


Realizada la instalación de Maven, el paso siguiente es crear la estructura básica de trabajo Maven.

Para esto creamos un directorio que en mi caso llamare Curso\_maven.



```
C:\Windows\system32\cmd.exe
E:\>g:
G:\>cd curso_maven
El sistema no puede encontrar la ruta especificada.
G:\>mkdir curso_maven
G:\>cd curso_maven
G:\curso_maven>ls
"ls" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
G:\curso_maven>dir
El volumen de la unidad G es Verbatim
El número de serie del volumen es: B040-0D3C

Directorio de G:\curso_maven
26/05/2012  02:57    <DIR>          .
26/05/2012  02:57    <DIR>          ..
               0 archivos                0 bytes
               2 dirs 85.218.824.192 bytes libres
G:\curso_maven>
```

Desde el directorio ejecutaremos el siguiente comando.

El primer paso que podemos hacer con maven es **crear un proyecto** desde cero. El comando de **maven** que tenemos que ejecutar es

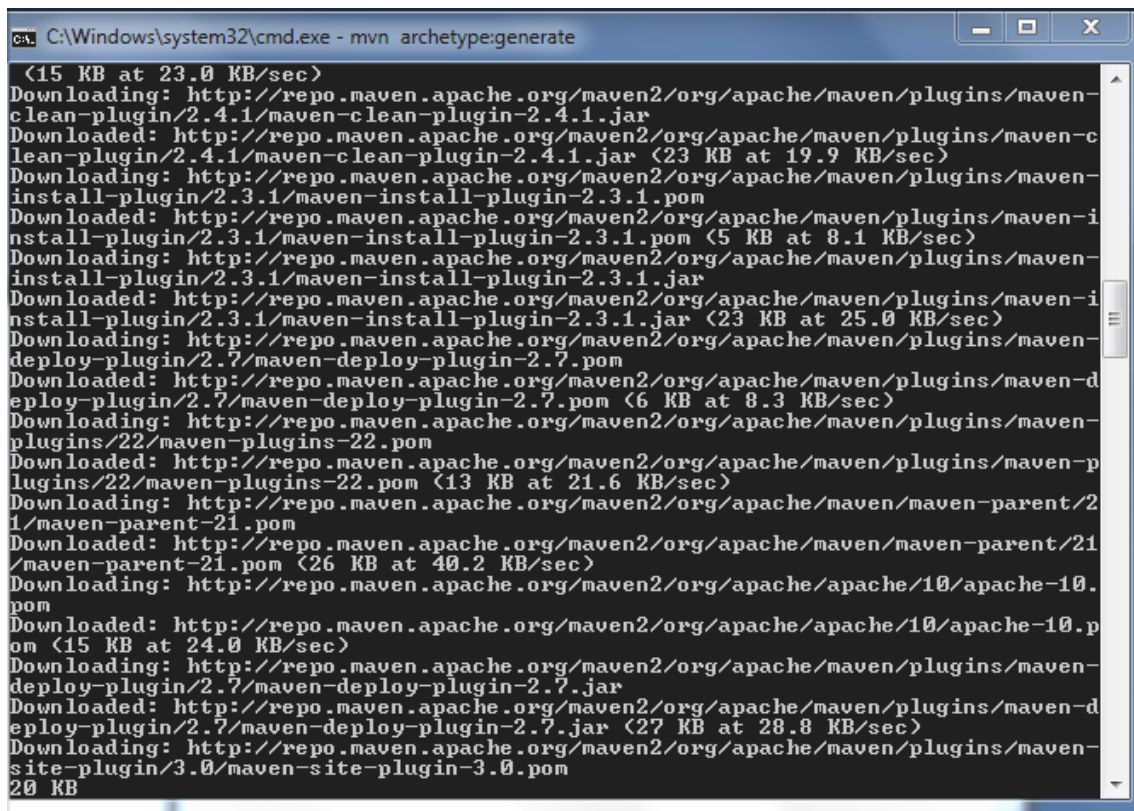
```
mvn archetype:create -DgroupId=organizacion.paquetes -
DartifactId=miproyecto
```

La primera vez que ejecutemos maven, **creará un repositorio local en tu disco duro**. En concreto, creará la carpeta *.m2* en la carpeta *home* del usuario. En ella se guardarán todos los artefactos que maneje maven. Cuando instalemos al manejo desde Maven, este quedara registrado en el repositorio, pero esto lo veremos con mayor detalle mas adelante.

**GroupId:** Es la compañía, organización, o equipo al que pertenece este proyecto. Una convención utilizada para el “groupId” es utilizar al comienzo el nombre de dominio de la organización. Ejemplo, un proyecto de apache, tendría como groupId, “org.apache”

**ArtifactId:** Es el identificador único de este proyecto dentro del grupo especificado en el groupId. Siguiendo con el ejemplo, Apache tiene una gran cantidad de proyectos distintos todos bajo el groupId “org.apache”, pero cada uno con su nombre único.

Una vez ejecutado este comando, **Maven** empezará a bajarse cosas de internet cuando lo ejecutemos por primera vez (en los próximos proyectos ya no necesita bajarse nada) y creará una estructura de directorios y ficheros como la siguiente



```
C:\Windows\system32\cmd.exe - mvn archetype:generate
<15 KB at 23.0 KB/sec>
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.4.1/maven-clean-plugin-2.4.1.jar
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.4.1/maven-clean-plugin-2.4.1.jar (23 KB at 19.9 KB/sec)
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-install-plugin/2.3.1/maven-install-plugin-2.3.1.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-install-plugin/2.3.1/maven-install-plugin-2.3.1.pom (5 KB at 8.1 KB/sec)
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-install-plugin/2.3.1/maven-install-plugin-2.3.1.jar
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-install-plugin/2.3.1/maven-install-plugin-2.3.1.jar (23 KB at 25.0 KB/sec)
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/maven-deploy-plugin-2.7.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/maven-deploy-plugin-2.7.pom (6 KB at 8.3 KB/sec)
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/22/maven-plugins-22.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/22/maven-plugins-22.pom (13 KB at 21.6 KB/sec)
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/21/maven-parent-21.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/21/maven-parent-21.pom (26 KB at 40.2 KB/sec)
Downloading: http://repo.maven.apache.org/maven2/org/apache/apache/10/apache-10.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/apache/10/apache-10.pom (15 KB at 24.0 KB/sec)
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/maven-deploy-plugin-2.7.jar
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/maven-deploy-plugin-2.7.jar (27 KB at 28.8 KB/sec)
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-site-plugin/3.0/maven-site-plugin-3.0.pom
20 KB
```

al finalizar su ejecución nos crea una estructura de directorio como la siguiente.

```
miproyecto
+---src
|   +---main
|       +---java    //Para nuestros fuentes
|           +---organizacion
|               +---paquetes
|                   +---App.java
|   +---test
|       +---java    //Para test de Junit
|           +---organizacion
|               +---paquetes
|                   +---AppTest.java
+---pom.xml
```

Dentro un fichero **pom.xml** que es un fichero que contiene datos de configuración de nuestro proyecto, como dependencias con otros jar, tipos de informes que queremos en la página web de nuestro proyecto, etc.. Inicialmente contiene una serie de cosas por defecto que podremos cambiar si lo deseamos.

Crea dos subdirectorios, uno **src** y otro **test**. Dentro de **src** debemos meter todo lo que sean fuentes y ficheros de configuración o datos propios del proyecto. En la parte de **test** debemos meter todos nuestros fuentes de prueba, clases de test de JUnit, ficheros de datos o de configuración de pruebas, etc. Es decir, en **src** va lo que es del proyecto y en **test** lo que nos ayude a probar el proyecto, pero no sea propio del proyecto.

Si vemos el fichero **pom.xml** de nuestro proyecto recién creado vemos lo siguiente

```
<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>organizacion.paquetes</groupId>
<artifactId>miproyecto</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<name>miproyecto</name>
<url>http://maven.apache.org</url>

<properties>
  <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

Pero analicemos un poco el archivo POM para entenderlo un poco mas, un **artefacto es un componente de software que podemos incluir en un proyecto como dependencia**. Normalmente será un **jar**, pero podría ser de otro tipo, como un **war** por ejemplo. Los artefactos pueden tener dependencias entre sí, por lo tanto, si incluimos un artefacto en un proyecto, también obtendremos sus dependencias.

En el ejemplo por defecto que genera podemos notar que registra:

- La información de base del manejo base del proyecto.
- Identifica el encoding para la ejecución.
- Proporciona información de las dependencias.

**Un grupo es un conjunto de artefactos.** Es una manera de organizarlos. Así por ejemplo todos los artefactos de Spring Framework se encuentran en el grupo **org.springframework**.

Veamos un ejemplo:

```

<dependency>
  <groupid>org.springframework</groupid>

```

```
<artifactid>spring-orm</artifactid>
<version>3.0.5.RELEASE</version>
<scope>runtime</scope>
</dependency>
```

Esta es la manera de declarar una dependencia de nuestro proyecto con un artefacto. **Se indica el identificador de grupo, el identificador del artefacto y la versión.**

El *scope* sirve para indicar el **alcance de nuestra dependencia** y su transitividad. Hay 6 tipos:

- **compile:** es la que tenemos por defecto sino 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.
- **test:** La dependencia es solo para testing que es una de las fases de compilación con maven. JUnit es un claro ejemplo de esto.
- **system:** Es como *provided* pero tienes 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
- **import:** este solo se usa en la sección *dependencyManagement*. Lo explicaré en otro artículo sobre transitividad de dependencias.

Por consiguiente si tuviéramos que agregar nuevos artefactos a nuevos proyecto de ejemplo de de seguro deberíamos agregar nuevas dependencias.

Pero continuemos analizando la generación inicial del proyecto Maven que tenemos como caso de estudio.

Este es el código fuente del App.java  
package organizacion.paquetes;

```
/**
 * Hello world!
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

**Nota:** Como podemos observar no hay mucha logica solo imprime un mensaje por pantalla

Este es el código fuente del AppTest.java, archivo para las pruebas de Junit.  
package organizacion.paquetes;

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 * Unit test for simple App.
 */
public class AppTest
    extends TestCase
{
    /**
     * Create the test case
     *
     * @param testName name of the test case
     */
    public AppTest( String testName )
    {
        super( testName );
    }
}
```

```

/**
 * @return the suite of tests being tested
 */
public static Test suite()
{
    return new TestSuite( AppTest.class );
}

/**
 * Rigourous Test :-)
 */
public void testApp()
{
    assertTrue( true );
}
}

```

**Nota:** Contiene los métodos necesarios para las pruebas del código fuente a probar.

Ahora ya habiendo analizado la generación de Maven, la pregunta es como realizamos acciones sobre el proyecto en cuestión. Aquí es donde introducimos el concepto de GOAL.

Un **Goal** no es mas que un comando que recibe maven como **parámetro** para que haga algo. La sintaxis sería:

```
mvn plugin:comando
```

Maven tiene una arquitectura de plugins, para poder ampliar su funcionalidad, aparte de los que ya trae por defecto.

Ejemplos de goals serían:

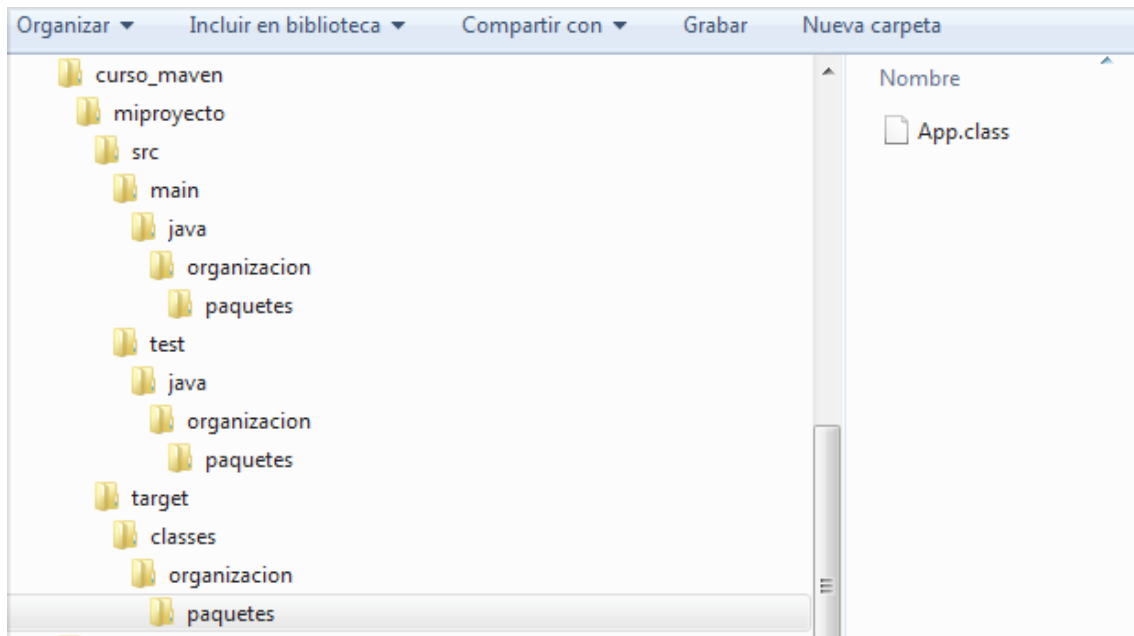
- **mvn clean:clean (o mvn clean):** limpia todas las clases compiladas del proyecto.
- **mvn compile:** compila el proyecto
- **mvn package:** empaqueta el proyecto (si es un proyecto java simple, genera un jar, si es un proyecto web, un war, etc...)

- **mvn install:** instala el artefacto en el repositorio local (/Users/home/.m2)

Ahora desde dentro del directorio del proyecto ejemplo de estudio y para retomar la practica, compilaremos con Maven con el siguiente comando.

```
mvn compile
```

**Nota:** con esta acción Maven genero los archivos class y agrego un nuevo directorio target para este fin.



A continuación generaremos el empaquetado jar con el siguiente comando en el directorio del proyecto.

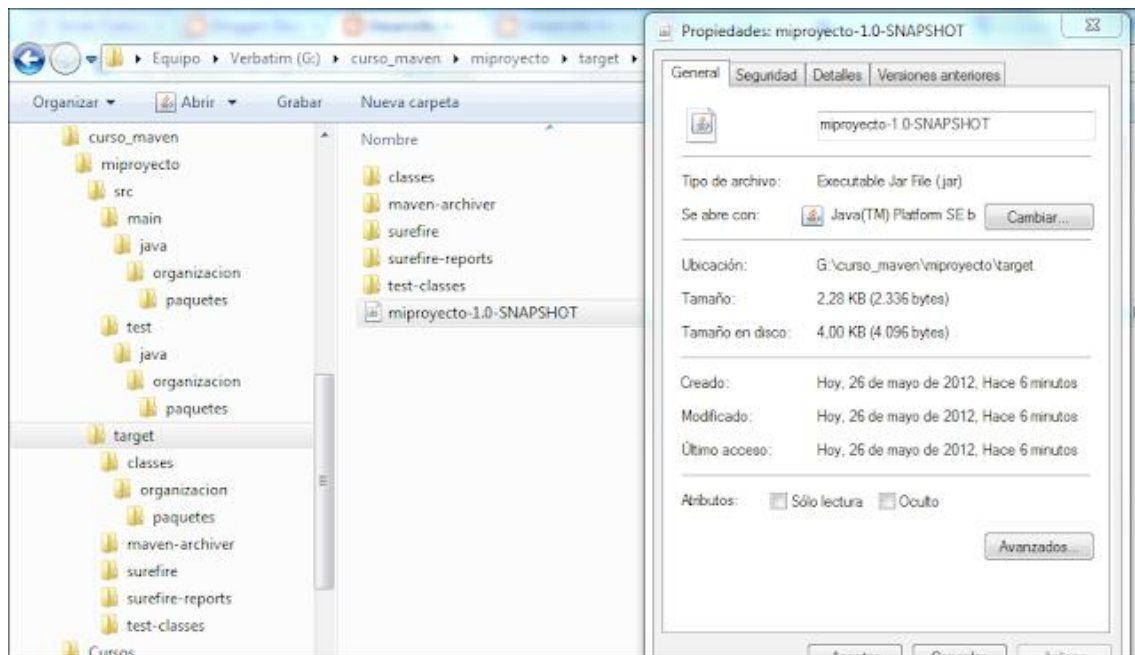


```
mvn package
```

```
C:\Windows\system32\cmd.exe
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @ miproyecto ---
[debug] execute contextualize
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory G:\curso_maven\miproyecto\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @ miproyecto ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.5:testResources (default-testResources) @ miproyecto ---
[debug] execute contextualize
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory G:\curso_maven\miproyecto\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @ miproyecto ---
[INFO] Compiling 1 source file to G:\curso_maven\miproyecto\target\test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.10:test (default-test) @ miproyecto ---
[INFO] Surefire report directory: G:\curso_maven\miproyecto\target\surefire-reports

-----
T E S T S
-----
Running organizacion.paquetes.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.012 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- maven-jar-plugin:2.3.2:jar (default-jar) @ miproyecto ---
[INFO] Building jar: G:\curso_maven\miproyecto\target\miproyecto-1.0-SNAPSHOT.jar
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 2.164s
[INFO] Finished at: Sat May 26 07:09:47 GMT 2012
[INFO] Final Memory: 3M/8M
[INFO]
G:\curso_maven\miproyecto>
```

**Nota:** donde podemos notar que en la consola nos indica la ruta donde genero el archivo JAR y casualmente es en la raiz del directorio target de la compilación.



ahora la prueba final es ejecutar este manejo desde la linea de comando para obtener la implementacion que en nuestro caso seria un simple mensaje por pantalla.

```
java -cp target/miproyecto-1.0-SNAPSHOT.jar organizacion.paquetes.App
```

y nuestro resultado final sera.

```
[INFO] --- maven-jar-plugin:2.3.2:jar (default-jar) @ miproyecto ---
[INFO] Building jar: G:\curso_maven\miproyecto\target\miproyecto-1.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.164s
[INFO] Finished at: Sat May 26 07:09:47 GMT 2012
[INFO] Final Memory: 3M/8M
[INFO] -----
G:\curso_maven\miproyecto>java -cp target/miproyecto-1.0-SNAPSHOT.jar organizacion.paquetes.App
Hello World!
G:\curso_maven\miproyecto>
```