¿Qué es Maven?



Brenda Souza

03/02/2023



Al desarrollar un proyecto de programación tenemos que utilizar, muchas veces, librerías externas al código actual para poder utilizar funciones que fueron creadas por otras personas y que es posible utilizar, haciendo referencia a su autor, ahorrando tiempo de desarrollo.

**Con Java esto no es diferente,** el lenguaje utiliza estas librerías con paquetes insertados en su proyecto Java, estos paquetes provienen de un origen conocido para que el proceso de importación sea posible. Generalmente, la librería se descarga y luego se importa al proyecto.

Sin embargo, dependiendo de la complejidad del proyecto, es necesario utilizar diferentes bibliotecas, lo que puede presentar un problema con la versión y/o compatibilidad con el código, esto pensado en un proyecto grande. Ahora resulta inviable realizar la importación y verificación de cada biblioteca manualmente.

**Aquí llega Maven**, una herramienta de gestión de dependencias que se creó en Java para ayudar con los proyectos creados en este lenguaje.

**Sobre Maven (breve historia)**

Para contar la historia que llevó a su creación, podemos mencionar dos herramientas hechas en Java: Apache Ant y Apache Ivy.

[Apache Ant](https://ant.apache.org/) : una herramienta de automatización de compilación en el desarrollo de aplicaciones, en otras palabras, ayuda al desarrollador a compilar los archivos del proyecto en el orden correcto según la dependencia entre los archivos. Este proyecto es software libre con licencia Apache.

[Apache Ivy](https://ant.apache.org/ivy/) : herramienta administradora de paquetes transitivos, Ivy es una subparte de Apache Ant y tiene su funcionalidad enfocada en resolver las dependencias de un proyecto y también de los repositorios JAR, sobre todo asiste en la integración y publicación de artefactos de los proyectos que están utilizando. También es un proyecto de software libre, bajo la misma licencia de Apache.

[Apache Maven](https://maven.apache.org/) es una herramienta de gestión y comprensión de proyectos así como Apache Ant (ambas en Java) pero tiene diferentes conceptos y funcionalidades, en especial podemos hablar de la base de Maven que el Modelo de Objeto de Proyecto, el modelo organiza toda la información del proyecto en un solo archivo: pom.xml.

Esta es la manera en que Maven realiza la construcción del proyecto, es decir, como dependencias son requeridas, POM se actualiza. El proyecto Maven puede poseer módulos y cada módulo puede tener su respectivo POM sin perder la organización y jerarquía del proyecto principal. También es posible realizar la construcción de informes y documentación con la herramienta.

Además, también es posible tener plugins que son fases individuales del proyecto. El ciclo de vida de la construcción de un proyecto también es un concepto muy importante en Maven, pero hablaremos de ello más tarde.

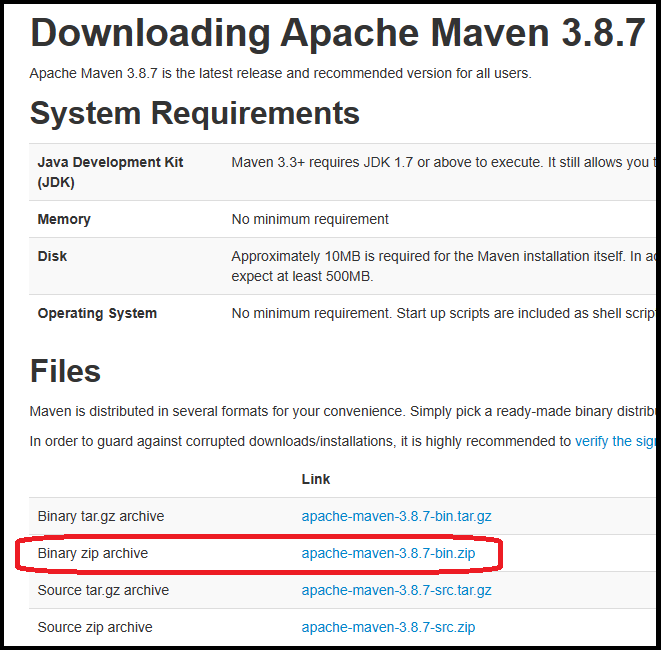
**Instalación de Maven**

En principio tenemos como requisito previo la instalación de JDK, especialmente en la versión 1.7 o superior. Después de obtener el JDK, independientemente del sistema operativo que esté utilizando, podemos continuar con la instalación de Maven. Si tienes dudas sobre cómo instalar JDK, tenemos una sugerencia de lectura que te puede ayudar: [Mi primer programa en Java](https://www.alura.com.br/artigos/me-primeiro-programa-em-java).

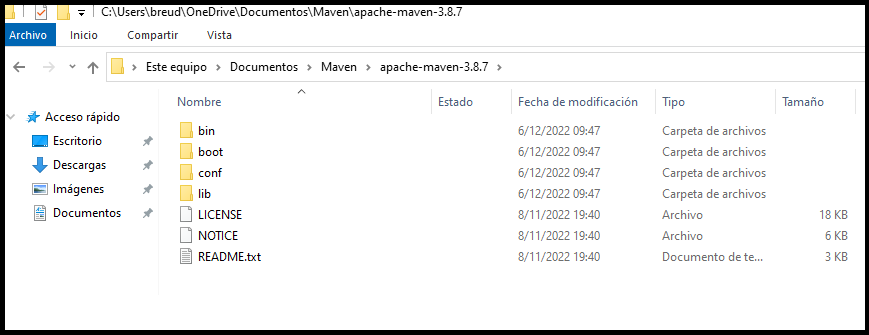
Acerca de [Maven](https://maven.apache.org/index.html), el propio sitio web siempre sugiere descargar la última versión, pero también es posible utilizar versiones anteriores. En particular, se sugiere de antemano toda la versión 3 ya que es una versión estable de esta herramienta.

**Instalación de Maven en Windows**

Para instalar Maven en Windows solo tiene que acceder a este [link](https://maven.apache.org/download.cgi) para poder descargarlo. Puede descargar el archivo binario o el código fuente como desee. En este artículo, vamos a utilizar la versión 3.8.7 en su formato binario.



Al finalizar la descarga del archivo, simplemente extraiga o descomprima la carpeta y almacene la carpeta en algún directorio de su máquina.



Note que en la carpeta resultante hay un archivo ejecutable titulado mvn, es el ejecutable de Maven. Con este archivo y directorio puedes gestionar Maven y sus proyectos como quieras, pero siempre tendrás que volver a este directorio para poder hacerlo.

Para poder usar Maven desde cualquier directorio de su máquina, debemos configurarlo como una variable de entorno. Para eso, sigue estos pasos:

* Abrimos el explorador de archivos de Windows.
* Seleccionamos la sección titulada 'Esta Computadora'.
* Damos clic derecho y seleccionamos Propiedades.
* Después de esto aparece una nueva pantalla y en ella seleccionamos la opción de 'Configuración avanzada del sistema'.
* Además, seleccionamos la sección 'Avanzado' y luego hacemos clic en el botón 'Variables de entornos...'.
* Añadimos el directorio C:/\*/apache-Maven-3.8.7 al final de la lista de directorios denominada M2\_HOME, para indicar cuál es.

Y, como se agregó como variable de entorno, también es necesario agregar su directorio bin a la variable PATH (variable de Windows que guarda los directorios de los archivos ejecutables del sistema):

* Sin embargo en la parte de la variable de entorno, seleccionamos la variable 'PATH' y presionamos el botón 'Editar'.
* Seleccionamos el botón 'Nuevo'.
* Hemos añadido el directorio C:/\*/apache-Maven-3.8.7/bin de Maven a la lista de directorios de PATH.
* Pulsamos Ok para finalizar la edición de PATH.
* Damos clic en Ok para finalizar la sección de variables de entorno.

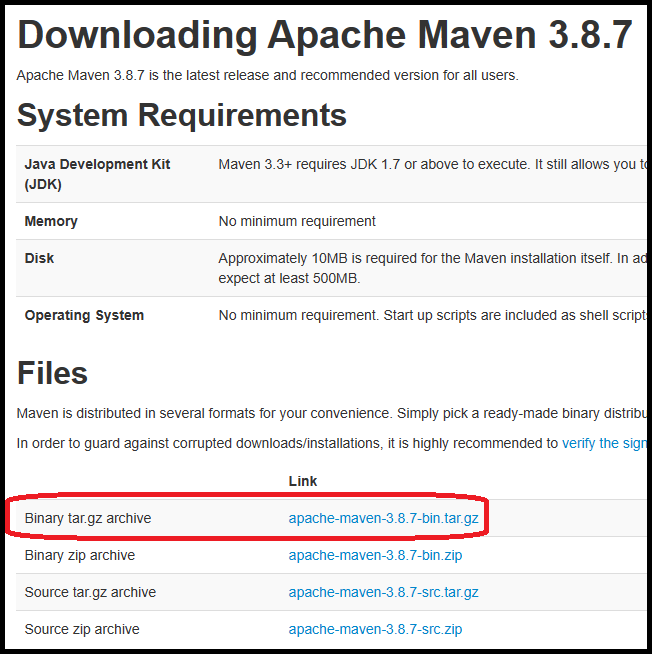
Finalmente, ejecutamos este comando para verificar su versión Maven en la máquina: mvn -version.

No mencionamos el JDK por nada, como dije anteriormente, Maven fue hecho en Java y para poder usarlo tenemos que tener el kit de desarrollo de Java presente en la máquina. Y el JDK definido como variable de entorno, puede usar Java desde cualquier directorio.

Los pasos para definir esta variable son los mismos que se realizaron para agregar la variable Maven, la diferencia es que el directorio utilizado es la ruta de la carpeta JDK.

**Instalación de Maven en Linux**

Para instalar Maven en Linux tenemos que descargar el archivo a través de la terminal de comandos. En este ejemplo, vamos a usar el archivo binario, pero puede usar el código fuente. Usemos la versión 3.8.7 en binario pero esta vez en formato .tar:



Sigamos los pasos para instalar Maven:

wget https://dlcdn.apache.org/maven/maven-3/3.8.7/binaries/apache-maven-3.8.7-bin.tar.gz

tar -xvf apache-Maven-3.8.7-bin.tar.gz

Tenga en cuenta que este enlace se obtiene copiando el enlace en el archivo seleccionado en la [página oficial](https://maven.apache.org/download.cgi) de Maven.

Ahora, definamos una variable de entorno para Maven accediendo al archivo $home/.profile y agregando este nombre de variable e información de directorio, guardamos y cerramos el archivo:

M2\_HOME='/opt/apache-Maven-3.8.7'

PATH="$M2\_HOME/bin:$PATH"

export PATH

Finalmente, ejecutamos este comando para verificar su versión Maven en la máquina: mvn -version.

**Instalación de Maven en MacOS**

Instalar Maven en MacOS funciona de manera similar a instalarlo en Linux, ya que estos son sistemas operativos basados en Unix, otro sistema operativo. Si puede usar el mismo archivo y formato, la exportación .tar se realiza de la misma manera. También usamos la versión 3.8.7 de Maven.

Lo que cambia es que en la parte de establecer una variable de entorno, debemos ir a la configuración del perfil o ejecutar $ source .bash\_profile para abrir el archivo que contiene las variables, para definir Maven como una variable.

Al abrir el archivo .bash\_profile, agregamos esta información, guardamos y cerramos el archivo:

export M2\_HOME='/opt/apache-Maven-3.8.7'

PATH="$M2\_HOME/bin:$PATH"

export PATH

Para verificar Maven en la máquina, haga lo mismo que en Linux: mvn -version.

**Ciclo de Vida de Construcción**

El concepto de Ciclo de Vida de Construcción es el concepto central de Maven, es la base para realizar el *Build* (traducción libre: Construcción) del proyecto Maven. Es donde utilizamos la idea de meta (o fase) para cada tarea que deseamos realizar, podemos tener más de meta en un comando, Maven ejecutará todas las metas secuencialmente.

Seguimos esta estructura para ejecutar metas en Maven con este comando (todos los comandos Maven son iguales para todos los sistemas operativos):

mvn [nombre de la meta 01] [nombre de la meta 02] [nombre de la meta 03]

Un comando puede tener uno o más objetivos, Maven se ejecutará según la secuencia definida.

Tenemos tres comandos principales de construcción:

* *default*: traducido del inglés 'estándar', es el comando que hace el *deploy* del proyecto, es decir, la implementación del proyecto en algún ambiente.
* *clean*: es el comando que limpia el proyecto, limpia la caché y otros datos generados por el proyecto;
* *site*: este es el comando que realiza la creación de la página web de su proyecto, que puede ser utilizada como documentación, por ejemplo. Este comando se utiliza luego de definir la estructura, el diseño y el dominio del sitio, esto se explica [aquí](https://maven.apache.org/guides/mini/guide-site.html).

Cada comando tiene una serie de comandos que se ejecutan para obtener el resultado final esperado, se puede ver el ejemplo del comando *default* en este [link](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#a-build-Lifecycle-is-made-of-phases).

Otros comandos importantes para el desarrollo de proyectos son:

* *install*: esta orden instala el proyecto en su repositorio local, añadiendo todas las dependencias especificadas en pom.xml del proyecto.
* *package*: este comando coloca el código compilado del proyecto en un paquete, es decir, en un formato que se puede distribuir para su uso. Existen diversos formatos, pero los más utilizados son JAR y WAR.
* *test*: este comando realiza las pruebas unitarias definidas en la carpeta src/test de su proyecto.

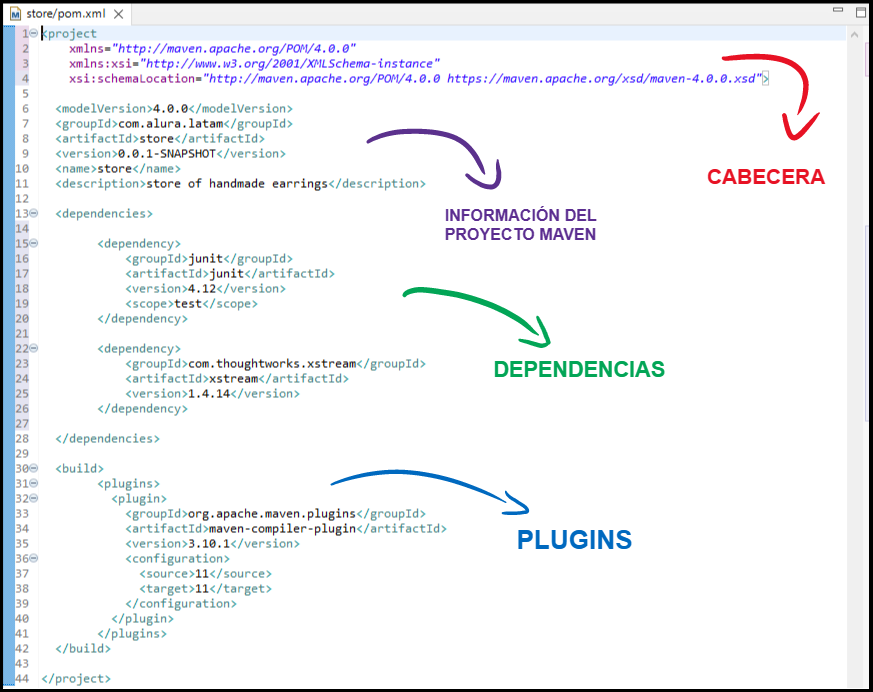
También es posible realizar estos objetivos a través de [plugins](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#plugins), que son artefactos que brindan etapas a Maven, para que las tareas se puedan realizar de esta manera.

Estos comandos también se pueden realizar a través de IDE, como Intellij y Eclipse, pero a veces hay un problema de compatibilidad y los comandos no se ejecutan de esa manera. Por eso, siempre es bueno saber cómo hacerlo según la documentación oficial, en este caso, cómo hacerlo a través del terminal.

**El archivo pom.xml**

El archivo POM es la base de un proyecto Maven, en este archivo tenemos toda la información sobre el proyecto y las dependencias que son importantes para la ejecución del proyecto. Es un archivo que puede y debe actualizarse según sea necesario, simplemente realizando la actualización del proyecto a través de Maven (vía terminal o IDE, lo que prefiera).

El ejemplo utilizado es una tienda de pendientes artesanales implementada en Java, donde tenemos una clase Producto con unos atributos: id, nombre, cantidad (inventario) y precio unitario. Para ello tenemos configurado el archivo pom.xml con la información importante del proyecto:



El encabezado del proyecto puede parecer confuso, pero se puede entender de la siguiente manera: el elemento project tiene el atributo xsi con este *namespace*, y xsi tiene un esquema definido; finalmente, el elemento debe ser validado según el esquema especificado. Este es un patrón para archivos XML.

<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/4.0.0.0. https://maven.apache.org/xsd/maven-4.0.0.xsd">

Ahora, vayamos a la información del proyecto, es decir, características que definen el proyecto no solo a nivel Build, pero también pensemos en la distribución del proyecto.

<modelVersion>4.0.0</modelVersion>

<groupid>com.alura.Latam</groupid>

<artifactId>store</artifactId>

<versión>0.0.1-SNAPSHOT</versión>

<name>store</name>

<Description>store of handmade earrings</Description>

Comencemos con la información necesaria sobre el proyecto:

* *project*: es la etiqueta raíz que agrupa la información del proyecto.
* *modelVersion*: se refiere al modelo POM compatible con Maven, por lo que siempre se define como 4.0.0 Pues es la actual versión de POM compatible y que tiene soporte para Maven.
* *groupid*: se refiere al grupo/empresa al que está asociado al proyecto.
* artifactId: se refiere al nombre de su proyecto, la identificación del proyecto.
* versión: se refiere a la versión actual del proyecto (según su groupid), pues en proyectos grandes acaba ocurriendo diferentes versiones, cada versión con dependencias específicas, entonces el POM lo considera eso también.
* name: nombre, según la definición en artifactId.
* description: descripción del proyecto en un texto breve.

Tenga en cuenta que puedes agregar más de una dependencia al proyecto. Maven tiene un [repositorio con dependencias](https://mvnrepository.com/) disponible para agregar al proyecto. Cada dependencia tiene una o más subsecuentes, y en caso de que el desarrollador desee no utilizar alguna de sus posteriores es posible realizar la exclusión utilizando la etiqueta [exclusions](https://maven.apache.org/pom.html#exclusions).

<dependencies>

<dependency>

<groupid>junit</groupid>

<artifactId>junit</artifactId>

<versión>4.12</versión>

<Scope>test</Scope>

</dependency>

<dependency>

<groupid>com.thoughtworks.Xstream</groupid>

<artifactId>xstream</artifactId>

<versión>1.4.14</versión>

</dependency>

</dependencies>

En este ejemplo tenemos dos dependencias: JUnit y XStream. [JUnit](https://junit.org/junit5/) es un framework creado para la ejecución de pruebas unitarias de un proyecto Java. Ya el [XStream](https://x-stream.github.io/) es una biblioteca construida en Java para realizar la serialización de objetos a XML y viceversa.

Podemos ver cómo se utilizan estas dependencias en el proyecto, en este caso nos centraremos en el uso de JUnit. Para utilizar JUnit tenemos que utilizar la anotación @Test para indicar que el método creado se usará para pruebas. En este caso, tenemos una clase ProductService con dos métodos: un método para verificar si el producto tiene atributos nulos, y otro método para agregar el producto a una lista de productos de la tienda.

package com.alura.Latam.service;

import java.util.ArrayList;

import java.util.List;

import com.alura.Latam.model.Product;

public class ProductService {

List<Product> listProducts = new ArrayList<Product>();

public Boolean checkProduct(Product product) {

if( product.getId() != 0

&& product.getName() != null && product.getName() != "

&& product.getUnitPrice() != 0 ){

Return true;

}

Return false;

}

public Boolean addProduct(Product product) {

if(checkProduct(product)) {

listProducts.add(product);

Return true;

}

Return false;

}

}

Dado los métodos, ahora vamos a la carpeta src/test/java que se encarga de todos los archivos Java relacionados con las pruebas del proyecto. En esta carpeta tenemos la clase ProductValidator con métodos para agregar un producto nulo y otro **no** nulo:

package com.alura.Latam.test;

import org.junit.Test;

import com.alura.Latam.model.Product;

import com.alura.Latam.service.ProductService;

public class ProductValidator {

ProductService productService = new ProductService();

@Test

public void addProductNotNull() {

Product example = new Product(1,"Golden Earring", 30, 7.5);

productService.addProduct(example);

}

@Test

public void addProductNull() {

Product example2 = new Product();

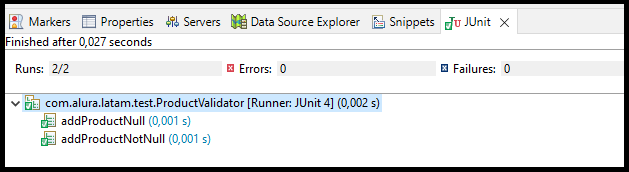
productService.addProduct(example2);

}

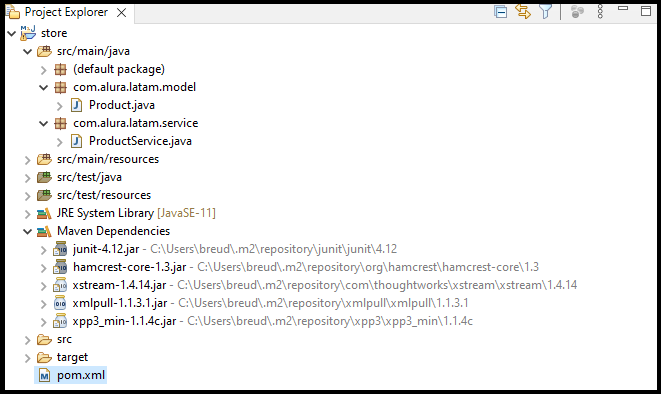
}

En ambos métodos creamos objetos de Product con valores fijos. Como el ejemplo es una tienda de pendientes, el primer método addProductNotNull() crea un producto descrito como 'pendiente dorada' (en inglés) e intenta agregar el producto; el segundo método crea un producto nulo. Con los métodos definidos, podemos ejecutar las pruebas utilizando el comando Maven mvn test .

Teniendo en cuenta lo que se ha implementado, el resultado esperado es que ambas pruebas se completen con éxito, como indica la interfaz de pruebas del IDE:



Además, tenemos información similar a la información de nuestro proyecto, con el atributo adicional Scope que hace referencia al alcance del proyecto, en el caso de JUnit, el alcance es de pruebas. Cada dependencia también tiene sus propias dependencias, por lo que puede ver las dependencias posteriores junto a ellas:



El concepto de dependencias es algo sólido en el desarrollo de proyectos que puede quitarle la preocupación al desarrollador de tener que descargar manualmente todo lo que necesita.

Maven descarga lo que se necesita y, además, actualiza la versión de dependencia simplemente cambiando el valor asociado a la versión y actualiza el proyecto con el comando mvn clean install, y ya está actualizado sin que la persona tenga ese esfuerzo manual. Al final, su proyecto tiene todos los JAR necesarios para su ejecución.

Por último, tenemos el último bloque del archivo que se refiere a los **plugins** utilizados en el proyecto.

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.10.1</version>

<configuration>

<source>11</source>

<target>11</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

Tenga en cuenta que este archivo se cierra después de la etiqueta '<build>', con el '</project>'. La parte de build almacena los plugins que usa el proyecto de acuerdo con la definición del desarrollador.

En nuestro ejemplo tenemos el [plugin de compilación de Maven](https://maven.apache.org/plugins/maven-compiler-plugin/) y tiene la función de compilar todos los archivos fuente del proyecto. Tenga en cuenta que también tiene groupId y artifactId según los elementos anteriores, y también tiene su versión de uso.

El último detalle es la etiqueta de configuración: en ella hemos definido la versión de Java - en este caso Java 11 - que compilará los archivos fuente del proyecto y archivos target (los generados por el proyecto).

**Plugins**

En principio el concepto de plugin es un código utilizado para mejorar proyectos de software. Y el propio Maven dice que su fuente está basada en plugins, no es extraño que en la documentación oficial existe una [lista de plugins disponibles](https://maven.apache.org/plugins/index.html).

Maven difierencia los plugins entre los de *build*: que se centran en el *build* del proyecto y se especifican en la etiqueta <build> del archivo pom.xml y los plugins de reporte que se utilizan durante la generación del sitio a través de Maven (¿recuerdas el comando? mvn site), estos están definidos en la etiqueta <Reporting> del pom.xml.

Son los plugins que nos ayudan a realizar tareas de construcción y configuración de proyectos. Target en el proyecto es el directorio de build, por lo que los targets de un plugin son tareas de build realizadas para componer la configuración y compilación del proyecto. Recuerde que cada plugin sigue esta sintaxis para ejecutar mediante comando:

mvn [nombre del complemento]:[target]

Un ejemplo genial de Plugin es [Archetype](https://maven.apache.org/archetype/maven-archetype-plugin/) con su target create, que genera un prototipo de un proyecto Maven a partir de un modelo existente, un arquetipo. Puede usar el plugin a través del comando:

mvn archetype:create

Después de este comando, Archetype solicita información de su proyecto como el tipo de proyecto y la información inicial que vimos en la sección sobre pom.xml para finalizar la generación de toda la estructura con sus correspondientes carpetas.

En comparación con Apache Ant, el valor predeterminado al usar targets es diferente. En Ant, se crea un *target* que llama a *javac,* mientras que en Maven usamos un target existente.

En Ant, es el usuario quien ejecuta los comandos en el archivo build.xml como lenguaje de programación, mientras que Maven tiene pom.xml como una estructura bien definida donde el build se crea con esa estructura y declaraciones, algo más autónomo.

Por último, cabe señalar que estas tareas terminan generando archivos compilados del proyecto y todos ellos se almacenan en la carpeta target, sin necesidad de una configuración previa y un manual realizado por el desarrollador.

**Integración con lenguajes y frameworks**

Maven se usa principalmente en proyectos Java, pero también se utiliza mucho para proyectos realizados con el framework Spring, un ecosistema hecho en Java con varios segmentos esenciales para el desarrollo de proyectos. El archivo POM junto con la administración de dependencias es una estructura conveniente para administrar un proyecto que utiliza Spring.

Por último, aunque Maven se ha hecho en Java y muchos proyectos Java siguen usándolo, también lo usan otros lenguajes como: C#, Ruby, Scala, y otros.

**Conclusión**

En este artículo vimos la historia de Maven, cómo instalar, cómo utilizar el archivo de Build hecho en XML llamado POM, que utiliza el Modelo de Objeto del Proyecto para estructurar en XML las informaciones y dependencias esenciales para el proyecto.

Hemos visto sobre el Ciclo de Vida de Construcción que Maven utiliza para administrar el proyecto utilizando tareas fragmentadas para que esas tareas - también vistas como fases - sean independientes entre sí.

Además, también hemos visto cómo Maven se diferencia de las herramientas anteriores, mostrando una mejora significativa en la gestión de paquetes y automatización de proyectos; en particular, hemos visto que con Maven es posible generar la documentación del proyecto directo a través del comando mvn site.

Maven es muy utilizado en el entorno Java, por lo que también hemos visto que además de proyectos puramente en Java, Maven también está presente en proyectos Java con Spring, un framework en Java que plantea distintos segmentos para el desarrollo de proyectos.