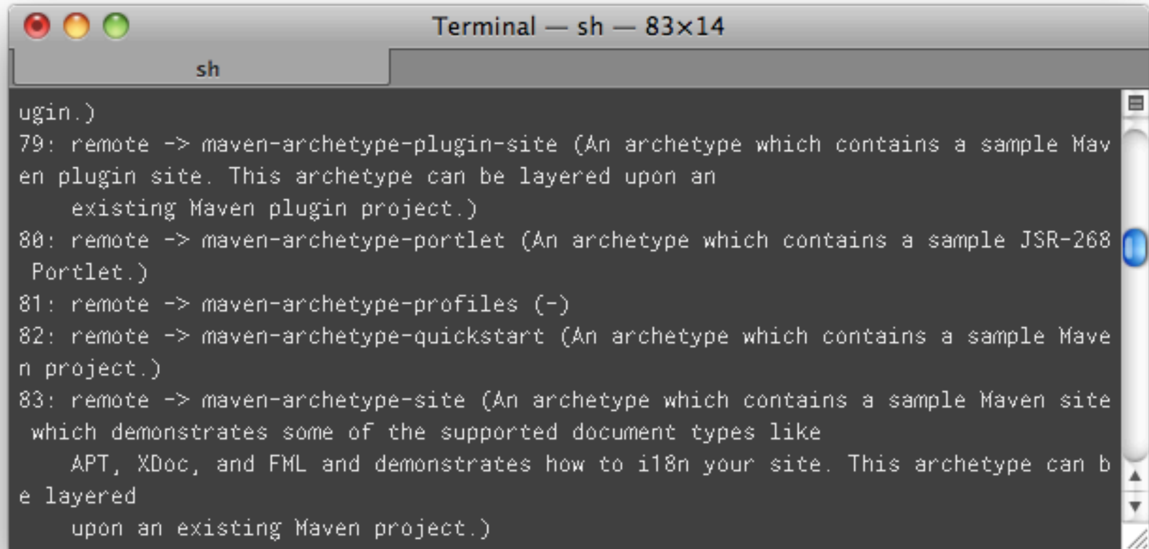
Para crear un proyecto en maven puedes hacerlo a mano siguiendo ciertas **convenciones**: el nombre de la carpeta debe ser igual al nombre del proyecto (en el caso de proyectos módulo, esto es obligatorio, si no son módulos, sino proyectos padre esto no es forzoso pero se recomienda) y a nivel raíz de la carpeta debe estar el archivo pom.xml con la descripción del proyecto. Tu puedes crear esto a mano, sin embargo, es más sencillo usar los **Maven Archetypes**.

Los **arquetipos** son artefactos especiales de Maven que sirven como plantillas para crear proyectos. Maven cuenta con algunos predefinidos y terceros han hecho los suyos para crear proyectos con tecnologías específicas, como proyectos web con Spring o proyectos con Adobe Flex e incluso la gente de AppFuse ha ido más allá y tiene arquetipos para crear proyectos 100% funcionales usando una mezcla de tecnologías y generación de código.

Por ahora, usaremos uno que nos crea un proyecto de tipo pom. Desde línea de comandos y dentro del directorio donde pondrás tu proyecto teclea

mvn archetype:generate

Te aparecerá la lista completa de artefactos en el repositorio central de Maven. Actualmente a mi me salen alrededor de 300 arquetipos (en tu versión puede variar el número). Con este comando puedes darte una idea de los tipos de proyectos que puedes generar con Maven, verás desde cosas muy básicas como j2ee-sample hasta proyectos avanzados con tecnologías no estándares. En este caso tendríamos que elegir la opción maven-archetype-quickstart (la 82 en este ejemplo), pero es un poco complicado buscar el tipo de proyecto que necesitas en una lista tan grande, por ello lo haremos de otra forma.



NOTA: Si es la primera vez que ejecutas maven, este paso tomará mucho tiempo ya que la herramienta empezará a descargar sus dependencias. Afortunadamente esto solo sucede la primera vez.

Si no quieres que te aparezca la lista, puedes definir el arquetipo desde línea de comandos tecleando:

mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-quickstart

El arquetipo te pedirá varios atributos sobre tu proyecto, introduce estos valores (los marcados en rojo):

Define value for groupId: : org.javahispano

Define value for artifactId: : tutorial_maven

Define value for version: 1.0-SNAPSHOT: : 1.0

Define value for package: org.javahispano: org.javahispano

Una breve explicación de ellos:

- **groupId**: Piensa en él como en el paquete de proyecto. Típicamente aquí se pone el nombre de tu empresa u organización, ya que conceptualmente todos los proyectos con ese groupId pertenecen a una sola empresa.
- **artifactId**: Es el nombre de tu proyecto.
- **version**: Número de versión de tu proyecto.
- **package**: Paquete base donde irá tu código fuente

Una vez terminado, verás que el arquetipo creó una carpeta con el nombre del artifactId que dimos (tutorial_maven) y dentro dos cosas: un archivo pom.xml y un directorio src.

Este arquetipo crea un proyecto de tipo jar, necesitamos ahora convertirlo a uno de tipo POM. Borra el directorio llamado **src**, ya que no necesitamos código fuente. Abre con un editor de texto el archivo pom.xml para editarlo. Cambia el tipo de packaging de **jar** a **pom**. Con esto ya cambiamos el tipo de proyecto. Esta es la estructura mínima de un proyecto Maven. Revisa el archivo pom: como ves, al principio están los datos del proyecto y además tenemos una sección de dependencias, estos son las librerías que necesitamos para compilar el proyecto. Guarda los cambios y vuelve a la línea de comandos.

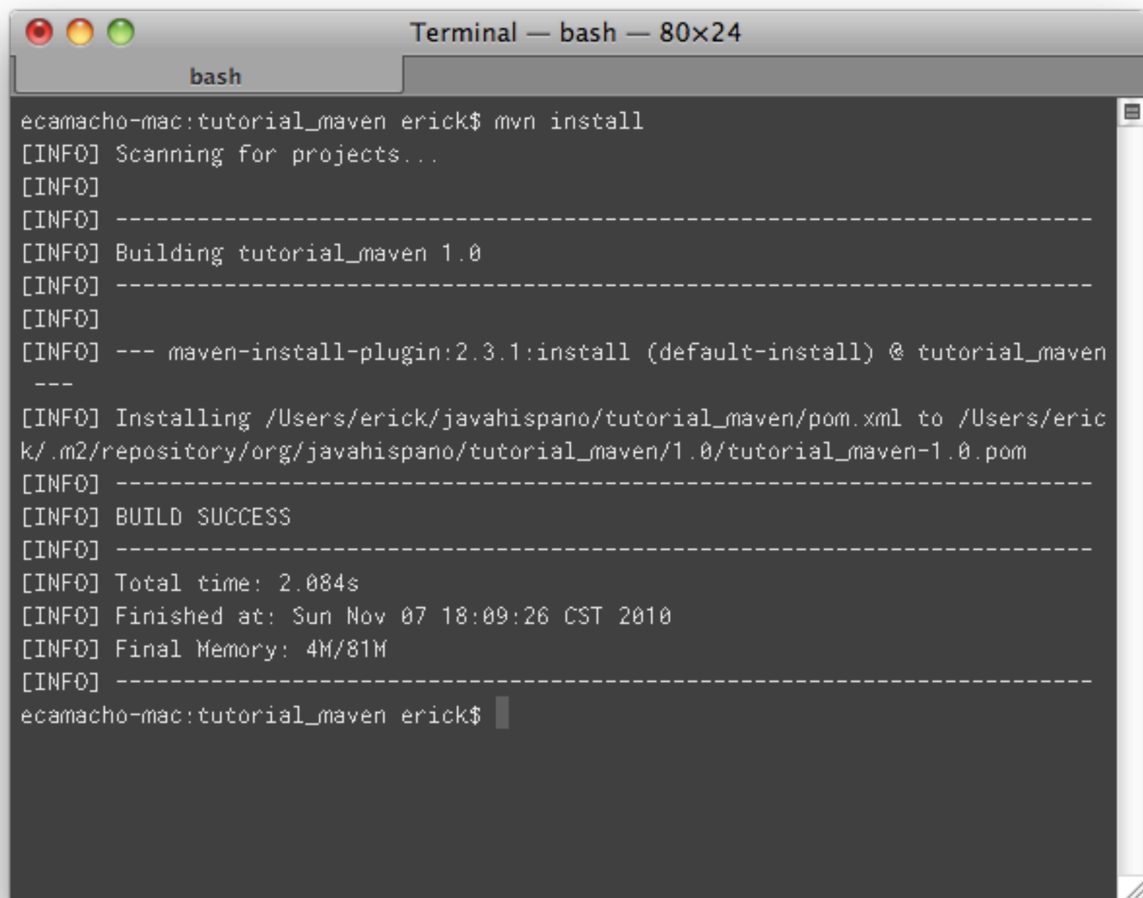
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.javahispano</groupId>
  <artifactId>tutorial_maven</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>tutorial_maven</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Compilación de un proyecto Maven

Ya tenemos un proyecto que genera un archivo pom. Para compilarlo teclea

mvn install

Con el comando install, crea el artefacto de tu proyecto y lo instala en tu repositorio central. La salida del comando será algo como lo siguiente:



¿Qué es el repositorio local? Maven descarga sus dependencias y las dependencias de tus proyectos de un repositorio central. Por default, usa el **repositorio central** de Maven (http://repo1.maven.org/maven2) aunque puedes definir que use otros incluido alguno que tu hayas creado dentro de tu intranet. Estas dependencias las almacena en un repositorio local, que no es otra cosa que una carpeta dentro de tu computadora con el fin de no tener que volver a descargarlas otra vez.

Además de eso, Maven copia ahí tus artefactos, los proyectos que tu generes. Por default, el repositorio local está en tu home de usuario dentro de la carpeta oculta .m2/repository. Recuerda que es oculta, así que debes habilitar el mostrar los archivos ocultos en tu sistema operativo para verla.

Dentro de este repositorio, Maven coloca los artefactos en carpetas de acuerdo a su groupId, artifactId y version. Por ejemplo, si revisas tu repositorio local debes de poder ver que ahí copió el pom de nuestro proyecto en la ruta:

<home>/.m2/repository/org/javahispano/tutorial_maven/1.0/tutorial_maven-1.0.pom

El ciclo de vida de los proyectos maven

En esta ocasión ejecutamos mvn install. Realmente estamos ejecutando una fase del ciclo de vida default de un proyecto Maven. Existen 3 ciclos de vida en Maven 2:

- **clean**. Elimina las clases compiladas y los archivos binarios generados del proyecto
- **default**. Genera los archivos binarios (artefactos) de nuestro proyecto
- **site**. Genera archivos html que describen nuestro proyecto

Para ejecutarlos, excepto el default, se debe de poner *mvn ciclo* (ej: *mvn clean* o *mvn site*). Cada ciclo de vida está constituido por varias fases. Las fases del ciclo de vida default son 24. Algo que a todas luces resulta complicado de comprender, puedes ver la lista completa en la guía de referencia de Maven (http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html). No te preocupes, no necesitas aprenderte todas. Entre las más relevantes (y en el orden que se ejecutan) están:

- **validate**. Valida el proyecto.
- **initialize**. Configura propiedades y crea directorios.
- **compile**. Compila el código fuente del proyecto.
- **test**. Ejecuta las pruebas.
- **package**. Genera el artefacto del proyecto.
- **verify**. Verifica el artefacto generado.
- **install**. Instala el artefacto en el repositorio local.
- **deploy**. Sube el artefacto a un repositorio Maven en la red.

Para ejecutar una fase, basta escribir *mvn fase.* Una fase contiene a todas las fases anteriores. Una fase puede contener varios goals. Para ejecutarlos hay que escribir mvn fase:goal.

Por último, se pueden concatenar varios de estos escribiéndolos uno después del otro: *mvn fase1 fase2:goal1 fase2:goal2*. No te preocupes si no entiendes esto en una primera instancia, en la práctica estarás usando *mvn package* o *mvn install* la mayor parte de las veces para compilar tu proyecto. Adicionalmente, puedes invocar antes a clean para asegurarte que no se

queden binarios de otras compilaciones anteriores, para lo que escribirías: *mvn clean install*.

Creación de proyecto de tipo Jar

Ahora crearemos el primer módulo de nuestra aplicación empaquetado como un jar. Desde línea de comandos y dentro de la carpeta donde creamos el proyecto (tutorial_maven), invocaremos otra vez al arquetipo para crear un nuevo proyecto dentro de éste:

mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-quickstart

Como artifactId, pondremos **tutorial_maven_jar**. Los demás datos, serán idénticos al proyecto pom:

Define value for groupId: : org.javahispano

Define value for artifactId: : tutorial_maven_jar

Define value for version: 1.0-SNAPSHOT: : 1.0

Define value for package: org.javahispano: org.javahispano

Como en el caso anterior, se crea una carpeta con el nombre tutorial_maven_jar. Antes de abrirla, echa un vistazo al archivo pom.xml del proyecto raíz, verás que al final del mismo se han agregado las siguientes líneas:

```
<modules>

   <module>tutorial_maven_jar</module>

</modules>
```

De esta forma, maven permite especificar módulos hijos de un pom.xml padre.

Ahora, si abres el archivo tutorial_maven/tutorial_maven_jar/pom.xml verás:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <modelVersion>4.0.0</modelVersion>
 <parent>
  <artifactId>tutorial_maven</artifactId>
  <groupId>org.javahispano</groupId>
  <version>1.0</version>
 </parent>
 <groupId>org.javahispano</groupId>
 <artifactId>tutorial_maven_jar</artifactId>
 <version>1.0</version>
 <name>tutorial_maven_jar</name>
 <url>http://maven.apache.org</url>
 <dependencies>
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>3.8.1</version>
   <scope>test</scope>
  </dependency>
 </dependencies>
</project>
```

El elemento <parent> indica que este proyecto es **hijo** del pom especificado. Tal como lo habíamos planeado.

Para compilar este proyecto, podemos escribir *mvn install* desde la carpeta tutorial_maven y se compilarán todos los módulos del proyecto, o solamente desde la carpeta tutorial_maven/tutorial_maven_jar si solo queremos construir éste módulo en específico. Por ahora, compilaremos desde el proyecto padre para que te familiarices con el output. Desde la carpeta tutorial_maven, ejecuta el comando

mvn install

Maven te informará sobre los proyectos compilados:

```
Terminal — bash — 80×24
bash
[INFO] Building jar: /Users/erick/javahispano/tutorial_maven/tutorial_maven_jar/
target/tutorial_maven_jar-1.0.jar
[INFO]
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ tutorial_maven
_jar ---
[INFO] Installing /Users/erick/javahispano/tutorial_maven/tutorial_maven_jar/tar
get/tutorial_maven_jar-1.0.jar to /Users/erick/.m2/repository/org/javahispano/tu
torial_maven_jar/1.0/tutorial_maven_jar-1.0.jar
[INFO] Installing /Users/erick/javahispano/tutorial_maven/tutorial_maven_jar/pom
.xml to /Users/erick/.m2/repository/org/javahispano/tutorial_maven_jar/1.0/tutor
ial_maven_jar-1.0.pom
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] tutorial_maven .................................... SUCCESS [0.481s]
[INFO] tutorial_maven_jar ................................ SUCCESS [4.122s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4.771s
[INFO] Finished at: Sun Nov 07 18:13:00 CST 2010
[INFO] Final Memory: 12M/81M
[INFO] ------------------------------------------------------------------------
ecamacho-mac:tutorial_maven erick$
```

Como ya explicamos, Maven compiló el jar y lo subió a tu repositorio local. Si entras a esa carpeta, podrás encontrarlo. Pero además, Maven crea el directorio **target** dentro de la carpeta de tu proyecto y coloca ahí tus binarios. Verifica que en tutorial_maven/tutorial_mave_jar/target, se encuentre el archivo tutorial_maven_jar-1.0.jar.

Revisemos la **estructura** del proyecto. Verás que el arquetipo creo las carpetas **src/main/java** y **src/main/test**. En la primera, se coloca el código fuente y en la segunda los tests. Maven tiene la inteligencia suficiente para incluir en tus binarios sólo las clases en la carpeta java y no la de los tests. Esta estructura es la **convención** usada por Maven y de alguna forma se ha convertido en una convención en proyectos Java. Puedes personalizar el nombre y localización de las carpetas, pero se recomienda usar esta convención dado que otras personas ya acostumbradas a ellas encontrarán más familiar tus proyectos por usarla.

Abre ahora de nueva cuenta el pom.xml de este proyecto. Como verás, solo se tiene una dependencia, la de JUnit que es usada en el test unitario de ejemplo que se creó:

```
<dependency>

    <groupId>junit</groupId>

    <artifactId>junit</artifactId>

    <version>3.8.1</version>

    <scope>test</scope>

</dependency>
```

Algo interesante de esta dependencia es que tiene un <**scope**> definido como de test. Esto indica a Maven que esta librería solo se usará durante la fase de testing y no será incluida en los binarios. Normalmente tus dependencias que definas no llevan este scope porque queremos que sean incluidas en el binario.

NOTA. Existen 6 scopes para las dependencias en Maven:

- **compile.** Por defecto, si no se especifica se usará éste scope. Estas dependencias se usan en el classpath del proyecto y serán incluidas en el artefacto final.
- **provided.** Estas dependencias se usan durante la fase compile y test. Pero no se incluyen el artefacto final. Es muy usado por ejemplo para incluir los jar de J2EE (como servlet-api.jar) ya que son necesarios para compilar pero ya están en tu servidor Java, por lo que no es necesario volverlas a incluir dentro de tu archivo war.
- **runtime.** Indica que la dependencia será necesaria durante la ejecución de tu aplicación pero no al compilar.
- **test.** Indica que la dependencia sólo es necesaria para compilar y ejecutar los tests del proyecto. Estas dependencias no serán incluidas en el artefacto final.
- **system**. Igual a provided pero aquí debes especifciar el path de tu disco duro al jar que contiene esta dependencia. Así evitas que Maven la busque en los repositorios.
- **import.** Solo funciona en Maven 2.0.9 y superior. Permite importar otros archivos pom para simular herencia múltiple ya que Maven solo permite heredar de un solo archivo POM.

Ahora vamos a hacer tres tareas para demostrar como trabajar más a detalle con proyectos Maven:

1. Definir que se usará java 5 para compilar.

2. Agregar una dependencia.

3. Importar el proyecto en Eclipse.

1. Definir que se usará java 5 para compilar.

Por default, Maven compila usando como target java 1.4. Así que una de las tareas que hay que hacer casi siempre, es modificar esto para indicarle que usaremos una versión superior de java. Internamente Maven está usando el plugin de compilación Java, como se usan los valores por defecto no es necesario declararlo en el archivo pom. Sin embargo como cambiaremos esa configuración por convención, tendremos que declararlo. Agrega al archivo lo siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
	<parent>
		<artifactId>tutorial_maven</artifactId>
		<groupId>org.javahispano</groupId>
		<version>1.0</version>
	</parent>
	<groupId>org.javahispano</groupId>
	<artifactId>tutorial_maven_jar</artifactId>
	<version>1.0</version>
	<name>tutorial_maven_jar</name>
	<url>http://maven.apache.org</url>
	 <properties>
	    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
	</properties>
	<build>
		<plugins>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-compiler-plugin</artifactId>
				<configuration>
					<source>1.5</source>
					<target>1.5</target>
				</configuration>
			</plugin>
		</plugins>
	</build>
	<dependencies>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>3.8.1</version>
			<scope>test</scope>
		</dependency>
```

Compila tu proyecto con *mvn install* y verifica que todo funcione correctamente.

2. Agregar una dependencia.

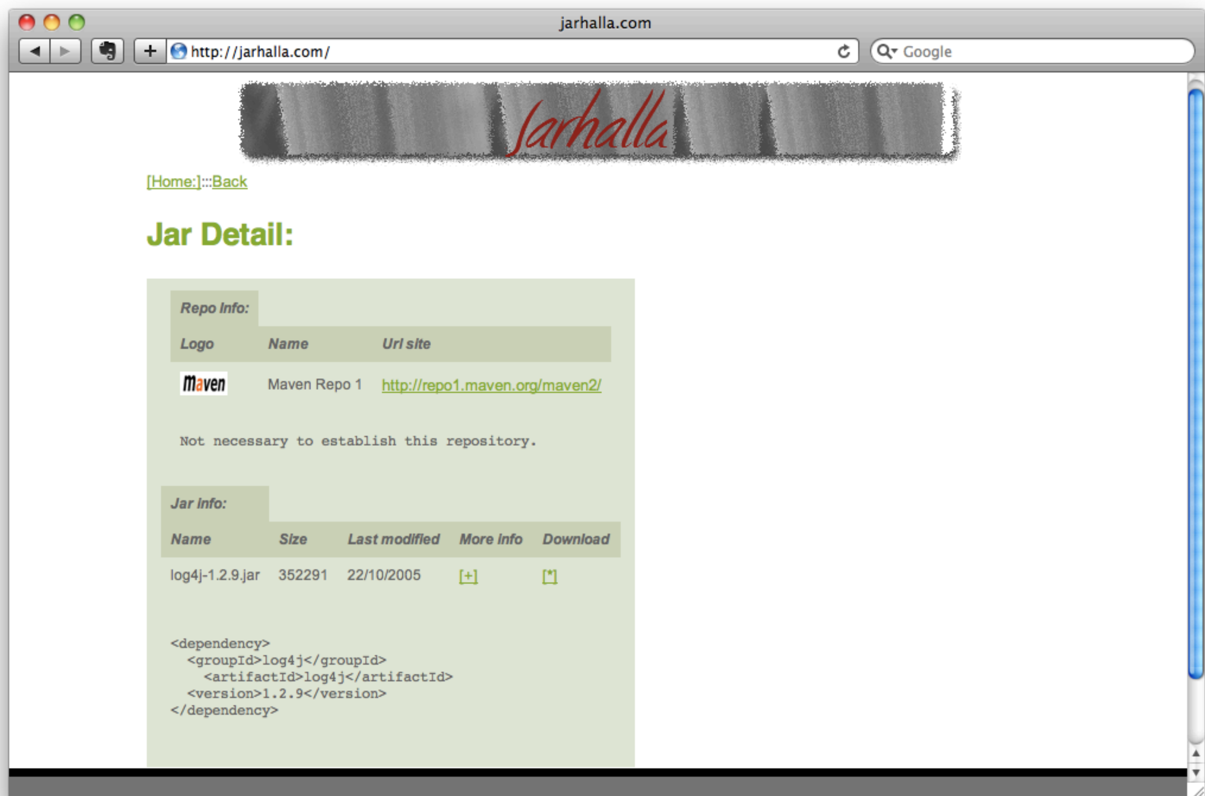
Ahora agregaremos una dependencia, para mostrar el proceso necesario. Pensemos que

queremos usar la librería log4j para el logging de nuestra aplicación, para ello necesitamos el jar de log4j. En ;aven basta con agregar un elemento <**dependency**> con los datos de log4j y eso es todo. El problema aquí es: ¿Cómo saber el groupId y el artifactId de una librería?

Normalmente, en el sitio de las herramientas viene esa información, pero el buscarla puede ser complicado. Otra opción es navegar por el repositorio central de maven, tarea bastante tardada. La opción más simple es utilizar un buscador de repositorios maven. Algunos de los existentes:

- http://www.jarhalla.com
- http://www.mvnrepository.com
- http://www.jarfinder.com
- http://www.jarvana.com

Por ahora usaré jarhalla.com. Realiza una búsqueda por Jar y con el criterio "log4j%". Te saldrá una lista de jars de log4j y da click en el correspondiente a la versión 1.2.9 (la última estable para ese proyecto). Jarhalla te dará el snippet de código que necesitas agregar a tu pom.xml:



Cópialo y pégalo en la sección de dependencies:

```
<dependencies>
	<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>3.8.1</version>
		<scope>test</scope>
	</dependency>
	<dependency>
		<groupId>log4j</groupId>
		<artifactId>log4j</artifactId>
		<version>1.2.9</version>
	</dependency>
</dependencies>
```

Compila el proyecto y revisa como se descarga el archivo log4j-1.2.9.jar. Recuerda que las dependencias se guardan en tu repositorio local, por lo que estará en

<home>/.m2/repository/log4j/log4j/1.2.9/log4j-1.2.9.jar

De esta forma, la siguiente vez que uses esta dependencia en en este o en otro proyecto, ya no se descargará el jar, sino que se usará el que se guardo en tu equipo.

3. Importar el proyecto en Eclipse.

Hemos creado un proyecto Java con Maven, sin embargo, normalmente usarás un IDE para editar el código Java. Si usas NetBeans o IntelliJ Idea, estos editores tienen ya plugins para Maven, así que puedes usarlos para importar proyectos a partir de un pom.xml.

Para el caso de Eclipse también existen plugins, el más completo es m2eclipse (http://m2eclipse.sonatype.org/) de la empresa Sonatype. Sin embargo, Maven tiene un plugin para generar proyectos Eclipse. En lo personal prefiero este plugin porque de esta forma, no necesito obligar a otros desarrolladores a instalar plugins en su Eclipse para poder trabajar con los proyectos, basta con el mismo Maven. Para usarlo, desde linea de comandos y en la carpeta tutorial_maven/tutorial_maven_jar, teclea

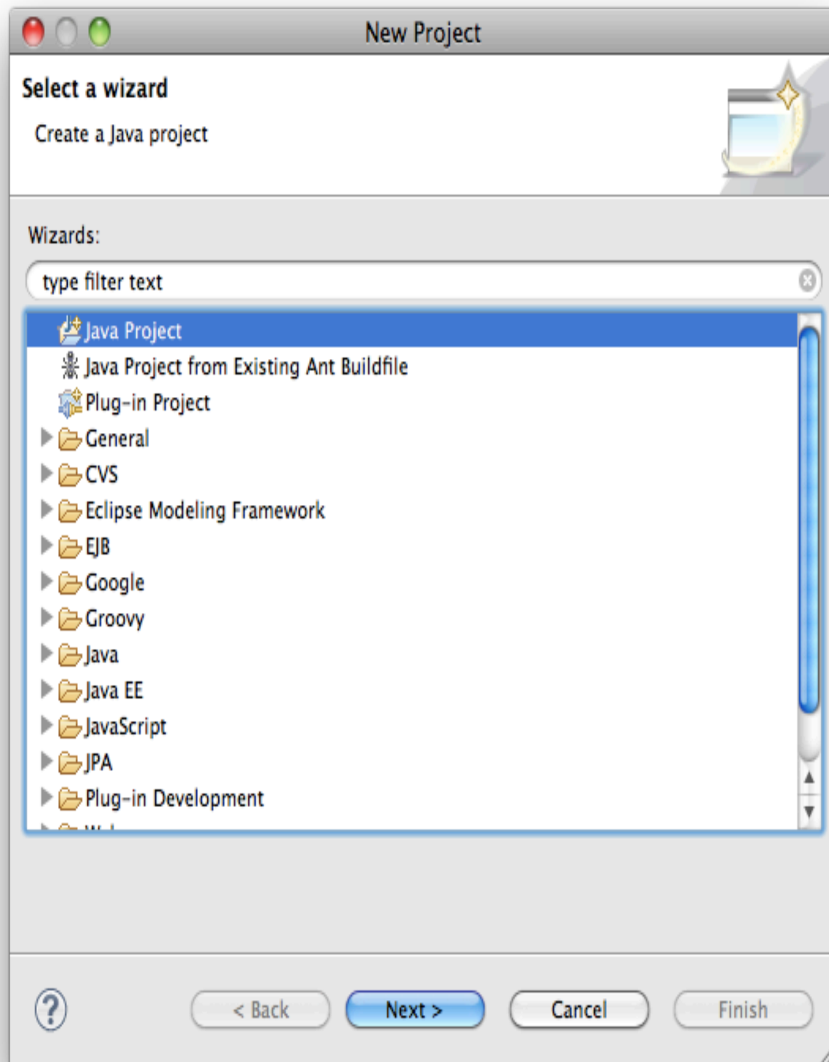
mvn eclipse:eclipse

Verás la siguiente salida

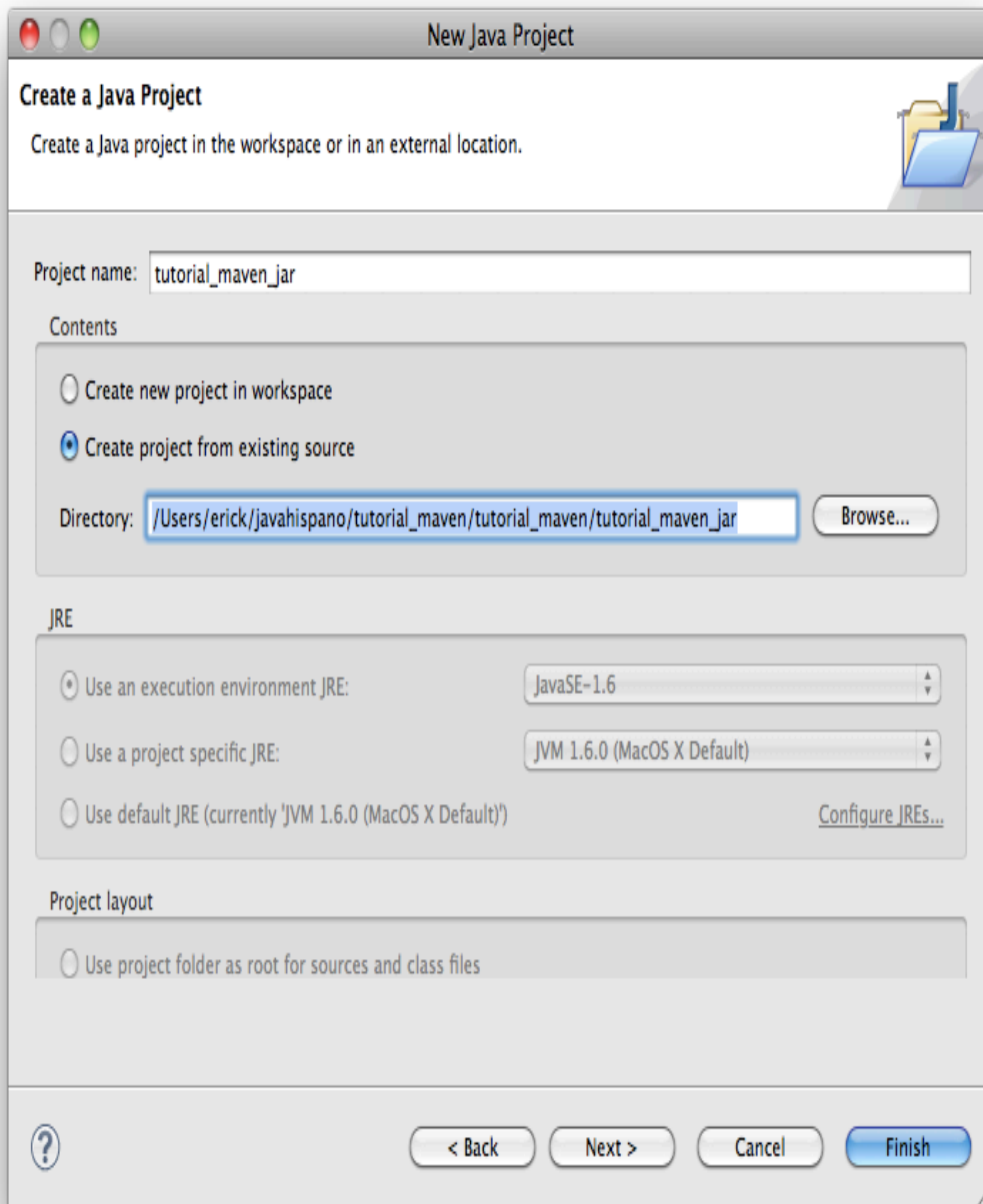
```
Terminal — bash — 80×24
bash
old settings to be removed.
[INFO] Wrote Eclipse project for "tutorial_maven_jar" to /Users/erick/javahispan
o/tutorial_maven/tutorial_maven_jar.
[INFO]
       Sources for some artifacts are not available.
       Please run the same goal with the -DdownloadSources=true parameter in ord
er to check remote repositories for sources.
       List of artifacts without a source archive:
         o junit:junit:3.8.1

       Javadoc for some artifacts is not available.
       Please run the same goal with the -DdownloadJavadocs=true parameter in or
der to check remote repositories for javadoc.
       List of artifacts without a javadoc archive:
         o junit:junit:3.8.1

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.690s
[INFO] Finished at: Sun Nov 07 18:25:12 CST 2010
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
ecamacho-mac:tutorial_maven_jar erick$
```

No te preocupes por los warnings, simplemente indican que no se encontró el código fuente ni los javadoc de junit en el repositorio maven. Si revisas los archivos ocultos de la carpeta, verás que maven generó los descriptores de Eclipse para indicar que este es un proyecto para dicho IDE (archivos .classpath y .project). Ahora abre tu Eclipse y elige cualquier carpeta vacía como workspace (exista o no). Ve al menú

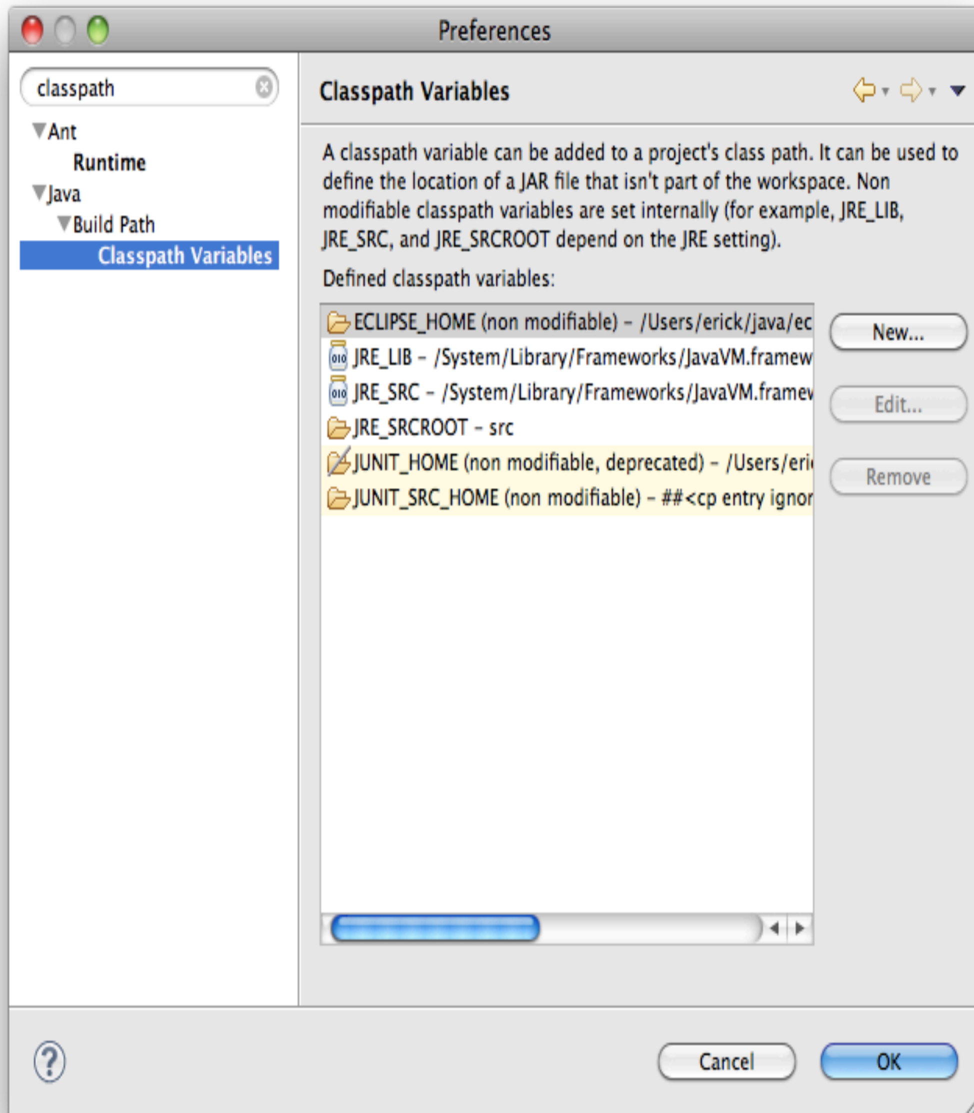
File ->New Project -> Project. En el diálogo elige Java Project:



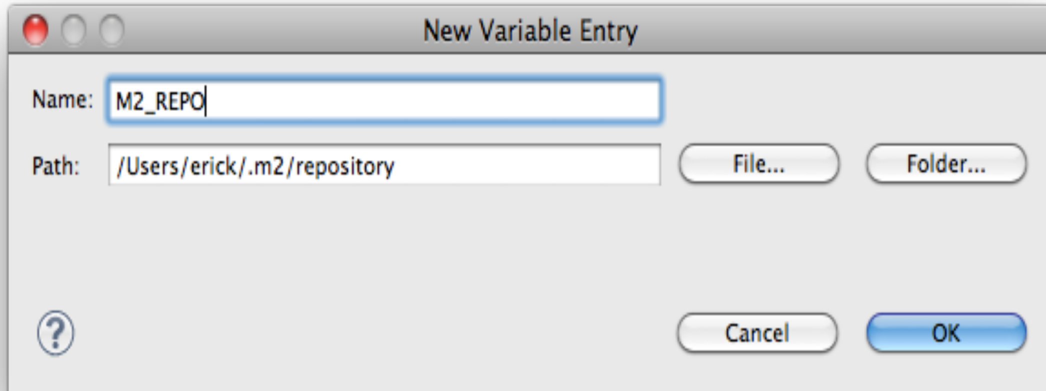
En la siguiente ventana, como Project name escribe "tutorial_maven_jar" y asegúrate de elegir Create Project from existing source. Elige la carpeta tutorial_maven_jar:



Dale en Finish para que se importe el proyecto. En la ventana de problems, verás que maven no pudo encontrar las dependencias del proyecto. El plugin de Maven indica el path a las dependencias apuntando al repositorio local. Para ello utiliza una variable de classpath que debes añadir a tu workspace de Eclipse. Ve a Preferencias (en Mac está en File->Preferences, en Windows y Linux en Help->Preferences). En la caja de búsqueda, escribe "classpath" para encontrar la opción Classpath Variables:



Da clic en New y agrega una llamada M2_REPO y como path pon la ruta a tu repositorio maven local:



Cuando des click en Ok, Eclipse volverá a compilar el proyecto y ya podrá encontrar las dependencias. La variable de classpath que creamos se guarda en el workspace, así que si importas más proyectos Maven dentro del mismo workspace, ya no tendrás que volverla a dar de alta. Si cambias de workspace tendrás que volver a hacerlo. Ahora ya está listo tu proyecto para poder trabajar con Eclipse sobre él. Recuerda que si agregas más dependencias, tendrás que volver a ejecutar mvn eclipse:eclipse para que se actualice el proyecto en Eclipse. Recuerda también no agregar dependencias a través de Eclipse, ya que a la hora de compilar con Maven, éstas no estarán disponibles. Eclipse es solo el editor de código, Maven es quien gestiona todo el proyecto.

Creación de proyecto de tipo War

Ya tenemos el proyecto padre, un módulo de tipo jar y ahora crearemos una aplicación web. Para ello utilizaremos un arquetipo especial para este tipo de aplicaciones. Desde la carpeta con el pom padre (tutorial_maven), ejecuta:

mvn archetype:create -DgroupId=org.javahispano -DartifactId=tutorial_maven_webapp -Dversion=1.0 -DarchetypeArtifactId=maven-archetype-webapp

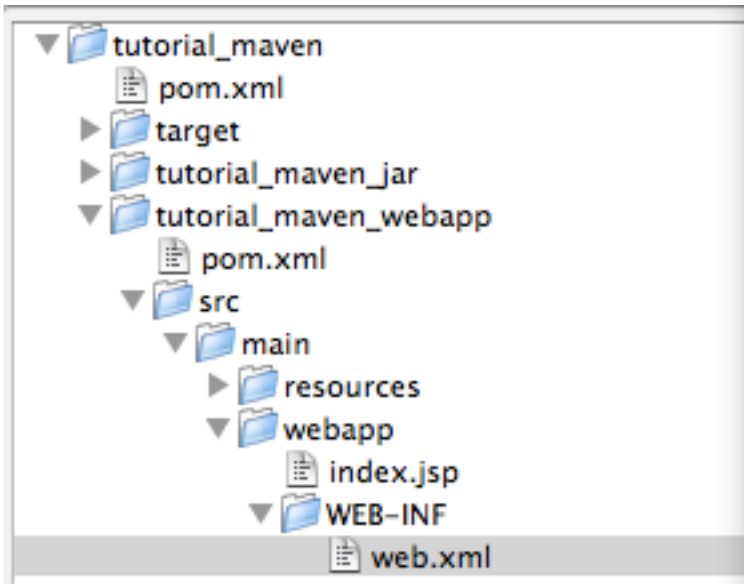
En esta ocasión pusimos el groupId y el artifactId desde un inicio, para que nos lo pregunte al ejecutarse

```
Terminal — bash — 80×24
bash
---
[INFO] Parameter: groupId, Value: org.javahispano
[INFO] Parameter: packageName, Value: org.javahispano
[INFO] Parameter: package, Value: org.javahispano
[INFO] Parameter: artifactId, Value: tutorial_maven_webapp
[INFO] Parameter: basedir, Value: /Users/erick/javahispano/tutorial_maven
[INFO] Parameter: version, Value: 1.0
[INFO] ********************* End of debug info from resources from generated POM
 ***********************
[INFO] project created from Old (1.x) Archetype in dir: /Users/erick/javahispano
/tutorial_maven/tutorial_maven_webapp
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] tutorial_maven .................................... SUCCESS [1.054s]
[INFO] tutorial_maven_jar ................................ SKIPPED
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.617s
[INFO] Finished at: Sun Nov 07 18:27:32 CST 2010
[INFO] Final Memory: 8M/81M
[INFO] ------------------------------------------------------------------------
ecamacho-mac:tutorial_maven erick$
```

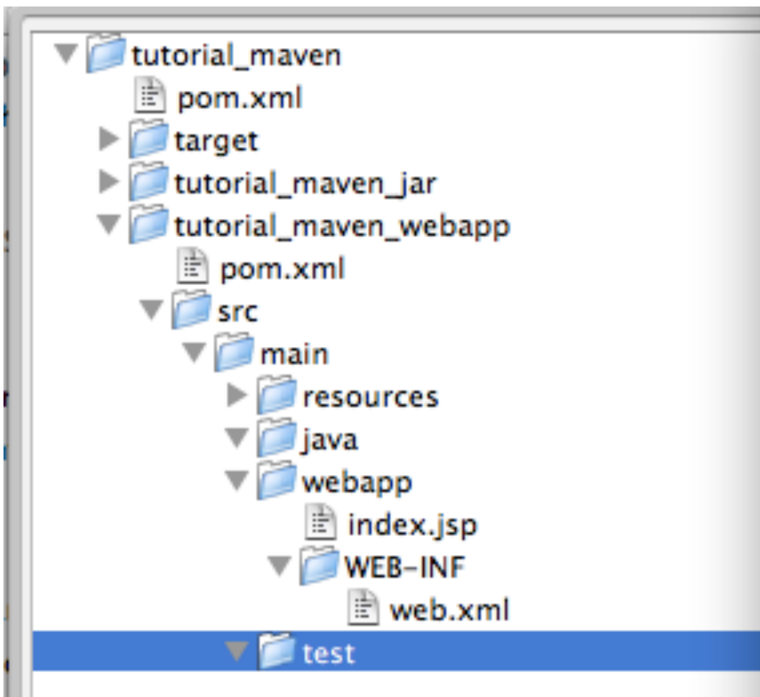
Si verificas el pom.xml padre, verás que ahora ya tenemos un nuevo modulo

```
<modules>
   <module>tutorial_maven_jar</module>
   <module>tutorial_maven_webapp</module>
</modules>
```

Si revisas el proyecto que se creó en tutorial_maven_webapp verás lo siguiente:



Como ves, Maven creó una carpeta **src/main/webapp** con la estructura de un **war**, incluyendo un index.jsp y un web.xml. Este arquetipo no crea la carpeta src/main/java ni src/main/test. Así que conviene que las crees para poder poner tu código java y tus tests:

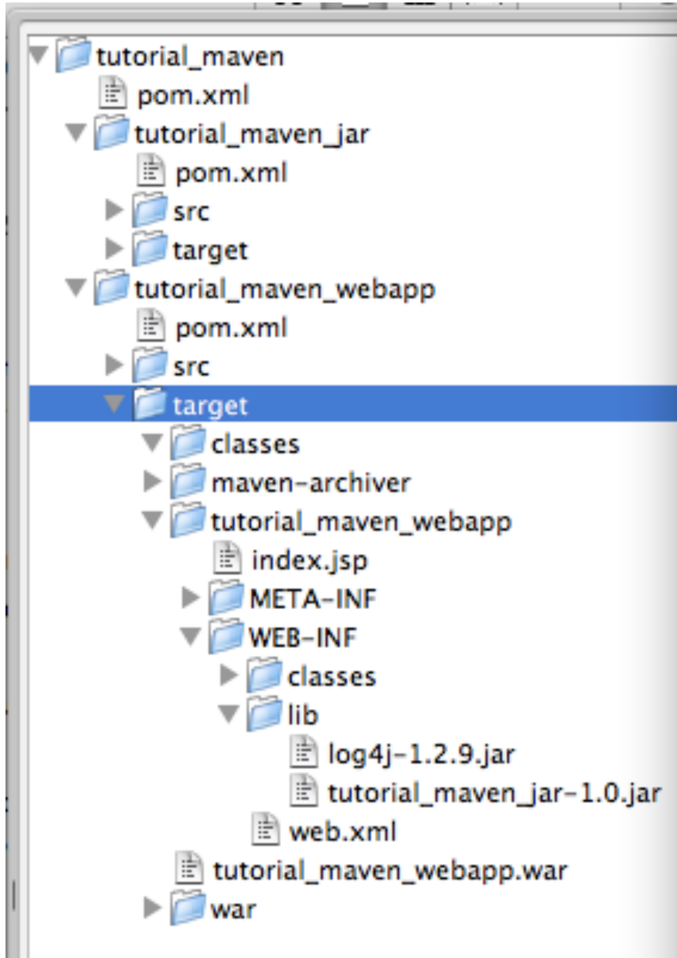


En esta ocasión, agregaremos una dependencia al proyecto tutorial_maven_jar:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>tutorial_maven</artifactId>
    <groupId>org.javahispano</groupId>
    <version>1.0</version>
  </parent>
  <groupId>org.javahispano</groupId>
  <artifactId>tutorial_maven_webapp</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>
  <name>tutorial_maven_webapp Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <build>
    <finalName>tutorial_maven_webapp</finalName>
  </build>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.javahispano</groupId>
      <artifactId>tutorial_maven_jar</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
</project>
```

El elemento buildFinalName, indica el nombre que tendrá el binario del proyecto. Por default Maven usa como nombre el artifactId + version + package extension. Es por ello que en el caso del proyecto de tipo jar, el binario se llama tutorial_maven_jar-1.0.jar. Para los wars, no es conveniente concatenarle el número de versión y por eso se sobreescribe el comportamiento por default para que el binario quede como **tutorial_maven_webapp.war**.

En este punto puedes compilar tu aplicación con *mvn install* e importarla a eclipse com *mvn eclipse:eclipse* si es que así lo deseas. Maven genera el archivo tutorial_maven_webapp/target/tutorial_maven_webapp.war con tu aplicación. Además podrás ver que crea una carpeta llamada target/tutorial_maven_webapp con el contenido que va dentro del war. Ahí podrás verificar que en la carpeta WEB-INF/lib ha puesto tus dependencias: tutorial_maven_jar.jar y log4j-1.2.9.jar:



Dependencias Transitivas.

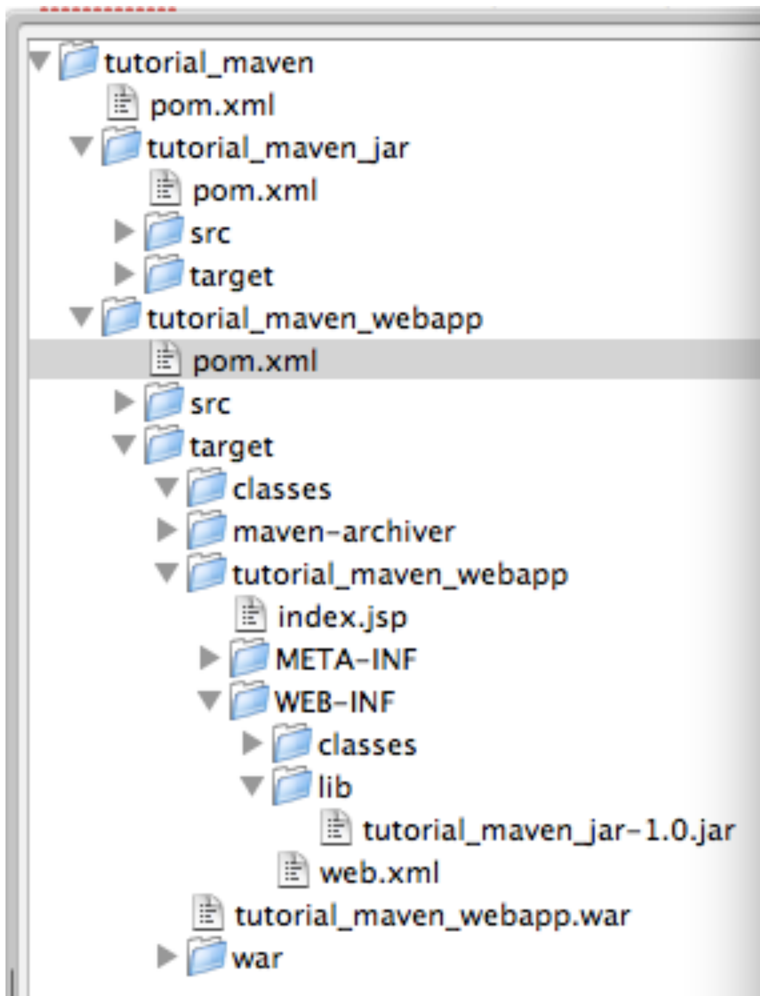
¿Por qué Maven incluyó el archivo log4j.jar en tu war? Porque es una dependencia de una de nuestras dependencias. A este tipo de dependencias se les llama **transitivas**. Estas dependencias son uno de los principales problemas de Maven, ya que es difícil lidiar con ellas.

Es muy común que tu incluyas una librería en tu proyecto y no sepas que esa librería va a descargarse N librerías transitivas. Esto no sería un problema en un mundo ideal donde quienes generan los archivos pom de las librerías ponen solamente las verdaderas dependencias de sus proyectos. Pero en el mundo real, los archivos pom suelen incluir dependencias que realmente no son necesarias, que además de "engordar" tu archivo war, pueden ocasionar problemas de incompatibilidades con librerías que sí son necesarias o incluso problemas legales al incluir librerías cuyas licencias son incompatibles con tu proyecto actual. Existen dos formas de arreglar este tipo de problemas.

La primera forma es editando el archivo pom.xml del proyecto que tiene la dependencia transitiva, en este caso de tutorial_maven_jar. Tenemos que agregar el elemento **<optional>** con un valor de "**true**" en la declaración de la dependencia log4j:

```
<dependency>

        <groupId>log4j</groupId>

        <artifactId>log4j</artifactId>

        <version>1.2.9</version>

        <optional>true</optional>

</dependency>
```

Este elemento indica a Maven que esta dependencia es opcional y no será agregada a los proyectos que usen a tutorial_maven_jar como dependencia. Si ejecutas mvn install a todo el proyecto, deberás ver que log4j es eliminado de tutorial_maven_webapp:



Esta forma funciona muy bien si tienes la oportunidad de editar el pom de los proyectos. Sin embargo, esto no es posible con proyectos que tu no controlas. Para poder quitar estas dependencias transitivas de las que no puedes editar su archivo pom.xml, puedes usar el elemento **<exclusions>**, este elemento se pone en el pom de tus proyectos, en este caso en el del proyecto tutorial_maven_webapp e indica que no incluya una dependencia transitiva en específico.

Para usar esta técnica necesitas modifica el archivo pom.xml del proyecto web para editar la dependencia a tutorial_maven_jar y asegúrate de quitar el elemento <optional> del pom de dicho proyecto:

```
<dependency>
        <groupId>org.javahispano</groupId>
        <artifactId>tutorial_maven_jar</artifactId>
        <version>1.0</version>
        <exclusions>
                <exclusion>
                        <groupId>log4j</groupId>
                        <artifactId>log4j</artifactId>
                </exclusion>
        </exclusions>
</dependency>
```

Si ejecutas mvn install comprobarás que Maven no incluyó a log4j en el war. Ambas opciones son muy tediosas y requieren estudiar a fondo las dependencias de las librerías que importas, por ello Maven ha sido muy criticado. Como regla general, revisa siempre la carpeta WEB-INF/lib de tus archivos war que generas con Maven para verificar cuáles librerías te está agregando, con el fin de que sepas exactamente qué librerías transitivas tienes.

Ejecución de una aplicación web

Maven a través ejecutar el comando install, te genera el archivo tutorial_maven_webapp.war en el directorio target. Este archivo está ya listo para desplegarlo en cualquier contenedor de servlets, como Tomcat, Jetty, Resin, WebLogic, JBoss, etc. Sin embargo esto no es muy recomendable durante el desarrollo. Si el programador pierde tiempo desplegando wars, tirando y levantando un servidor, etc, es tiempo que se le quitará al desarrollo en sí de la aplicación . Una opción es usar un plugin dentro de tu IDE como **WTP** de Eclipse que te

permite desplegar tu aplicación en un **Tomcat** con un solo click. Sin embargo, este tipo de plugins requieren que tú adaptes tus proyectos a su estructura, por lo que pierdes portabilidad de tu aplicación y para desplegarla requerirás que los otros desarrolladores tengan el mismo IDE y el mismo plugin que tu.

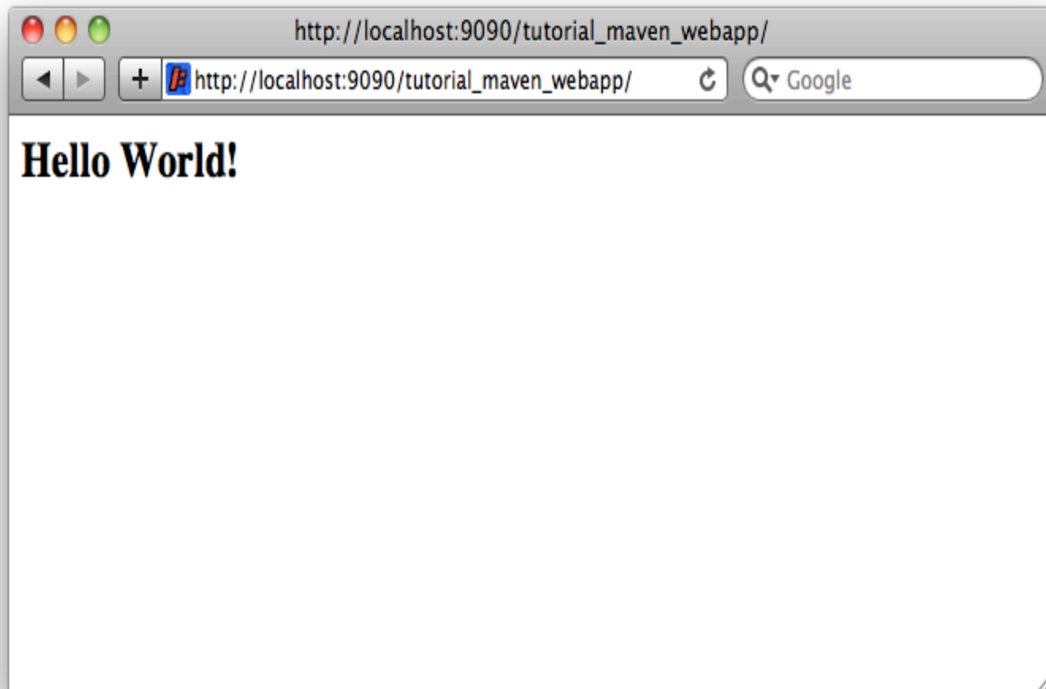
Para este problema, puedes usar el plugin de **Jetty** para Maven (http://docs.codehaus.org/display/JETTY/Maven+Jetty+Plugin). Jetty es un contenedor de servlets muy ligero y flexible. Mediante su plugin, Maven desplegará tu aplicación web en una instancia de Jetty que incluso podrá detectar cuando hagas cambios a tu código para recargarla automáticamente. Para configurar este plugin, se debe de modificar el archivo pom de tu proyecto de tipo war . En este caso, el nuestro quedaría así:

```
 <?xml version="1.0" encoding="UTF-8"?>
<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>
	<parent>
		<artifactId>tutorial_maven</artifactId>
		<groupId>org.javahispano</groupId>
		<version>1.0</version>
	</parent>
	<groupId>org.javahispano</groupId>
	<artifactId>tutorial_maven_webapp</artifactId>
	<version>1.0</version>
	<packaging>war</packaging>
	<name>tutorial_maven_webapp Maven Webapp</name>
	<url>http://maven.apache.org</url>
	<build>
		<finalName>tutorial_maven_webapp</finalName>
		<plugins>
			<plugin>
				<groupId>org.mortbay.jetty</groupId>
				<artifactId>maven-jetty-plugin</artifactId>
				<configuration>
					<connectors>
						<connector
implementation="org.mortbay.jetty.nio.SelectChannelConnector">
							<port>9090</port>
						</connector>
					</connectors>
				</configuration>
			</plugin>
		</plugins>
	</build>
	<dependencies>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>3.8.1</version>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>org.javahispano</groupId>
			<artifactId>tutorial_maven_jar</artifactId>
			<version>1.0</version>
			<exclusions>
				<exclusion>
					<groupId>log4j</groupId>
					<artifactId>log4j</artifactId>
				</exclusion>
			</exclusions>
		</dependency>
	</dependencies>
</project>
```

Aquí configuramos el plugin para levantar nuestra aplicación en el puerto 9090. Para ejecutarlo, desde línea de comandos:

mvn jetty:run

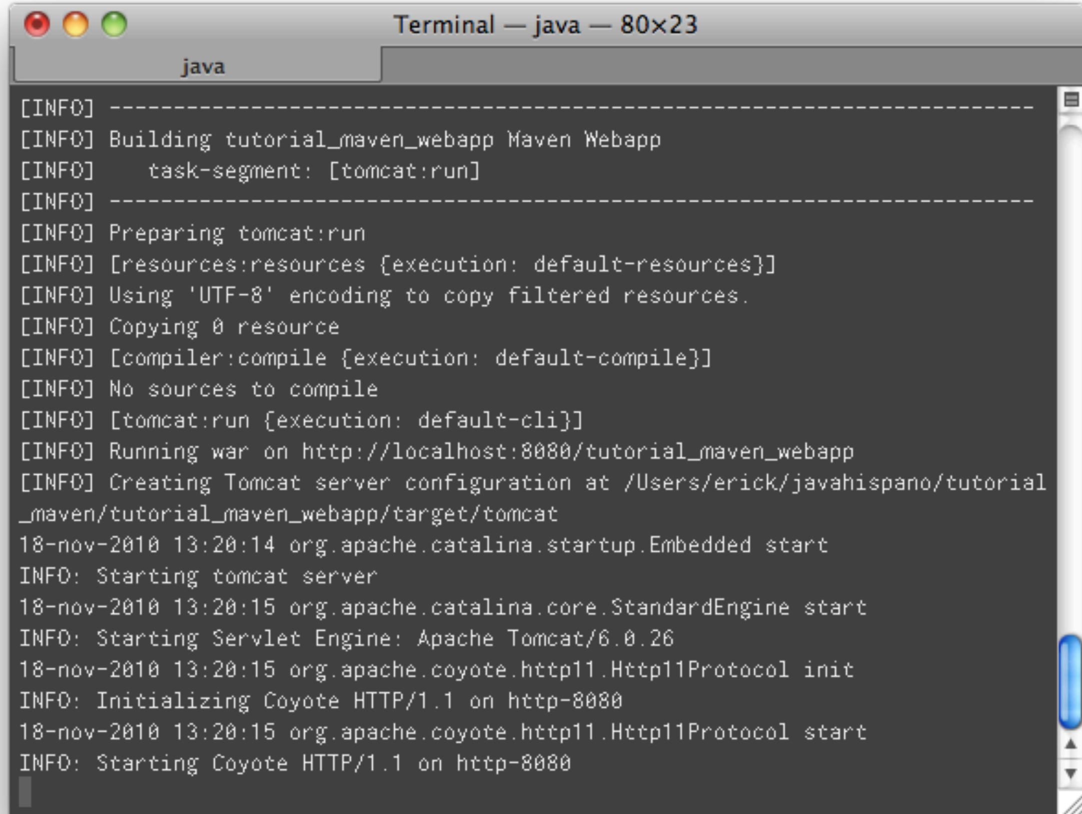
Una vez que les aparezca la leyenda *[INFO] Started Jetty Server*, pueden entrar a http://localhost:9090/tutorial_maven_webapp/



Lo que aparece es el index.jsp que creó Maven por default. De esta forma podemos trabajar en nuestro ambiente local, por defecto el plugin de Jetty despliega todo los archivos estáticos que están en la carpeta src/main/webapp por lo que cualquier cambio que hagas a alguno de estos ficheros, se reflejará instantáneamente con solo recargar tu browser. Por su parte, las clases las carga del directorio target/classes y escanea constantemente dicho directorio. Si modificas alguna clase y la compilas (por ejemplo desde tu Eclipse al guardar un archivo modificado, se compila automáticamente), Jetty volverá a cargar tu aplicación para reflejar el cambio. De esta forma tendrás un entorno muy ágil para programar.

Existe también un plugin de Tomcat. Para este plugin no necesitas configurar nada para usarlo con los valores por default, así que si simplemente escribes en la consola *mvn tomcat:run* Maven descargará el plugin, iniciará Tomcat y desplegará tu aplicación en el puerto 8080:

Puedes encontrar más opciones de configuración en el sitio del plugin (http://mojo.codehaus.org/tomcat-maven-plugin/deployment.html).



Mejorando tu aplicación - técnicas avanzadas

Ya hemos visto lo básico para desarrollar aplicaciones Java con Maven. Uno de los fuertes de Maven es que es una herramienta muy completa (y por momentos compleja) que te permite administrar a un nivel muy fino tu aplicación. En esta sección veremos algunas técnicas para ello.

Centralizando configuración en el pom padre.

Al principio de este tutorial mencioné que el usar un pom padre permite centralizar configuración común de tus módulos. En esta sección mostrare cómo se hace eso. Como primer tarea crearemos una variable para controlar el número de versión de log4j. Abre el archivo pom padre y dentro del elemento properties agrega estas líneas:

```
<?xml version="1.0" encoding="UTF-8"?>
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.javahispano</groupId>
  <artifactId>tutorial_maven</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>tutorial_maven</name>
  <url>http://maven.apache.org</url>
  <properties>
     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
     <log4j.version>1.2.9</log4j.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <modules>
    <module>tutorial_maven_jar</module>
    <module>tutorial_maven_webapp</module>
  </modules>
```

Con el tag **properties** declaramos estas variables, éstas pueden tener el nombre que quieras. En este caso estamos declarando una que contenga el número de versión de log4j. Ahora abre el archivo tutorial_maven_jar/pom.xml y hay que remplazar el valor en duro de la dependencia a log4j por la variable que acabamos de crear:

```
<dependencies>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>3.8.1</version>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>log4j</groupId>
			<artifactId>log4j</artifactId>
			<version>${log4j.version}</version>
		</dependency>
</dependencies>
```

Compila con mvn install para verificar que no haya errores.

Además de las variables personalizadas, Maven tiene **variables de proyecto**. Por ejemplo, para guardar el groupId o la versión de los mismos. Estas variables las podemos usar en los archivos pom hijos. Vamos a cambiar los archivos pom.xml para heredar el groupId y la versión del padre.

Edita el archivo tutorial_maven_jar/pom.xml:

```
<groupId>${parent.groupId}</groupId>

<version>${parent.version}</version>
```

Edita ahora el archivo tutorial_maven_webapp/pom.xml:

```
<groupId>${parent.groupId}</groupId>

<artifactId>tutorial_maven_war</artifactId>

<version>${parent.version}</version>

<name>tutorial_maven_war</name>

<packaging>war</packaging>
```

Adicionalmente, podemos usarlas para la parte de la dependencia al proyecto jar que tenemos en el mismo archivo pom:

```
<dependency>
        <groupId>${parent.groupId}</groupId>
        <artifactId>tutorial_maven_jar</artifactId>
        <version>${parent.version}</version>
        <exclusions>
                <exclusion>
                        <groupId>log4j</groupId>
                        <artifactId>log4j</artifactId>
                </exclusion>
        </exclusions>
</dependency>
```

De esta forma, será más fácil cambiar de versión de tu proyecto. En la guía de referencia de Maven (http://docs.codehaus.org/display/MAVENUSER/MavenPropertiesGuide) puedes ver una lista de las variables de proyecto que Maven maneja y que puedes usar en tus archivos pom.

Uso de rangos para control de versiones

Hasta ahora hemos usado un número de versión específico en nuestras dependencias. Maven también permite especificar rangos. Para ello se usan estos símbolos:

[,] -> Significan inclusive.

(,) -> Significan exclusivo.

Ejemplifiquemos esto con la property log4j.version que creamos en el archivo tutorial_maven/pom.xml

Si queremos especificar que se use como mínimo la versión 1.2.7 (inclusive) de log4j y cómo máximo cualquier versión menor a la 1.2.9 (exclusive), pondríamos:

```
<properties>

    <log4j.version>[1.2.7, 1.2.9)</log4j.version>

</properties>
```

Además podemos dejar un límite abierto, por ejemplo si queremos especificar que como mínimo se use la versión 1.2.7 de log4j y como máximo la que sea:

```
<properties>

    <log4j.version>[1.2.7, )</log4j.version>

</properties>
```

O al revés, si queremos indicar que se use cualquier versión menor a la 1.2.9:

```
<properties>

    <log4j.version>(, 1.2.9)</log4j.version>

</properties>
```

Estos rangos son muy útiles para mejorar nuestros archivos pom.xml. Por ejemplo si otras dependencias de nuestro proyecto tienen como dependencia transitiva una versión en específico de log4j y nosotros además especificamos una distinta, podemos acabar teniendo varias versiones de log4j en nuestra aplicación web, lo que sería un problema. Al usar rangos en vez de una versión determinada, estamos abriendo la posibilidad de reusar las dependencias transitivas, por lo que al final tendríamos una sola versión de log4j en nuestro archivo war.

Versiones Snapshot

Hasta ahora hemos usado una versión 1.0 en nuestro proyecto. Supongamos que hay un proyecto que depende del nuestro, ¿qué sucede si hacemos un cambio en nuestra aplicación y queremos que el otro proyecto actualice su dependencia al nuestro? Pues tendríamos que cambiar el número de versión y los desarrolladores del otro proyecto deberán de especificar esta nueva versión en su pom. Esto es bastante trabajo y si te olvidas de cambiar de versión o los otros de especificar la nueva, nunca tendrán la versión actualizada de tu artefacto.

Para estos casos existe una versión especial llamada Snapshot. Esta versión denota a proyectos en desarrollo, en constante cambio. Maven en estos casos siempre descargará la nueva versión del repositorio. De esta forma tú puedes estar haciendo cambios constantes a tus artefactos y Maven garantiza que aquellos que los usen, tendrán la última versión sin necesidad de estar cambiando de número de versión. Vamos a cambiar la versión de nuestro proyecto a una versión Snapshot.

Edita el archivo pom.xml padre y cambia la versión:

```
<groupId>org.javahispano</groupId>

<artifactId>tutorial_maven</artifactId>

<packaging>pom</packaging>

<version>1.0-SNAPSHOT</version>

<name>tutorial_maven</name>
```

Es necesario también actualizar la referencia al pom padre en los archivos pom de tutorial_maven_jar y tutorial_maven_webapp:

```
<parent>

   <artifactId>tutorial_maven</artifactId>

   <groupId>org.javahispano</groupId>

   <version>1.0-SNAPSHOT</version>

</parent>
```

Recuerda que los SNAPSHOT se usan solo en fase de desarrollo, cuando ya tengas una primera versión Release, cambia a una notación normal como 1.0.

Añadir otros repositorios.

Hasta ahora hemos estado usando el repositorio central para descargar las dependencias. Lamentablemente no todas las dependencias que necesitemos en un proyecto real estarán en el repo central de Maven. Esto es porque pueden ser artefactos creados dentro de nuestra empresa, porque son productos que nunca llegan al repositorio central (típico en proyecto en estado beta que se distribuyen como Snapshots) o versiones muy nuevas que todavía no han pasado los criterios de aceptación para ser incluidos en el repositorio central.

Es por ello que existen otros repositorios en internet, por ejemplo JBoss tiene uno propio donde podrás encontrar las versiones más nuevas de Hibernate y SpringSource tiene uno para las versiones de desarrollo de Spring. Para el caso de artefactos internos, puedes montar tu propio repositorio. Para ello existen varios productos , los más usados son

•**Nexus** de Sonatype (http://nexus.sonatype.org/)

•**Artifactory** de JFrog (http://www.jfrog.org/products.php)

•**Archiva** de Apache (http://archiva.apache.org/)

Vamos a agregar una dependencia que no está en el repo central, una versión mileston del proyecto **Spring**. Estas versiones son liberadas antes de tener una versión final y nunca llegan al repo central. En el repo central solo se tienen versiones finales. Así que supongamos que queremos utilizar esta versión milestone por alguna razón.

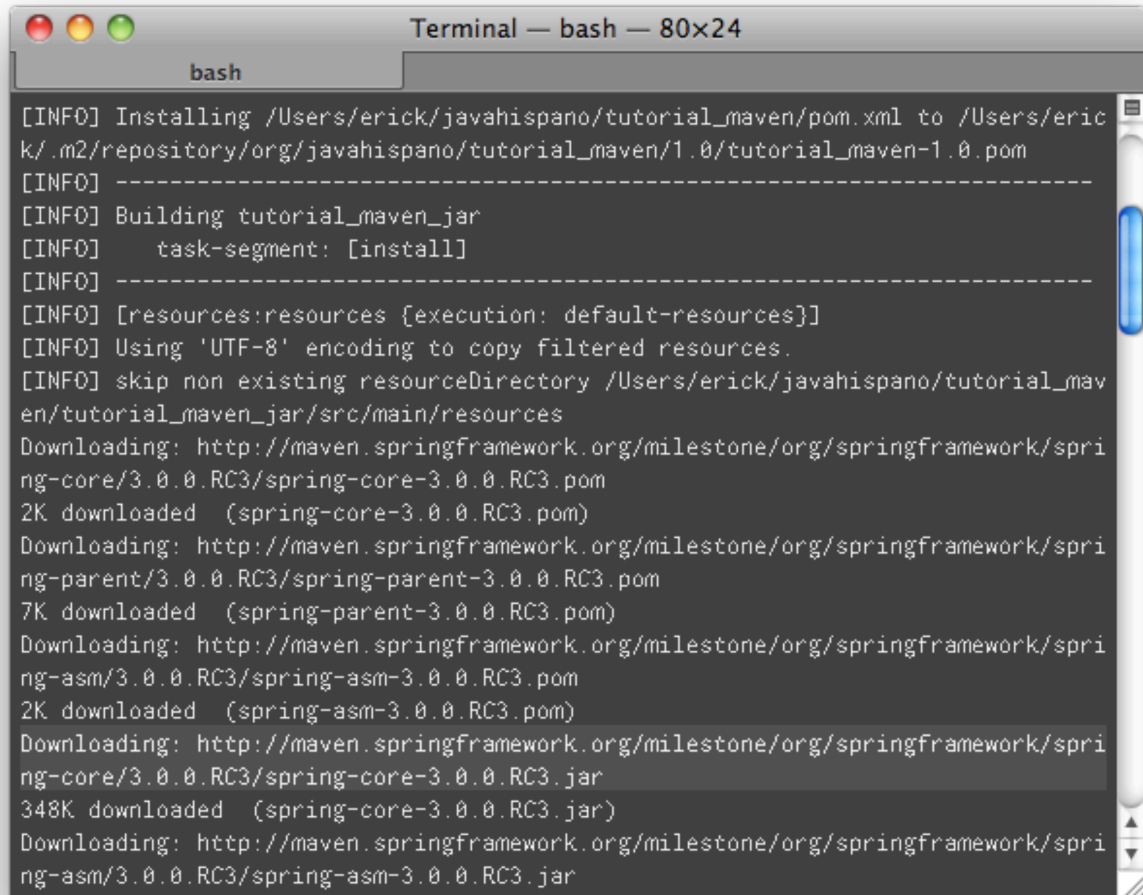
Edita el archivo padre pom.xml y al final agrega el tag repositories:

```
</modules>

 <repositories>

        <repository>

                <id>org.springframework.maven.milestone</id>

                <url>http://maven.springframework.org/milestone</url>

        </repository>

 </repositories>

</project>
```

Ahora edita el archivo tutorial_maven_jar/pom.xml para incluir una dependencia a Google Collections:

```
<dependencies>
   <dependency>
           <groupId>junit</groupId>
           <artifactId>junit</artifactId>
           <version>3.8.1</version>
           <scope>test</scope>
   </dependency>
   <dependency>
           <groupId>log4j</groupId>
           <artifactId>log4j</artifactId>
           <version>${log4j.version}</version>
   </dependency>
   <dependency>
           <groupId>org.springframework</groupId>
           <artifactId>spring-core</artifactId>
           <version>3.0.0.RC3</version>
    </dependency>
</dependencies>
```

Compila tu proyecto, podrás ver que Maven descarga del repositorio que especificamos el jar. Podemos poner los repositorios que queramos, Maven buscará tus dependencias en todos ellos, empezando por el repo central y siguiendo con los que hayas declarado en el orden en que estén declarados.



Conclusión

Hemos visto en este tutorial cómo crear una aplicación con Maven. Recuerda que hemos estado usando en muchos casos la configuración por convención de Maven. Esta herramienta es mucho más flexible de lo que aquí se muestra. El mejor recurso para obtener documentación sobre esta herramienta es la guía de referencia que puedes encontrar en el sitio del proyecto (http://maven.apache.org) y el libro gratuito de la empresa Sonatype llamado "**Maven: The Definitive Guide**" (http://www.sonatype.com/products/maven/documentation/book-defguide). Ademàs en **javaHispano** (http://www.javahispano.org) estaremos listos para responder a tus dudas en los foros.