Material descargable

egg

Teoría **JAVA VIII**

Maven

En esta clase de introducción a **Apache Maven** exploraremos las diversas facetas de esta potente herramienta que nos permite simplificar y seguir el estándar de los procesos de construcción de software.

Maven es una herramienta de construcción de proyectos y gestión de dependencias ampliamente utilizada en la comunidad Java. Está diseñada para manejar el ciclo de vida completo de un proyecto de software, desde la etapa de compilación hasta la etapa de prueba, empaquetado, despliegue y más allá.

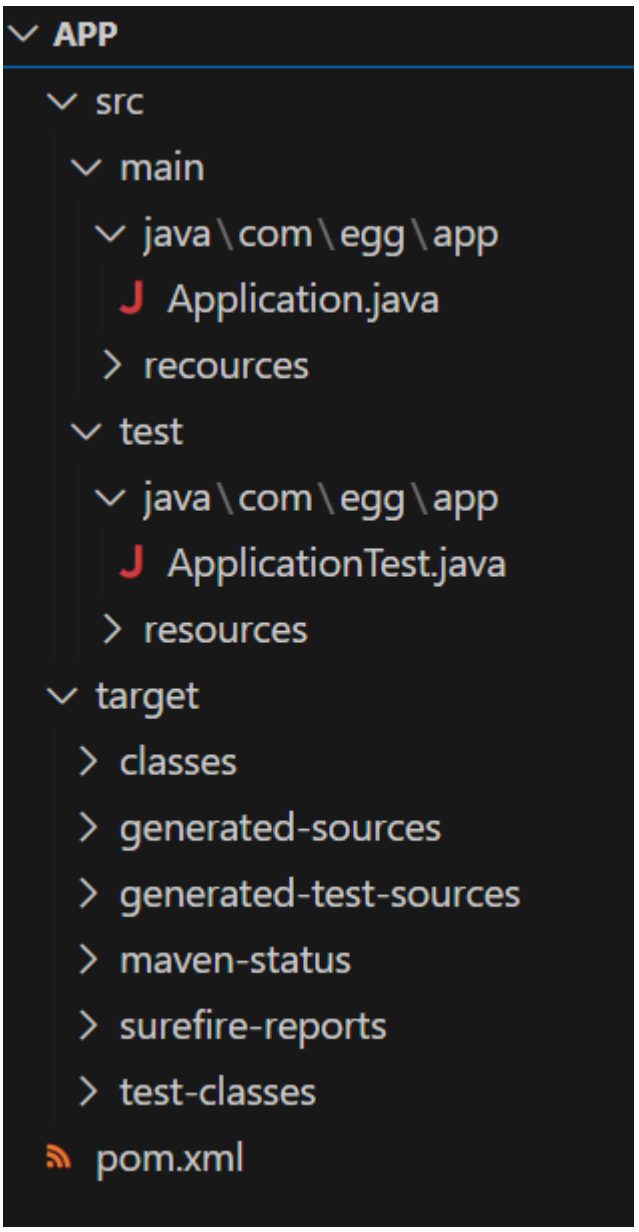
Estructura

Hasta ahora creábamos archivos Java en un directorio y cambiamos los ejercicios con el nombre del archivo Java.

En la sección de testing aprendimos a ser un poco más ordenados y separamos los ejercicios en una carpeta *src/* (source/fuente) y los test en una carpeta *test/*.

Ahora que ya estamos más familiarizados con Java, podemos dar un paso más y adoptar el estándar de estructuración de proyectos con Maven.

Cuando creamos un proyecto con Maven este tiene que seguir una estructura de directorios:



Como puedes observar ahora tenemos una carpeta con el nombre de la aplicación y dentro un archivo pom.xml de configuración y luego subdirectorios para ordenar nuestros archivos, veamos para qué es cada uno de ellos:

pom.xml - Este es el archivo de configuración principal de un proyecto Maven. Contiene información sobre el proyecto, incluidas las dependencias, los plugins de Maven que se utilizan, los perfiles de construcción, entre otras cosas (ahora daremos más detalle sobre esto).

src/main/java/com/my-company/app/ - Este es el directorio donde se colocan todos los archivos de código fuente de la aplicación. (my-company y app tienen

que ser reemplazados por los nombres que correspondan, daremos más detalles sobre esto).

src/main/resources - Este es el directorio donde se colocan todos los recursos de la aplicación, como archivos de configuración y propiedades (por ahora no lo usaremos).

src/test/java/com/my-company/app/ - Este es el directorio donde se colocan todos los archivos de código fuente de prueba. Estas son pruebas unitarias que se ejecutan para verificar la lógica en tus clases.

src/test/resources - Este es el directorio donde se colocan todos los recursos de prueba, como los archivos de configuración de prueba (aquí podemos poner nuestros archivos CSV).

target/ - Este es un directorio donde Maven coloca archivos que se generan automáticamente como los archivos de compilación del proyecto u otros que se crean por medio de plugins.

target/classes - Este es el directorio donde Maven coloca los archivos .class compilados cuando se construye el proyecto.

Archivo pom.xml

Como dijimos anteriormente, este es un archivo de configuración donde pondremos toda la información que necesita Maven para poder funcionar bajo enfoque de **Convención sobre Configuración**.

Convención sobre configuración

La **"Convención sobre Configuración"** es una forma de abordar la configuración de proyectos que promueve la simplicidad, la consistencia y la estandarización mediante el uso de convenciones predefinidas.

En lugar de requerir una configuración exhaustiva y personalizada para cada aspecto de un proyecto, Maven establece convenciones y estructuras por defecto que se deben seguir. Esto evita la necesidad de especificar configuraciones detalladas y permite a los desarrolladores concentrarse en la lógica de su aplicación en lugar de en la configuración. Esto tiene varios beneficios:

👌 **Menos código de configuración:** Maven proporciona un conjunto de convenciones que se aplican automáticamente a tu proyecto, lo que reduce la cantidad de configuración manual necesaria.

👌 **Configuración consistente:** Al seguir las convenciones, todos los proyectos de Maven tienen una estructura y configuración similar. Esto facilita la comprensión y colaboración en proyectos compartidos.

👌 **Mejor interoperabilidad:** Al utilizar convenciones comunes, los proyectos de Maven son más inter operables. Esto significa que es más fácil integrar proyectos de Maven con herramientas y sistemas externos.

👌 **Productividad mejorada:** Al no tener que configurar cada aspecto del proyecto, puedes centrarte en el desarrollo y la lógica de tu aplicación, lo que aumenta la productividad.

Estructura

Aquí tienes un archivo **pom.xml** base para poder crear una aplicación Java sin dependencias:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany</groupId>
    <artifactId>app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
    <java.version>20</java.version>
        <maven.compiler.source>${java.version}</maven.compiler.source>
        <maven.compiler.target>${java.version}</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- aquí se encontraron tus dependencias -->
    </dependencies>
```

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>${maven.compiler.source}</source>
                    <target>${maven.compiler.target}</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Como verás, este tipo de archivo es parecido a HTML, pero su extensión es **XML** y sus siglas significan e**X**tensible **M**arkup **L**anguage, es un lenguaje de marcado que se utiliza para codificar documentos de manera que sean legibles tanto por máquinas como por humanos. A diferencia del HTML, que tiene un conjunto definido de etiquetas para describir la presentación de la información, XML permite a los desarrolladores definir sus propias etiquetas para describir la estructura y el significado de la información.

Veamos para qué significan sus etiquetas:

<project>: Esta es la etiqueta raíz en el archivo pom.xml. Todos los demás elementos están anidados dentro de esta etiqueta. Los atributos xmlns, xmlns:xsi y xsi:schemaLocation proporcionan información sobre el espacio de nombres y el esquema XML.

- **Esquema XML**: Un esquema XML proporciona un plano que describe la estructura y los tipos de datos de un documento XML. Puedes considerarlo como un conjunto de reglas que un documento XML debe seguir para ser considerado válido.
- **Espacio de nombres XML**: Los espacios de nombres XML son una manera de evitar conflictos de nombres en los documentos XML. Cuando trabajas

con XML, puede que estés combinando documentos XML o usando etiquetas definidas en varios lugares diferentes. Esto puede causar problemas si dos diferentes etiquetas tienen el mismo nombre, pero se supone que deben tener diferentes significados.

<modelVersion>: Esta etiqueta define la versión del modelo de objeto de proyecto (POM) que se está utilizando. En este caso, se está empleando la versión 4.0.0 de la especificación POM que se introdujo con Maven 2.0 publicado en 2005, y no hay planes de cambiarlo.

👉 *Nota: Esta etiqueta y las anteriores son iguales en todos los proyectos Maven modernos, así que no te preocupes mucho en entender el esquema y espacio de nombres XML, céntrate más en entender las siguientes etiquetas.*

<groupId>: Esta etiqueta identifica el grupo o la organización a la que pertenece el proyecto. Por lo general, se utiliza el nombre de la empresa o el del paquete raíz del proyecto en formato invertido. Por ejemplo, *egg.com* al escribirlo como groupId sería *com.egg.*

<artifactId>: Esta etiqueta identifica el artefacto (es decir, el proyecto) dentro del grupo. Esto es fundamentalmente el nombre de tu proyecto. Por ejemplo, *app*.

<version>: Esta etiqueta define la versión del proyecto. Si usas Git puedes ir cambiando este valor sin miedo a perder las versiones anteriores.

<properties>: Esta etiqueta se utiliza para definir propiedades que se pueden reutilizar en el archivo pom.xml. En este caso, se están definiendo las propiedades maven.compiler.source y maven.compiler.target para ser reutilizadas en la configuración del plugin del compilador. El formato para luego invocar estas propiedades en el documento sería ${nombre.de.la.etiqueta}

<dependencies>: Esta etiqueta contiene todas las dependencias que necesita tu proyecto. Una dependencia se especifica utilizando las etiquetas <groupId>, <artifactId> y <version>.

<build>: Esta etiqueta se utiliza para especificar detalles de construcción del proyecto, como los plugins de Maven que se deben utilizar.

<plugins>: Esta etiqueta se utiliza para enumerar los plugins que se utilizarán durante la construcción del proyecto.

<plugin>: Esta etiqueta se utiliza para especificar un plugin que se utilizará durante la construcción del proyecto. Dentro de esta etiqueta, se especifica el <groupId>, <artifactId> y <version> del plugin, así como cualquier configuración específica del plugin en la etiqueta <configuration>. La configuración específica del plugin incluye etiquetas como <source> y <target>, que aquí se utilizan para especificar la versión de Java para compilar el código fuente y la versión de la JVM en la que se espera que se ejecute el código.

Dependencias

Una dependencia en Maven es otro proyecto o biblioteca necesario para compilar, probar, o ejecutar tu proyecto. Maven se encarga de descargar e integrar estas dependencias en el proceso de construcción de tu proyecto, es decir, no tienes que descargar a mano las librerías de tu proyecto, solo tienes que declararlas en tu pom. Esto también es muy útil cuando necesitas compartir tu proyecto con otros programadores, ya que no tienen que buscar las librerías usas en el mismo, solo tienen que correr el proyecto y Maven se encargará de buscar y descargar las dependencias (librerías) necesarias.

Por ejemplo, para añadir la biblioteca JUnit 5 (utilizada para pruebas unitarias en Java), agregarías la siguiente dependencia en tu pom.xml:

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.9.3</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Fíjate que para declarar una dependencia estás usando la **Convención de Nombres para Dependencias** (la combinación de groupId, artifactId y versión) de la misma manera que los usamos para declarar la identificación de nuestro proyecto, en otras palabras, nuestro proyecto puede ser usado como

dependencia de otro y automáticamente Maven descargará sus dependencias transitivas.

Las dependencias transitivas son las dependencias de tus dependencias. Por ejemplo, si tu proyecto A depende del proyecto B, y B depende del proyecto C, entonces C es una dependencia transitiva de A. Maven maneja automáticamente las dependencias transitivas para asegurarse de que todas las bibliotecas necesarias estén disponibles para tu proyecto.

¿De dónde se descargan estas dependencias?

Por defecto, Maven busca las dependencias en el repositorio central de Maven, un repositorio público donde muchos proyectos de código abierto publican sus librerías.

Repositorio central de Maven

Nota: Es posible configurar repositorios externos adicionales o alternativos si la dependencia no se encuentra en el repositorio central, pero esto no lo abordaremos en el curso.

Las descargas de estas dependencias se realizan cuando se ejecuta por primera vez un comando de Maven que requiere esas dependencias, como "mvn compile", "mvn package", entre otros. Por ejemplo, si estás compilando tu proyecto por primera vez o agregaste una nueva dependencia en tu archivo pom.xml, Maven se encargará de descargar todas las dependencias necesarias antes de compilar el proyecto.

¿Dónde se guardan estas dependencias?

No se guardan en el directorio del proyecto. En realidad, Maven mantiene un repositorio local en tu sistema de archivos para todas las dependencias y plugins necesarios para tus proyectos. Por defecto, este repositorio local se ubica en tu directorio de usuario bajo el path .m2/repository (en Windows podría ser C:\Users\TuUsuario\.m2\repository).

Este enfoque tiene una serie de beneficios. En primer lugar, no necesitas descargar la misma dependencia una y otra vez para cada proyecto que lo requiere, lo que te ahorra tiempo y ancho de banda. Además, como las dependencias se mantienen fuera de los directorios del proyecto, puedes

mantener tus proyectos más limpios y ligeros, lo cual es especialmente beneficioso si tienes que compartir el código de tu proyecto o trabajar en equipo.

Alcances (Scopes): La etiqueta scope de las dependencias determina en qué etapas del ciclo de vida de construcción se requiere la dependencia. Los alcances más comunes son:

compile (por defecto): La dependencia es necesaria en todas las etapas del ciclo de vida.

test: La dependencia es solo necesaria para compilar y ejecutar pruebas.

runtime: La dependencia no es necesaria para compilar, pero se necesita para ejecutar.

provided: La dependencia es necesaria para compilar el proyecto, pero se espera que la JVM o el contenedor proporcionen la dependencia en tiempo de ejecución. Es comúnmente usado en proyectos Java EE.

system: Similar a provided pero debes especificar la ruta del archivo JAR en tu sistema.

Nota: no tienes que preocuparte por pensar cuál debes usar, se te indicará el scope siempre que sea necesario.

Plugins, goals, ciclo de vida y comandos

Plugin: Es una pieza de software que añade una funcionalidad específica a otra pieza de software. En el caso de Maven, los plugins añaden funcionalidades al proceso de construcción de Maven, permitiendo que Maven compile código, cree documentación, ejecute pruebas, y más, están generalmente escritos en Java. Pero nosotros no crearemos plugins, usaremos los que ya existen y sean necesarios para nuestros proyectos.

Goals: Un goal es una tarea específica que realiza un plugin de Maven. Cuando ejecutas un comando como mvn compile, estás diciendo a Maven que ejecute el goal compile del plugin del compilador.

Ciclo de vida: Maven define tres ciclos de vida principales: "default", "clean" y "site". Cada ciclo de vida está compuesto por una serie de fases principales y secundarias.

- **Ciclo de vida Default**: Es el ciclo de vida principal y se ejecuta si no se especifica ningún otro ciclo de vida. Incluye las siguientes fases principales:
    - **Validate**: Verifica que el proyecto es correcto y que toda la información necesaria está disponible. Resultado visible: En la consola, simplemente no se mostrarán errores si la validación es exitosa.
    - **compile**: Compila el código fuente del proyecto. Resultado visible: los archivos de clase compilados en el directorio target/classes.
    - **test**: Ejecuta las pruebas utilizando un framework de pruebas adecuado. Estas pruebas no necesitan un paquete o un proyecto desplegado. Resultado visible: si las pruebas son exitosas, no deberías ver ningún cambio en los directorios pero en la consola deberías ver un resumen del resultado de las pruebas.
    - **package**: Empaqueta el código en un formato distribuible, como JAR o WAR. Resultado visible: un archivo JAR o WAR en el directorio target.
    - **verify**: Ejecuta cualquier verificación de control de calidad en el paquete. Por ejemplo, puedes tener reglas que dicen que todas las pruebas deben pasar antes de hacer un despliegue. Resultado visible: si las verificaciones son exitosas, no deberías ver ningún cambio en los directorios. En la consola, deberías ver un mensaje que indica que la verificación fue exitosa.
    - **install**: Instala el paquete en el repositorio local de Maven, para su uso como dependencia en otros proyectos localmente. Resultado visible: el archivo JAR o WAR del proyecto en tu repositorio local de Maven.
    - **deploy**: Copia el paquete final al repositorio remoto para compartirlo con otros desarrolladores y proyectos. Resultado visible: Es necesario realizar algunas configuraciones para usar un repositorio remoto (y no abordaremos este apartado en el curso) por lo tanto obtendrás un error indicando que no se pudo implementar o enviar los artefactos al repositorio remoto.

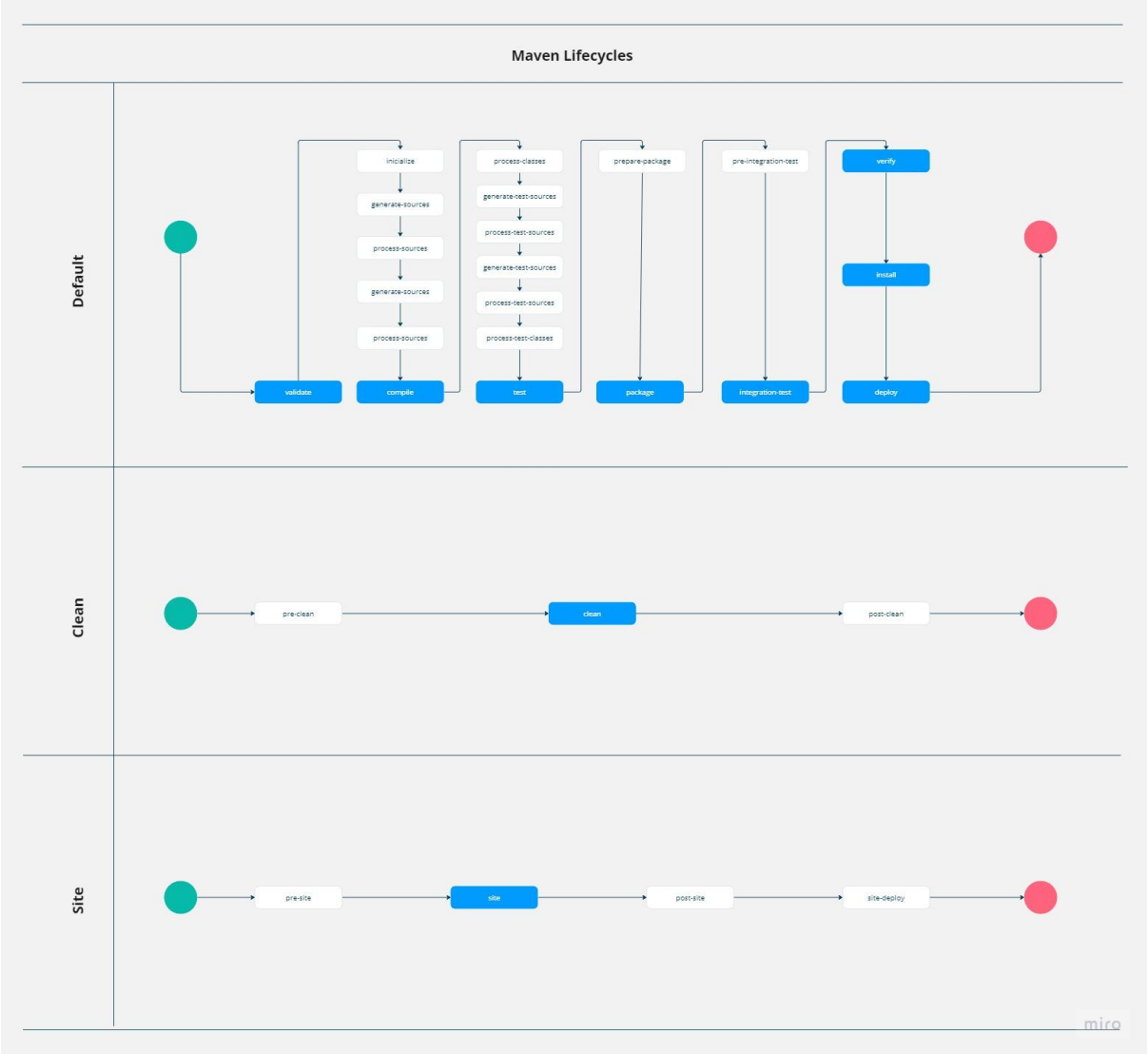
Nota: No abordaremos la configuración de un repositorio remoto de Maven para usar *mvn deploy*. Cuando necesitemos compartir nuestro proyecto usaremos Git.

Ejecutar una fase también ejecutará todas las fases que la preceden. Por ejemplo, mvn install ejecutará validate, compile, test, package, verify e install.

- **Ciclo de vida Clean**: Este es el ciclo de vida que se encarga de limpiar el proyecto y eliminar todos los archivos generados anteriormente, su fase principal es:
    - **clean**: Elimina la carpeta *target/*. Resultado visible: el directorio target debería estar vacío o eliminado.
- **Ciclo de vida Site**: Este es el ciclo de vida de generación de documentación. Genera un sitio web o una documentación para tu proyecto, su fase principal es:
    - **site**: Genera la documentación. Resultado visible: verás un nuevo directorio target/site que contiene un sitio web con documentación para tu proyecto.

Estos comandos se ejecutan en la línea de comandos en el directorio raíz del proyecto donde se encuentra el archivo pom.xml. Para ejecutar una fase, utilizas el comando mvn seguido del nombre de la fase, como mvn compile, mvn test, mvn package, etc. También puedes combinar *clean* con *compile* ejecutando *mvn clean compile* para limpiar tus archivos compilados y volver a compilarlos desde cero.

Las dependencias y los plugins interactúan con el ciclo de vida de Maven en la medida en que las dependencias se necesitan en diferentes fases del ciclo de vida de Maven, y los plugins se utilizan para ejecutar goals/tareas específicas en diferentes fases del ciclo de vida.



[Link a diagrama]

Target

El directorio target en un proyecto Maven es donde se colocan todos los archivos de salida cuando se construye el proyecto. Cuando compilas tu proyecto con mvn compile, el código compilado (archivos .class) se colocará en target/classes. Cuando empaquetas tu proyecto con mvn package, el paquete resultante (por ejemplo, un archivo JAR, WAR o EAR) se colocará en el directorio target.

Archivos JAR, WAR y EAR:

- **JAR (Java ARchive):** Es un archivo ejecutable de java, como un .exe de Windows. Si compartes el jar con alguien solo necesitará el JRE (que incluye la JVM)para poder hacerle doble click y ejecutar el programa.
- **WAR (Web Application aRchive) y EAR (Enterprise ARchive):** Se utiliza para distribuir aplicaciones web Java y subirlas a un servidor que se encarga de desplegar la aplicación en la web, haciéndola accesible a través de Internet.

Por defecto, Maven genera un archivo JAR cuando empaquetas una aplicación. Puedes cambiar este comportamiento mediante la configuración del elemento <packaging> en tu archivo pom.xml. Por ejemplo, para crear un archivo WAR, debes modificar el archivo pom.xml de esta manera:

```
<project>
  ...
    <groupId>com.mycompany</groupId>
    <artifactId>app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
  ...
</project>
```

De esta forma, cuando ejecutes *mvn package*, Maven generará un archivo WAR en lugar de un archivo JAR. Para un archivo EAR, simplemente reemplaza "war" con "ear" en el elemento <packaging>.

Arquetipos

Un arquetipo en Maven es un **patrón o modelo de proyecto**, que proporciona una estructura de directorios y un conjunto de archivos de proyecto predeterminados. Los arquetipos son una forma de reutilizar un estilo de proyecto común y reducir la repetición al iniciar un nuevo proyecto.

Por ejemplo, si a menudo comienzas nuevos proyectos con una configuración específica, puedes tener un arquetipo que incluye esa configuración en el pom.xml inicial con las dependencias comunes de tus proyectos, y tal vez algunos archivos de código fuente o recursos predeterminados.

Maven viene con una serie de arquetipos predefinidos que puedes usar para iniciar nuevos proyectos. Por ejemplo, el arquetipo maven-archetype-quickstart es un arquetipo simple que crea una estructura de proyecto con una clase principal y una prueba de unidad.

Para crear un nuevo proyecto a partir de un arquetipo, puedes usar el plugin archetype:generate de Maven. Aquí hay un ejemplo de cómo podrías hacer esto en la línea de comandos:

```
mvn archetype:generate \
  -DgroupId=com.mycompany.app \
  -DartifactId=my-app \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DinteractiveMode=false
```

Esto crea un nuevo proyecto en un directorio llamado "my-app" con la estructura de directorios y los archivos predeterminados proporcionados por el arquetipo maven-archetype-quickstart.

Aquí están los significados de las opciones:

- **DgroupId:** El ID del grupo para tu proyecto. Esto generalmente se basa en el nombre del dominio de tu organización al revés.
- **DartifactId:** El ID del artefacto para tu proyecto. Este es el nombre del proyecto.
- **DarchetypeArtifactId:** El ID del artefacto del arquetipo que deseas usar para crear tu proyecto. En este caso, estamos utilizando el arquetipo maven-archetype-quickstart.
- **DinteractiveMode:** Si esto se establece en *false*, Maven no hará ninguna pregunta y creará el proyecto con los valores predeterminados. Si esto se omite o se establece en true, Maven hará una serie de preguntas para personalizar el nuevo proyecto.

Si quieres crear tu propio arquetipo sigue los siguientes pasos:

Crea un proyecto Maven estándar:

Este proyecto actuará como la plantilla para tu arquetipo. Puedes establecer la estructura de directorios que prefieras, agregar cualquier

archivo predeterminado y configurar el pom.xml de acuerdo a tus necesidades. Este proyecto debería representar la estructura base y los archivos que deseas para cada nuevo proyecto que crees con este arquetipo.

Crea un descriptor de arquetipo:

Este es un archivo XML que le dice a Maven qué archivos incluir en el arquetipo. Normalmente, lo colocarás en el directorio src/main/resources/META-INF/maven.

Aquí tienes un ejemplo de cómo podría verse este descriptor (archetype-metadata.xml):

```
<archetype-descriptor name="myArchetype">
  <fileSets>
    <fileSet filtered="true" packaged="true">
      <directory>src/main/java</directory>
      <includes>
        <include>**/*.java</include>
      </includes>
    </fileSet>
    <fileSet filtered="true">
      <directory>src/main/resources</directory>
    </fileSet>
  </fileSets>
</archetype-descriptor>
```

En este ejemplo, se incluyen todos los archivos .java en src/main/java y todos los archivos en src/main/resources. El atributo filtered indica si Maven debe reemplazar los marcadores de posición en estos archivos. El atributo packaged indica si los archivos deben colocarse en directorios que correspondan a su paquete (es relevante para los archivos .java).

Agrega el plugin de arquetipo al pom.xml:

Deberás configurar el plugin de arquetipo en tu pom.xml para crear el arquetipo. Aquí tienes un ejemplo:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-archetype-plugin</artifactId>
      <version>3.2.0</version>
    </plugin>
  </plugins>
</build>
```

Genera y despliega tu arquetipo:

Una vez que tienes tu proyecto de arquetipo configurado, puedes crear el arquetipo con mvn archetype:create-from-project. Esto creará un proyecto de arquetipo en el directorio target/generated-sources/archetype.

Entra al directorio target/generated-sources/archetype

y en la consola ejecuta *mvn install* para instalar el arquetipo en tu repositorio local de Maven, o mvn deploy para desplegarlo en un repositorio remoto.

Usa tu arquetipo:

Ahora puedes usar tu arquetipo para crear nuevos proyectos con mvn archetype:generate, especificando el groupId, artifactId y versión de tu arquetipo.

Aquí tienes un ejemplo de cómo podría verse un pom.xml básico para un proyecto de arquetipo:

```
<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>com.mycompany</groupId>
    <artifactId>my-archetype</artifactId>
    <version>1.0.0</version>
```

```
    <packaging>maven-archetype</packaging>
    <name>My Custom Archetype</name>
    <description>This is my custom archetype for new projects</description>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>20</java.version>
        <maven.compiler.source>${java.version}</maven.compiler.source>
        <maven.compiler.target>${java.version}</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.9.3</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-archetype-plugin</artifactId>
                <version>3.2.0</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.1.2</version>
            </plugin>
        </plugins>
    </build>
</project>
```