**Maven** – фреймворк для управления проектом и автоматизации его сборки.

**Особенности Maven:**

1. **Стандартизация структуры проекта**
2. **Декларативное управление зависимостями**
3. **Управление конфигурацией проекта**
4. **Управление жизненным циклом проекта**

**Стандартизация структуры проекта** – Maven предоставляет стандартную конфигурацию проекта, основываясь на *соглашении о конфигурации* – предоставляет некоторые умолчания, снижая тем самым кол-во опций (настроек) - определяет структуры папок (исходники (src\main\java), тесты (src\test\java), ресурсы (src/test/resources), скомпилированные классы (target), наличие POM-файла), плагины по умолчанию и др.

«Умолчания» реализованы в виде файла super-POM, который определяет глобальные настройки и является родителем для других POM-файлов.

Наличие соглашения о кодировании снижает кол-во решений, которые необходимо принимать для конфигурации проекта и сокращение объема требуемого конфигурирования. Настройки берутся из файла super pom – в нем содержатся стандартные опции конфигурации.

Что дает: уменьшение сроков онбординга за счет наличия стандартизированных структур, независимость от используемой IDE (различные IDE используют разные структуры папок).

Для стандартизации структуры используется **архетип** – готовый шаблон проекта с предопределенной структурой, настройками и зависимостями. Реализованы в виде maven archetype plugin. mvn archetype:generate – вспомогательный плагин.

**Декларативное управление зависимостями** – объявление зависимостей в POM-файле <dependencies> и их последующий поиск и загрузка:

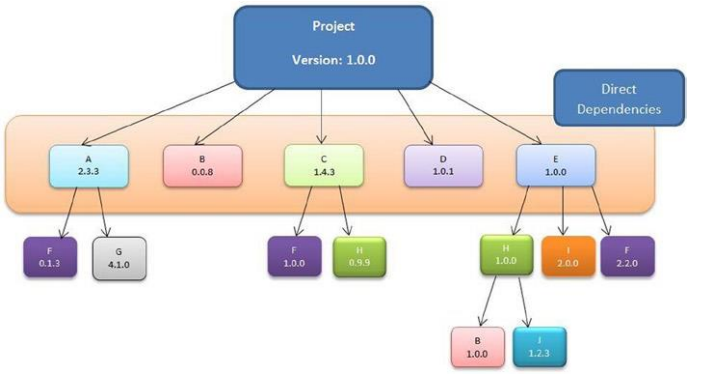
1. поиск в локальном Maven репозитории (папка .m2\repository)
2. поиск в центральном Maven репозитории
3. поиск в удаленном репозитории (если определен)

Плагин dependency определяет goal для работы с зависимостями проекта, позволяет контролировать зависимости и получать структурированную информацию о них ( :list – получить список всех зависимостей, :analyze – анализирует зависимости проекта на конфликты версий, используются ли они и тд, :tree – получить иерархию зависимостей, :help – подробно описаны 21 goal этого плагина)

Обработка транзитивных зависимостей – автоматическая загрузка транзитивных зависимостей, которые требуются для работы прямых зависимостей. Наличие транзитивных зависимостей определяется наличием POM файла у прямой зависимости, после чего строится дерево зависимостей.

Подход “посредничества зависимостей” - это процесс разрешения конфликтов версий зависимостей, которые могут возникать при использовании различных библиотек в проекте. При возникновении такого конфликта, будет включена та зависимость, которая будет ближайшей к проекту (корню дерева зависимостей) в дереве зависимостей.

Пример: Будет загружен артефакт B 0.0.8, а не B 1.0.0, F 0.1.3, а не F 1.0.0 и не F 2.2.0



Исключать транзитивные зависимости из определенных артефактов можно с помощью <exclusions> в том случае, если несколько зависимостей конфликтуют, а ближайшая зависимость имеет не самую актуальную версию (эту проблему также можно решить путем определения версий зависимостей в родительском pom с помощью <dependencyManager>) или необходимо простое исключение транзитивной зависимости. Например, библиотека Junit4 содержит транзитивную зависимость harmcrest, ее исключение будет выглядеть следующим образом:

<dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.13.2</version>  
 <scope>test</scope>  
 <exclusions>  
 <exclusion>  
 <groupId>org.hamcrest</groupId>  
 <artifactId>hamcrest-core</artifactId>  
 </exclusion></exclusions>  
</dependency>

Что дает: решает проблему хранения списка подключенных зависимостей и их транзитивных зависимостей, организацию централизованного хранилища для этих библиотек и их версий.

**Управление конфигурацией проекта** – определение и настройка аспектов проекта: зависимости, плагины, репозитории, профили и т.д. Определяются за счет файла POM.xml (project object model), который содержит информацию о различных деталях конфигурации проекта.

**Структура POM файла**

Все элементы POM заключены в корневой тэг <project>, в шапке располагается информация о версии модели POM и информация о XSD схеме (XML Schema Definition), которая описывает структуру и типы данных допустимые для каждого элемента в файле POM. В результате XML-редакторы могут использовать эту схему для автоматической проверки синтаксиса и валидации POM файлов.

<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>

Также обязательно указывается уникальный идентификатор проекта (gav-тэг), который состоит из 3х тэгов:

<groupId> – обычно это название компании (название группы или организации, которой принадлежит проект).

<artifactId> – название проекта в рамках группы.

<version> – версия проекта в формате major.minor.increment – qualifier (1.2.10-SNAPSHOT).

<packaging> – необязательный тэг для идентификации проекта, устанавливает формат, в который будет упакован проект на стадии package (если не указать, то по умолчанию .jar)

<properties> - определение пользовательских переменных, которые используются для определения параметров проекта, которые могут быть использованы в других частях проекта (для упрощения конфигурации проекта).

${имя-переменной} – используется для обращения к указанной переменной из блока <properties> и получения ее значения.

Используется для централизации конфигурации проекта (все собрано в одном месте), переиспользование ранее определенных свойств. Использование переменных, определенных ранее позволяет избежать ошибок при прямом введении значений в тэге. Вместо этого значения определяются один раз и затем используется многократно в различных частях проекта.

<repositories> - тэг для определения удаленных репозиториев, удаленный репозиторий также может быть настроен в settings.xml. settings.xml определяет конфигурации локально, то есть на определенной машине, POM – на уровне проекта.

**Остальные группы тэгов:**

1) *General project information* – включает метаинформацию о проекте для документации (не используется в жизненном цикле проекта) - <name>, <url>, <contributions>, <licenses>, <developers> и др.

2) *Environment* – настройка процесса сборки проекта, в зависимости от его окружения <profile>

<profiles> - позволяет настраивать профили сборки, которые определяют конфигурацию POM файла (зависимости, плагины и др.) в зависимости от сценария активации. Активация профиля:

через консоль (флаг –P)   
средствами Maven   
через тэг <activation> (наличие или отсутствие определенного файла, версия JDK, операционная система, переменной окружения и тд)

3) Build settings – настройка жизненных циклов - определение подключаемых плагинов, директорий, ресурсов и др для определенного жизненного цикла. Настраивать можно 2 жизненных цикла – цикл default (build) <build> и цикл site <reporting>, цикл clean не настраивается.

4) Relationships – включает информации о зависимостях проекта <dependencies>, информацию о наследовании POM-файла <modules>, <parent>.

<dependencies> - зависимости подключаются путем указания их gav-тэга, также может добавляться тэг <scope>, для указания области видимости этой зависимости.

- compile – значение по умолчанию, зависимость будет видима на всех этапах (компиляция, рантайм, тесты).

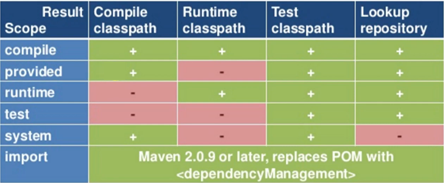
- provided – доступна только во время компиляции и тестирования, не будет доступна в рантайме (будет предоставлена кем-то другим в рантайме). Обычно это используется для библиотек, которые предоставляются контейнером приложения.

- runtime – не будет видна во время компиляции, но будет доступна во время выполнения (JDBC-драйверы).

- test – используется в целях тестирования, не попадет в скомпилированный код и не будет использоваться в рантайме.

- system – deprecated, работает как provided, но зависимость будет искаться не в репозитории, а по указанному пути.

- import – определяется в дочернем модуле, указывает на то, что будет использована <version> и <scope>, которые были определены в родительском модуле, в тэге <dependencyManagment>



Тэги наследования:

Используются при организации многомодульной структуры проекта - состоит из родительского проекта и дочерних, где каждый дочерний модуль представляет собой Maven-проект, который может быть собран независимо, но при этом все они связаны общим родительским POM-файлом.

<modules> - тэг в родительском pom.xml, включает информацию о дочерних модулях проекта, <parent> - тэг в дочернем модуле, который включает информацию о родительском модуле.

Родительский модуль представлен типом <packaging>pom</packaging>, что указывает на то, что данный проект не производит артефакт типа jar, war, а служит для управления дочерними модулями, то есть содержит их общую базовую конфигурацию (зависимости, плагины и т.д.), а каждый модуль представляет отдельный проект, который может воспользоваться конфигурацией по умолчанию или определить собственное поведение.

В агрегаторе может быть использовано 2 способа описания зависимостей:

<dependencies> - зависимости будут включены в каждый дочерний модуль (по правилу наследования - будет наследоваться всеми дочерними модулями этого POM-файла, если они не переопределяют эту зависимость.), <dependencyManagement> - описанные в этом блоке зависимости не будут фактически включены в модули проекта без явного их указания в блоке <dependencies> каждого модуля. При этом секция <dependencyManagement> устанавливает рекомендацию для <version> и <scope>, которую нужно использовать для подключаемой зависимости. Тот же механизм действует для <plugins> и <pluginManagment>.

Тэг <dependencyManagment> обычно выносится в специальный BOM.xml для централизованного управления версиями зависимостей.

*Преимущества модульной структуры в Maven:*

1) Легкость управления зависимостями и плагинами: Каждый модуль может иметь свои собственные зависимости, что делает управление зависимостями более гибким и позволяет избежать дублирования зависимостей в проекте.

2) Упрощение сборки проекта: Модульная структура позволяет собирать только те модули, которые изменились, что ускоряет процесс сборки проекта.

3) Логическая организация проекта: Разбиение проекта на модули по функциональным или логическим границам делает его структуру более понятной и облегчает сопровождение и разработку.

Super POM – базовый POM, который наследуется всеми остальными POM-файлами. Он определяет стандартные настройки и конфигурации, которые применяются к проекту, если эти настройки не переопределены в наследующих POM-файлах, что позволяет реализовать соглашение о конфигурации, что сокращает объем требуемого конфигурирования.

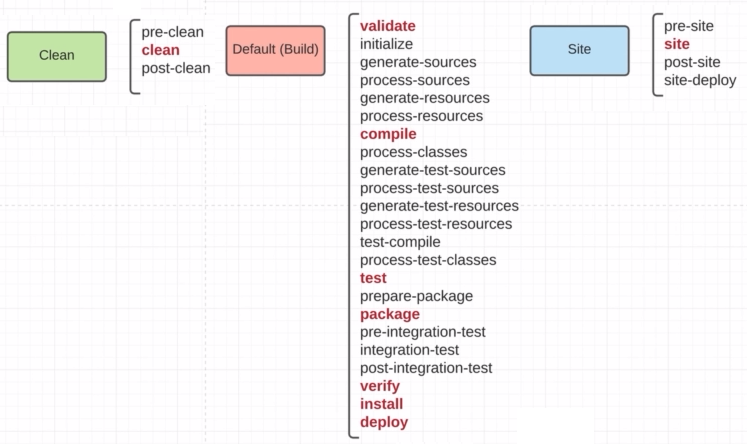
Результирующий POM проекта можно получить с помощью плагина help – mvn help:effective-pom , этот результирующий POM будет содержать дефолтные плагины (их нет в super pom) + super pom + pom.

**Управление жизненным циклом проекта** –осуществляется за счет возможности определения функционала в жизненном цикле проекта, жизненный цикл может расширяться за счет определения в нем плагинов, отличных от стандартных.

Maven состоит из стандартного набора плагинов, предоставляемых по умолчанию, функционал плагина выражен конкретным действием или задачей, которую исполняет плагин – goal.

mvn plugin:goal (plugin – имя плагина, goal – его цель). mvn plugin:help – вызовет справочную информацию об этом плагине и о его goal.

**Жизненный цикл** – последовательность фаз, через которые может быть пропущен проект, каждая фаза вызывает определенные в ней goal (может быть несколько) плагина. Некоторые фазы не имеют определенного goal, что предоставляет возможность подключения собственного плагина и привязке его goal к фазе жизненного цикла. Особенностью является то, что при запуске какой-либо основной фазы (выделены красным), исполняются все предшествующие фазы. Goal не привязанные ни к одной фазе жизненного цикла могут быть вызваны вне жизненного цикла с помощью прямого вызова (с помощью mvn plugin:goal) т.е. goal привязанные к ЖЦ при обращении к ним с помощью прямого вызова все равно вызываются в жизненном цикле (mvn compile:compile все равно вызовет все предыдущие фазы, т.к. привязана к циклу build).



*3 вида ЖЦ:*

1) Цикл очистки – имеет три фазы, по умолчанию определен только один - *clean* - удаляет временные файлы и артефакты предыдущей сборки - скомпилированные файлы (директория target), war, jar файлы.

2) Цикл сборки (deafult) – состоит из 23 фаз, основные:

*validate* - проверка POM-файла на корректность и проверка доступности файлов и каталогов.

Фазы initialize, generate-sources, process-sources, generate-resources не имеют goal по умолчанию

*Process-resources* – копирует ресурсы в папку target.

*compile* – компилирует исходный код (помещает в target), кеширует промежуточные результаты сборок. При сборке большого проекта Maven может пересобрать только ту его часть, которая изменилась, ускорив, таким образом время сборки.

Фазы process-classes, generate-test-sources, process-test-sources, generate-test-resources не определены по умолчанию.

*process-test-resources* – копирует ресурсы из папки тест в target.test-classes

*test-compile* – компилирует тесты и добавляет их в папку target.test-classes

*test* - запускает все тесты в директории test. При выполнении следующих этапов этот можно пропустить с помощью mvn исполняемая фаза –DskipTests или mvn фаза -Dmaven.test.skip=true тоже самое можно прописать в POM файле

<properties>  
 <maven.test.skip>true</maven.test.skip>  
</properties>

Или если переопределен плагин установить переменную в нем:

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.11.0</version>  
 <configuration>  
 <skip>true</skip>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

Явно установил нужную версию плагина и установил флаг maven.test.skip Таким же способом можно настраивать переменные либо глобально через <properties> , либо посредством установки непосредственно у плагина, например, установить какие классы будут запускаться для тестов можно с помощью

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-surefire-plugin</artifactId>  
 <version>3.2.2</version>  
 <configuration>  
 <includes>\*\*/\*.java</includes>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

\*\*/\*.java устанавливает шаблон поиска тестов на файлы с любым названием.

Фаза prepare-package не связана с каким-либо goal

*package* - упаковка проекта в установленный артефакт (war, jar и др.) Будет содержать только исходный код без зависимостей.

При запуске этой фазы этапы жизненного цикла будут зависеть от установленного формата упаковки в <packaging> [дока](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html)

Фазы pre-integration-test, integration-test, post-integration-test, verify не связаны с каким-либо goal.

*verify* – связана с запуском и проверкой результатов интеграционных тестов. (failsafe-plugin плагин для интеграционных тестов) интеграционные тесты определяются также как и Unit – посредством определенных префиксов или суффиксов в названии класса (IT\* или \*IT или \*ITCase по умолчанию)

install – добавляет пакет из фазы package в локальный репозиторий (.m2), что позволяет другим проектам, использующим этот репозиторий использовать этот пакет.

deploy - загрузка артефактов в удаленный репозиторий, для настройки удаленного репозитория нужно использовать settings.xml , который располагается либо в корневой папке Maven, либо в .m2

3) site – имеет 4 фазы (pre-site, site, post-site, site-deploy), goal определен только для site – отвечает за создание документации для проекта, отчетов для тестов и тд для конфигурации используется <reporting> в pom.xml. Для формирования документации для тестов используется Jacoco плагин.

**Структура jar и war**

Jar – включает директорию META-INF, которая содержит файл MANIFEST.MF и директорию с класс-файлами, может содержать папку resources.

War – zip архив, содержит статические веб ресурсы который содержит в себе веб-приложение META-INF (classes, lib – зависимости, web.xml – дескриптор развертывания), WEB-INF – хранит конфигурационные файлы и ресурсы (недоступны напрямую клиенту – нельзя запросить напрямую через URL), ресурсы веб приложения (css, js, images, jsp и др.)

**MANIFEST.MF** – файл метаданных Jar-архива, содержит информацию о его конфигурации. При создании jar-архива с помощью package Maven создает Manifest по умолчанию в папке META-INF, создание этого файла вручную приведет к его замене (Maven затрет сконфигурированный манифест), что требует его конфигурации средствами Maven. Для этого необходимо настроить maven-jar-plugin.

Для возможности запуска jar-архива необходимо указать путь к Main в MANIFEST.MF

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-jar-plugin</artifactId>  
 <version>3.3.0</version>  
 <configuration>  
 <archive>  
 <manifest>

<mainClass>org.example.App1</mainClass>  
 </manifest>  
 </archive>  
 </configuration>  
</plugin>

Если запускаемый jar-файл использует зависимости, то потребуется определить в MANIFEST.MF путь, по которому эти зависимости будут искаться:

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-jar-plugin</artifactId>  
 <version>3.3.0</version>  
 <configuration>  
 <archive>  
 <manifest>  
 <addClasspath>true</addClasspath>  
 <classpathPrefix>lib</classpathPrefix>  
 </manifest>  
 </archive>  
 </configuration>  
</plugin>

Так же необходимо обеспечить наличие самих зависимостей, что можно сделать двумя способами:

1) Определить в жизненном цикле создание директории и копирование в нее зависимостей с помощью maven-dependency-plugin (привязать плагин к ЖЦ и определить папку, в которую будут скопированы зависимости)

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-dependency-plugin</artifactId>  
 <version>3.6.1</version>  
 <executions>  
 <execution>  
 <id>create-jar</id>  
 <goals>  
 <goal>copy-dependencies</goal>  
 </goals>  
 <phase>prepare-package</phase>  
 <configuration>  
 <outputDirectory>${project.build.directory}\lib</outputDirectory>  
 </configuration>  
 </execution>  
 </executions>  
</plugin>

2) Собрать общий jar-файл, который уже будет содержать все необходимые для него зависимости с помощью maven-assembly-plugin (привязать к ЖЦ и указать путь к main и дескриптор)

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-assembly-plugin</artifactId>  
 <version>3.6.0</version>  
 <executions>  
 <execution>  
 <id>some</id>  
 <goals>  
 <goal>single</goal>  
 </goals>  
 <phase>package</phase>  
 <configuration>  
 <descriptorRefs>jar-with-dependencies</descriptorRefs>  
 <archive>  
 <manifest>  
 <mainClass>org.example.App1</mainClass>  
 </manifest>  
 </archive>  
 </configuration>  
 </execution>  
 </executions>  
</plugin>

Maven-wrapper – скрипт оболочка, которая будет использовать ту версию Maven, которая определена в проекте.

Для написания собственного плагина необходимы зависимости: maven-plugin-api, maven-plugin-annotation.