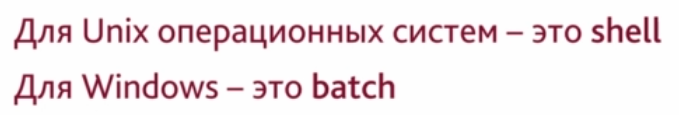
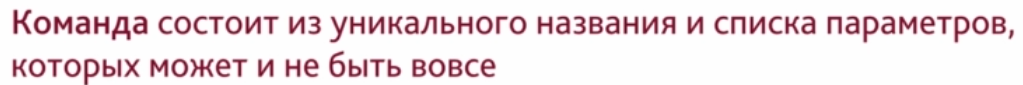


Команды в PowerShell называются командлеты.









Пример:



“~” – в командах обозначает директорию текущего пользователя

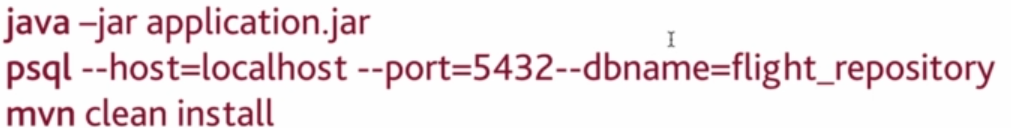
cat first.json – отобразить содержимое файла first.json на консоли

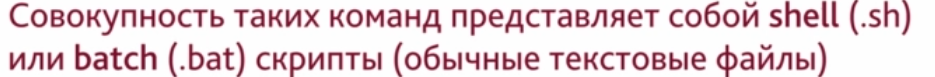
 - скопировать из директории текущего пользователя папки Download файл apache-maven.zip в директорию текущего пользователя в папку maven.

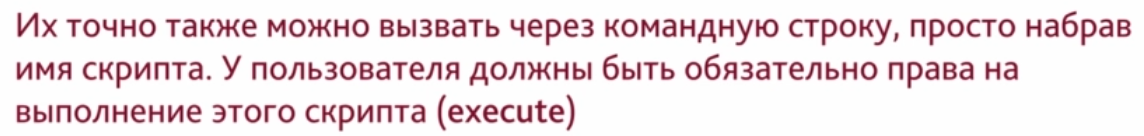
 - меняем текущую директорию на Work/test-project

Это стандартные команды, которые отличаются от того это shell или batch.

Но есть команды и для установленных приложений

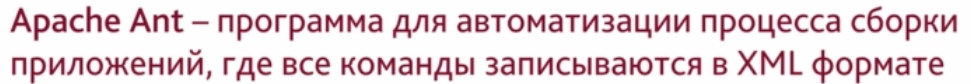


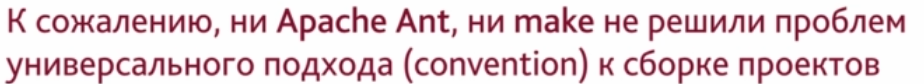


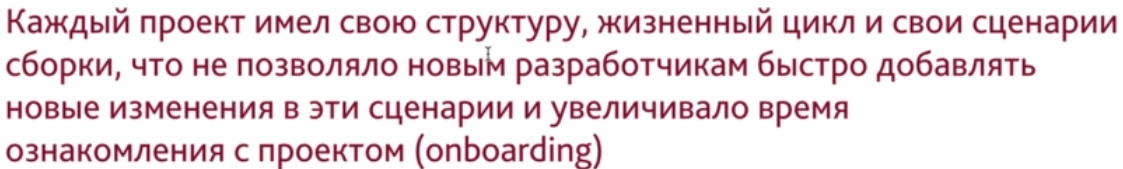


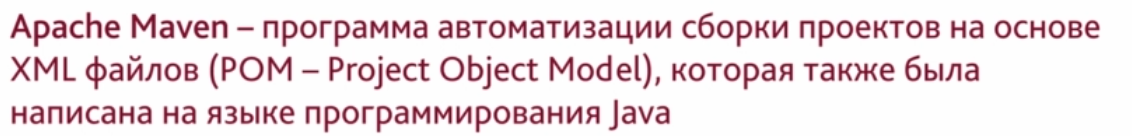
Обычно, чтобы выполнить какой-либо скрипт вызвают файл из командной строки, в котором указаны несколько таких команд.

Shell скрипты используются только для linux-систем, batch – для windows.

















Maven – совокупность плагинов, плагин – основная часть как ядро(core) и при общении с maven мы общаемся с этими плагинами.

core – ядро maven

goal – цели

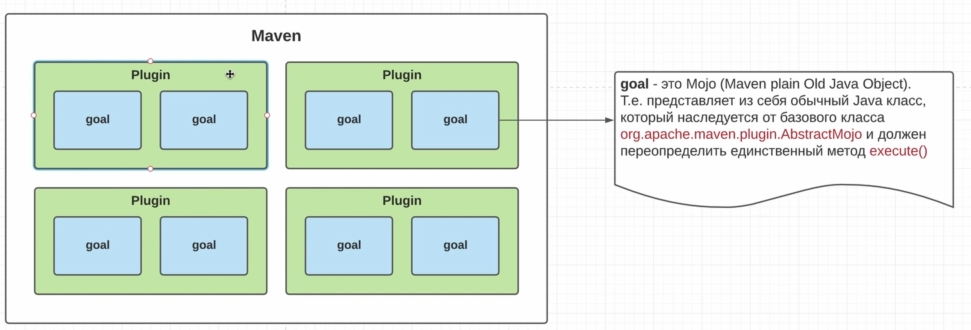
Плагин – java-проект состоящий из набора классов, и главные из этих классов это goal-ы, от 1 до нескольких. При этом каждый плагин содержит обязательно одну цель это help(служит для описания плагина)

При работе с плагинами вызывается тот или иной goal или их совокупность.

В core содержаться плагины которые всегда используются в java-приложениях:

* Clean
* Compiler
* Deploy
* Install
* Resources
* Site
* Verifier
* Surefire

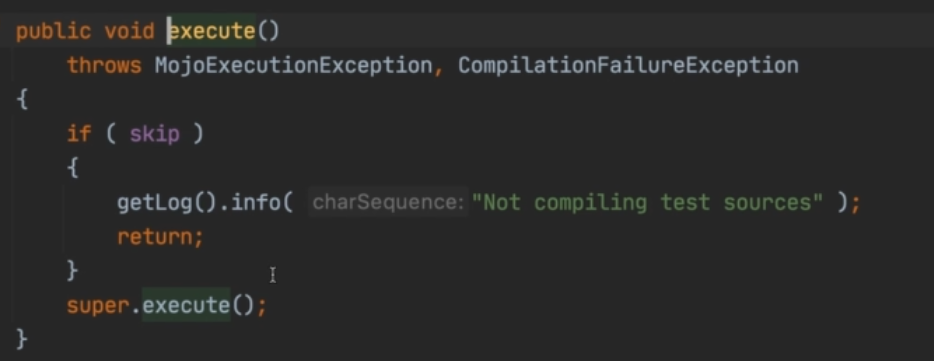
Кроме того, goal-ы в maven называются MOJO



Mojo java класс который наследуется от AbstractMojo, который в свою очередь реализует интерфейс Mojo c методами execute(), setLog() и getLog()



Пример переопределения метода execute()



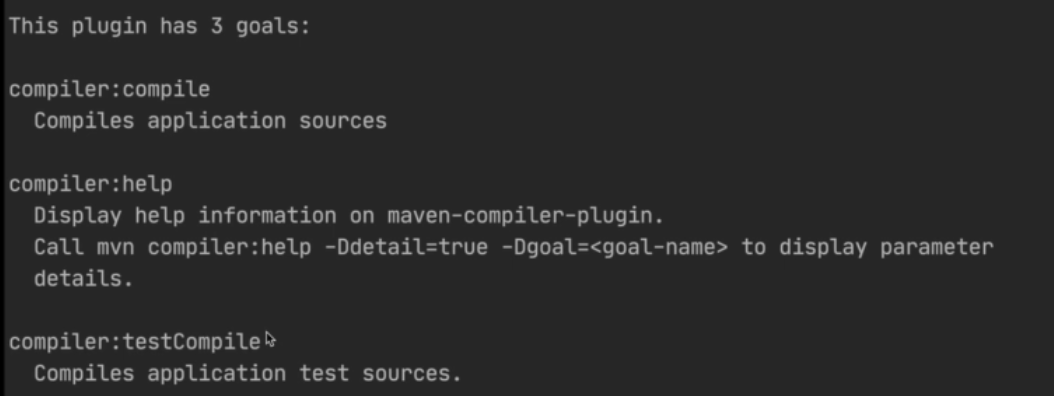
А реализация метода execute() это и есть то, что выполняет goal в плагине.

Чтобы набрать команду для maven пишем

mvn – основная команда, далее идет набор параметров

Чтобы обратиться к плагину нужно набрать его имя compiler и далее после двоеточия набираем цель(goal), которая нас интересует

**mvn compiler: help** - таким образом вызывается необходимый goal в нужном плагине



Как видно из рисунка выше, во время запуска нашего приложения мы можем дополнительно передавать какие-то параметры через -D, это характерно для любых java приложений. Такие параметры называются ещё JVM-аргументы. Есть также и обычные аргументы которые принимаются в виде массива строк в нашем методе **main(String[] args)**

Создадим **java-**приложение выводящее на консоль Hello world.

Зайдем в конфигурацию ранера Edit configuration и здесь видно, что есть два способа передать наши аргументы

* VM options – это с -D
* Program arguments – то, что мы принимаем в качестве массива строк в методе main

И то и другое мы можем передавать через командную строку, (на примере плагина compiler выше)

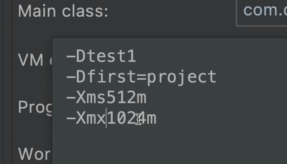
Добавим JVM options – кликаем в строку – в развернувшемся окне пишем

(этот вид аргументов делится на 3 типа:

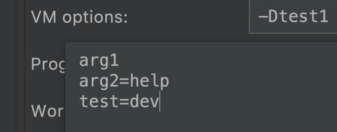
1. -D Format: -Dkey=value это user jvm args и могут принимать любые значения, это то что мы сами придумаем. Разделяются они друг от друга пробелом или переносятся на новую строку
2. Зарезервированные jvm options, начинаются с -Х, мы их не можем придумывать сами, они есть уже существующие и их можно увидеть в документации. Так например, можно задать размер нашего heap и максимальный его размер, который он может принимать во время работы

-Xms512m -Xmx1024m

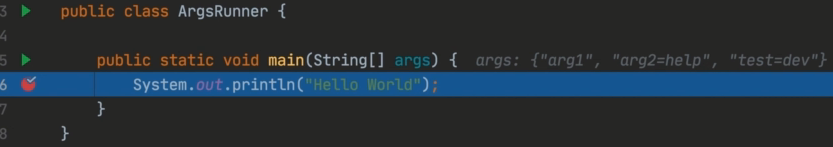
1. -XX для конфигурации компилятора или garbage коллектор, метаспэйсов и пр. и они также зарезервированы и мы не придумываем их сами.



Programm arguments – это аргументы, которые мы можем принимать во время запуска java-приложения в качестве массива строк в методе main (они также могут принимать любые значения, также перечисляются через пробел и именуются как угодно так как не зарезервированы)

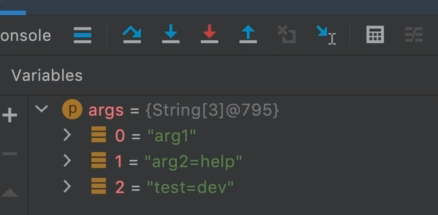


Запускаем приложение в режиме дебаг и попытаемся найти заданные аргумены.

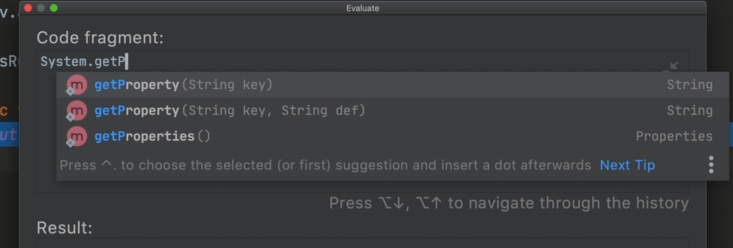


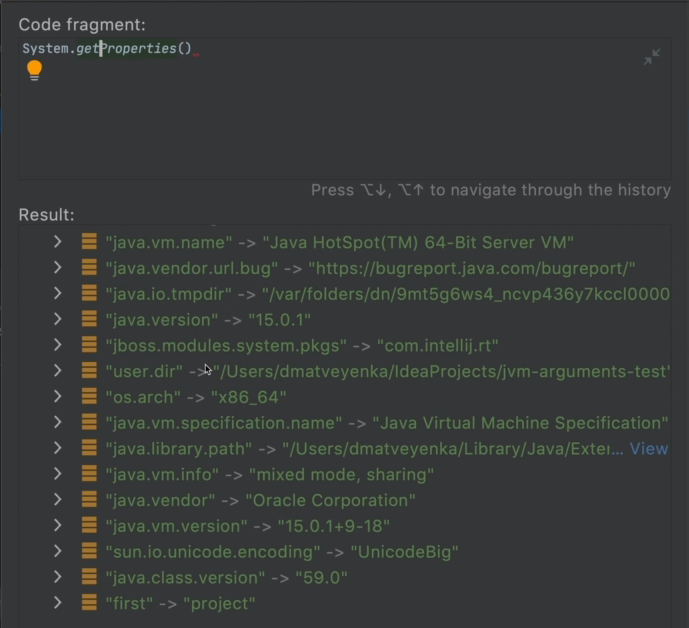
Важное замечание: аргументы args[] метода main не имеют формата в отличие от пользовательских JVM –D

Также знак «=» является пользовательским сепараторам аргумента, он может быть любой а может и не быть его вообще как в первом аргументе.



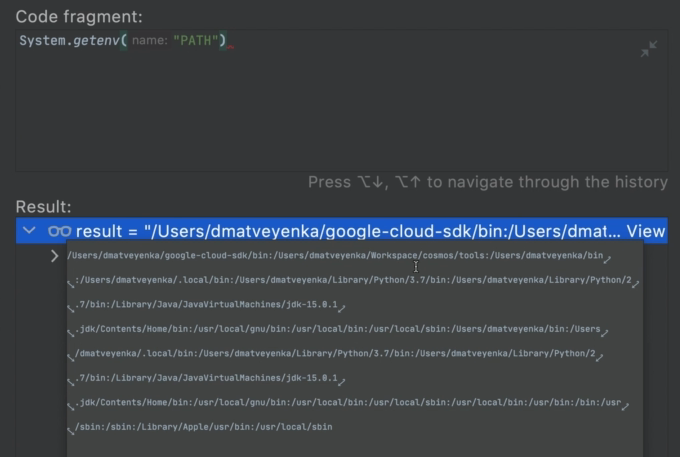
Нас интересуют jvm options. Чтобы их получить есть спец класс System (alt + F8) и по ключу можно получить эти аргументы или все сразу.





Аргументы которые заданы через -Х -ХХ мы получить не можем, потому как мы их использовать не должны, так как они используются нашей JVM

Также через System можно получить environment переменные, то что мы задаем в нашем компьютере(нашей операционной системе). Так мы например можем получить переменную PATH:



Задание таким образом аргументов используется, чтобы настраивать процесс запуска и конфигурации нашего проекта.

Ещё раз:

Чтобы передавать пользовательские значения настраиваются VM Options со значком -Dключ=значение,

Чтобы передавать системные значения, они именуются -Xзначение, -XXзначение, но получить на стороне приложения мы эти значения не можем, потому что они нами не могут использоваться.

Также можно передавать Program arguments во время запуска приложения

А через класс System можем получать JVM options пользовательские или environment

Схема вызова любого goal-a любого плагина maven:

**mvn имя\_плагина: goal**

например: mvn archetype:help

Создание структуры проекта с помощью maven.

Используем maven-archetype-plugin

Находится в папке tools, в папке archetype в ней плагин

Перейдя в командную строку и введя mvn archetype:help

Мы видим, что данный плагин содержит 7 голов. Гол, который нам нужен – generate.

Этот плагин позволяет нам создавать новые проекты и создает необходимую maven-структуру для проекта(набор папок и их цепочки).

Воспользуемся голом generate для создания самого простого java-проекта.

Для этого будем использовать архетип quickstart

Создаем директорию

Открываем командную строку и пишем

mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.5

Таким образом мы задаем с помощью передаваемых параметров (пользовательские JVM options-ы) нужный archetype

После ввода строки запрашивает система groupId

Что такое groupId?

groupId (также мы использовали его, чтобы указать какой архетип я хочу использовать) в мавен есть спец конвенция по именованию проектов, плагинов и пр.

Название проекта должно быть уникально в рамках всех проектов, которые создавались в maven, а не только нашего локального компьютера.

**Уникальное имя проекта состоит из трёх значений:**

**groupId : artifactId : version**

groupId – имя компании наоборот, составляющие разделяются точкой com.pavlov

artifactId – название проекта в рамках одной группы, составляющие разделяются точкой или тире как хотите(не так строго как groupId)

**version – состоит из трёх состовляющих:**

**major : minor : increment – qualifier**

major – главная отметка

minor – релиз произошёл, версию поменяли

increment – после баг-фиксов

qualifier чаще всего это SNAPSHOT (означает, что это не продакшен, а разработка)

т.о. version используется для релизов проекта

Далее система предлагает назвать пакет внутри приложения как и groupId или переименовать его.

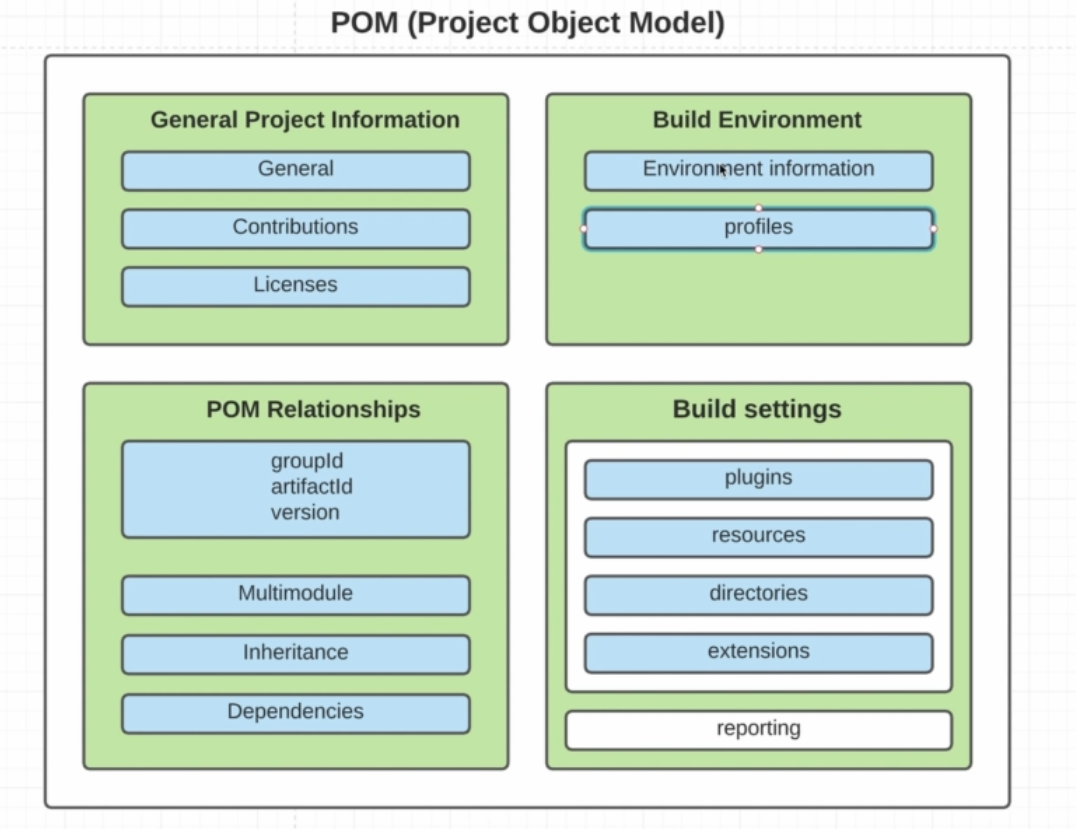
ОК – Создание окончено.

Теперь с помощью идеи можно открыть сформированный проект в данной папке.

Основной конфигурационный файл – pom.xml

(Project Object Model)

Делится на 4 основные части:



1. GPI – собрана общая информация, которая не влияет на сборку проекта.

Здесь название проекта, урлики на сайты нашего проекта, контрибьюторы(программисты) которые разрабатывали проект, лицензии и прочее.

Служит для документации в последующем.

1. BE – здесь происходит конфигурация различных environment-ов, те наше приложение может работать как на локальном environment-e так и на qa, где тестируют его, так и на prodaction-e и соответственно может отличаться процесс сборки нашего проекта. Для этого указывают различные профайлы(profiles) environment information и пр.
2. POM Relationship – pom coordinate – уникальные идентификаторы проекта.

**groupId : artifactId : version**

Здесь же конфигурируется многомодульность – multimodule

Есть также наследование – inheritance

Одна из самых основных - это зависимости – dependencies.

1. Build settings – жизненный цикл проекта

Их 3:

- дефолтный (ещё называют build). В нем можно указывать плагины, которые используются, различные ресурсы(папка resources) как для исходников так и для тестовых классов, различные директории, extention-ы и пр.

- второй жизненный цикл – это reporting – генерация отчетов по проекту

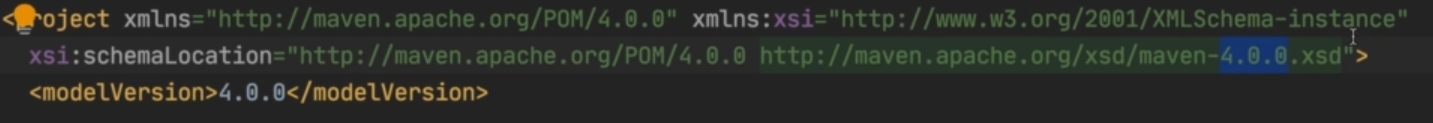
- третий - не настраивается

Рассмотрим наш pom.xml:

Каждый из xml состоит из стартовой строки и рутового тэга – project

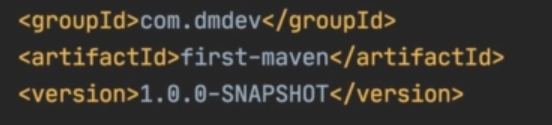
Все остальные теги располагаются внутри тэга project

Значение modelVersion должно соответствовать значению xsd схемы

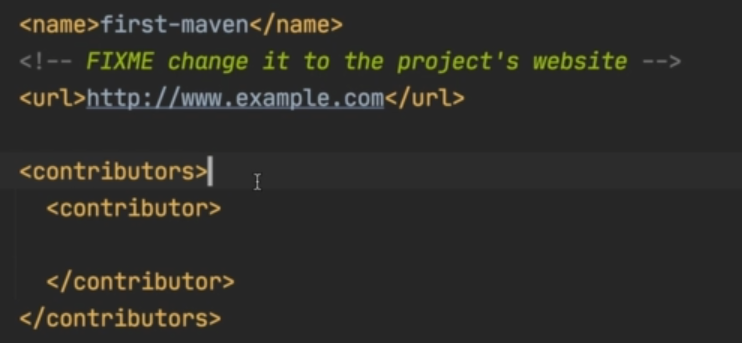


Так как используется xsd схема, то нельзя внутри project указать невалидный(произвольный) тег. Навести на схему ctrl+click заходим в схему и видим какие тэги можно использовать.

Далее pom coordinate

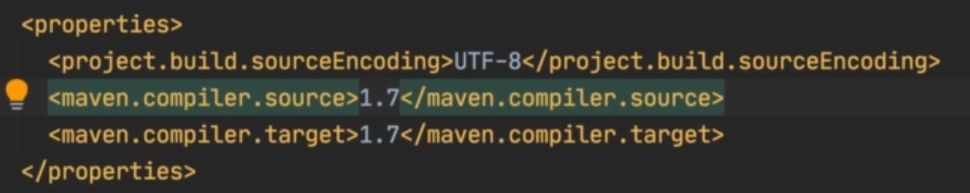


Далее метаинформация



Где можно и разработчиков указать

Далее properties – это нечто вроде глобальных переменный в java, мы можем ссылаться на них из всех наших pom-ов



Далее dependencies – Это библиотеки от которых мы зависим

Далее секция build (4-я группа) – относится к дефолтному life cycle, в нём

тэг directory – указывает где будет происходить результат выполнения сборки проекта, по умолчанию она называется target.

Также как и в java в pom есть наследование т.е. базовый класс от которого наследуются все другие pom-ки. Этот базовый класс называется superPom.

В нем есть также конфигурация, которая передается производным pom-кам по наследству.

В нем указывается дефолтный репозиторий откуда все maven проекты будут скачивать все свои зависимости. Также здесь есть pluginRepository, который указывает откуда все плагины будут скачиваться.

Эта директория находится в папке lib директории, куда мы его установили.

В секции build есть множество заполненных тэгов, это и directory (target) и outputDirectory (classes) и др. таким образом достигается однообразие структуры проекта. Конечно их можно переопределить, но не желательно.

Здесь же есть дефолтные плагины. Они предустановлены, поэтому мы сразу можем к ним обращаться.

ВСЕ КОМАНДЫ ОБРАЩЕННЫЕ К maven , БУДУТ ОТНОСИТЬСЯ К ТОЙ ДИРРЕКТОРИИ, ГДЕ ЛЕЖИТ pom-ка.

Т.е. для обращения к конкретной pom-ке нужно зайти в эту директорию(В её модуль).

Учитывая 3 составляющих:

- superPom

- дефолтные плагины

- наша pom

Можно создать результирующую pom

Перейдя в терминал и набрав

**mvn help:effective-pom**

этот плагин сгенерирует все эти три части в одну pom.

effective pom – та pom, с которой в итоге работают

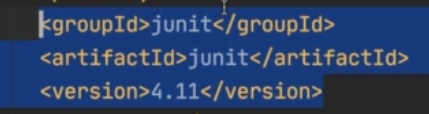
**Dependency management**

Тэг dependency используется нами, когда нам нужно привнести новую зависимость в наш проект (jar-файлы). Ранее чтобы подключить такую зависимость нам нужно было её скачать и положить в директорию lib, помечая эту директорию как classpath(это пути по которым наше java-приложение будет искать классы которые мы используем в нём) Чтобы упростить это дело и служит тэг dependency.

Подключая таким образом зависимости они автоматом будут в classpath нашего проекта.

Для подключения зависимости нужно указать её уникальный идентификатор

А он состоит из 3-х элементов



Если установить курсор в версии и нажать ctrl + пробел, то откроются все актуальные версии зависимости имеющейся в репозитории maven2

ТО есть работает след схема:



У нас есть first-maven проект и он ищет свои зависимости у нас локально на машине в папке ~/.m2/repository/ если здесь не находит, то идет в удаленный репозиторий, который прописан в superPom и оттуда скачивает зависимость в наш локальный репозиторий.

Директорию .m2 можно переписать, но это делается в настройках нашего maven-a.

В директории conf в файле settings.xml

Автоматическая настройка зависимостей(подтягивание зависимостей):

Settings – Build – Build Tools – Any changes

Замечание: хорошо для маленьких проектов;

Для продакшен лучше не использовать, иначе долго можно ждать процесса синхронизации.

Dependency Scope

Существует 5 типов scope:

* compile (идет по умолчанию – обозначает, что эта зависимость нужна для компиляции нашего проекта)
* provided (означает что эта зависимость будет предоставлена кем-то другим)
* runtime (пример – драйвер в jdbc, во время компиляции не надо, а во время выполнения нужно)
* system (означает, что мы указываем зависимость, что лежит у нас на компе, соответственно мы указываем путь к jar-ке, что лежит у нас на компе)
* test (зависимость нужна только для выполнения наших тестов)

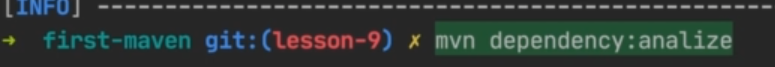
Есть dependency плагин, который позволяет посмотреть все зависимости, которые наследуются от других pom, других модулей

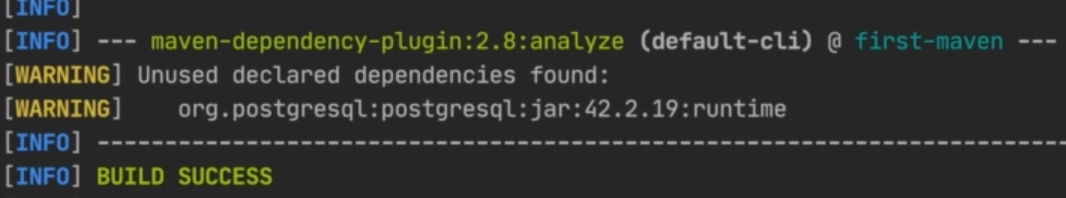
Для просмотра всех голов плагина набираем:

mvn dependency:help

нас интересуют следующие голы:

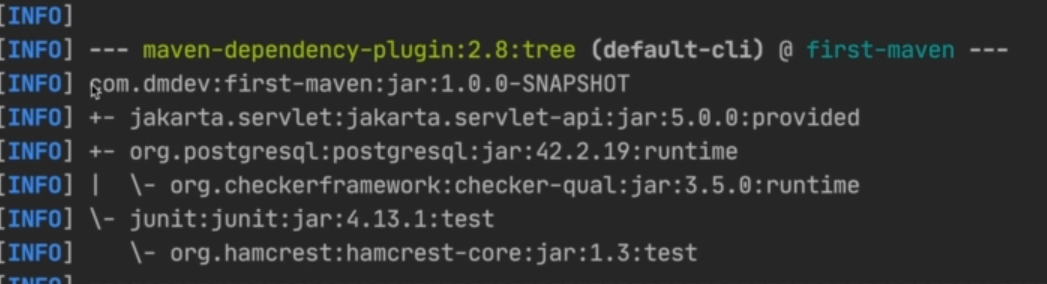
analyze – анализирует наши зависимости, что можно убрать, что нет.





Например, есть зависимость, которая никогда не используется(postgresql).

tree – билдит всю иерархию зависимостей(отображает дерево зависимостей проекта)



Видно, что проект собирается в jar-ку и он зависит от:

- Jakarta;

- postgresql;

- junit.(в свою очередь junit зависит от hamcrest, такая зависимость называется транзитивной)

Как maven определяет транзитивные зависимости?

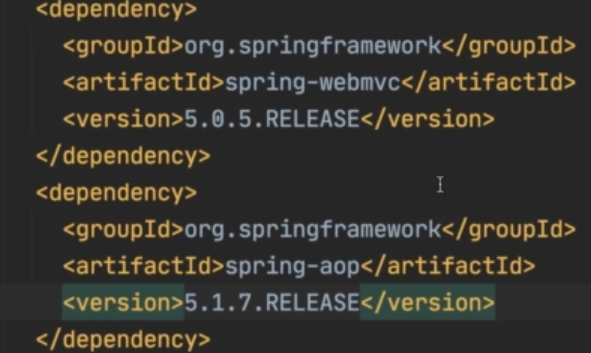
Когда он скачивает зависимость из репозитория с сервера, скачивается и pom файл этой зависимости(библиотеки), в котором и приписаны эти зависимости.

Какие есть нюансы с транзитивными зависимостями:

Одни транзитивные зависимости будут перекрывать другие транзитивные зависимости, если «главная» зависимость идет выше в нашем конфигурационном файле(поэтому ТрЗ spring-webmvc перекрывает ТрЗ spring-aop, не смотря на то, что у spring-aop версия выше. В Gradle решает версия, не зависимо от порядка)

Чтобы решить эту проблему используют тэг <exclusions>

Подключим зависимость spring-a:



Проблемы возникают тогда когда эти зависимости не совпадают по версии с другими транзитивными зависимостями(то же название, но версия другая)

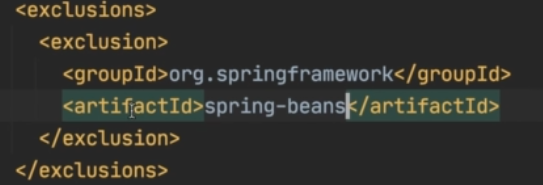
Запустим гол tree с доп флагом:



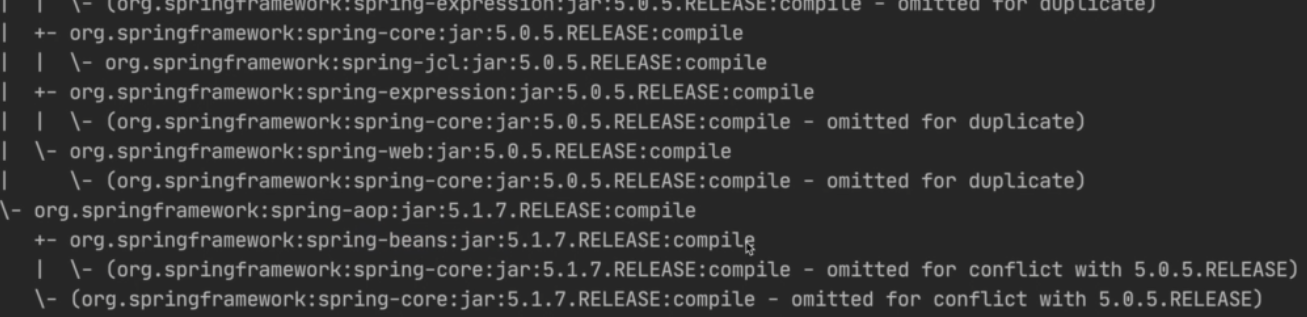
Одни транзитивные зависимости будут перекрывать другие транзитивные зависимости, если «главная» зависимость идет выше в нашем конфигурационном файле(поэтому ТрЗ spring-webmvc перекрывает ТрЗ spring-aop, не смотря на то, что у spring-aop версия выше. В Gradle решает версия, не зависимо от порядка)



Чтобы решить эту проблему используют тэг <exclusions> , где указываются зависимости, которые мы хотим исключить

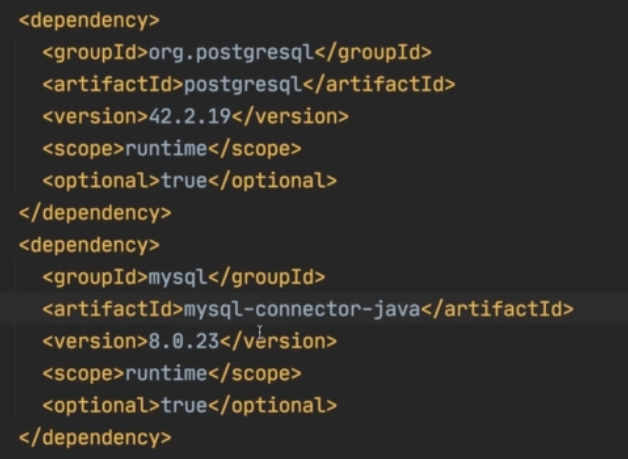


И если сейчас сгенерировать дерево, то получим:



Как видим версия уже более новая.

Тэг <Optional> - true/false(по умолчанию)



Другой проект, который будет в качестве зависимости подключать наш **first-maven** транзитивно должен подтянуть **postgesql** и **mysql-connector-java,** но он этого делать не будет т.к. **optional** установлено **true,** но он должен выбрать одну из этих опционных зависимостей, и тот кто подключает, должен сам выбрать, какая зависимость ему нужна.

На практике лучше не использовать – доп сложности.

**Интеграция Intellij и maven**

В отдельном окне наш проект first-maven

-в нем его Lifecycle (4-я секция Build Settings),

- все плагины (Plagins), которые у нас используются и используются для жизненного цикла нашего проекта.

- Dependencies (основные, транзитивные, какие опущены, ввиду конфликтов)

**Project lifecycle**

PL – набор фаз(phase), которые следуют по очереди друг за другом.

Phase – этап жизненного цикла, который состоит из целей(goal-ов).

Жизненный цикл проекта позволяет позволяет полностью управлять жизнедеятельностью нашего проекта

Аналогия PL – сутки, phase – часть этих суток

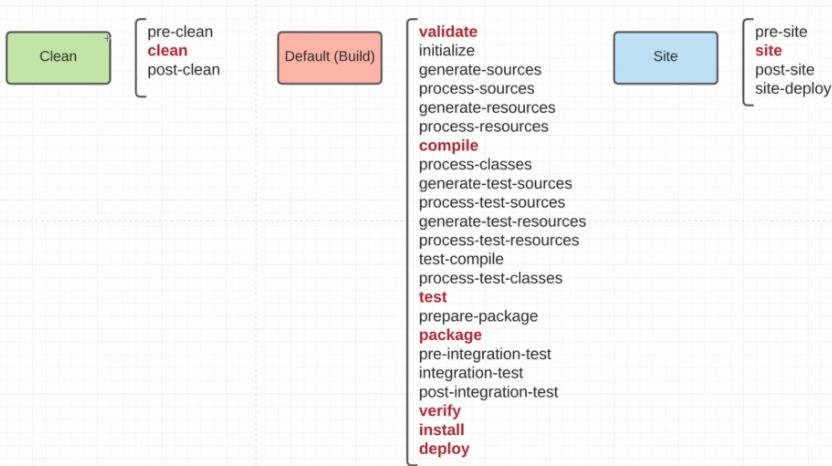
По коду phase – это enam, с перечнем всех возможных фаз в maven

В maven существует 3 жизненных цикла:

- Clean;

- Default (build)

- Site



У каждого из них есть набор фаз, которые следуют друг за другом.

Напр. В clean : pre-clean – clean – post-clean

Основная фаза отмечена красным – clean.(чистит директорию target, директория жизнедеятельности нашего проекта)

Default-ый является главным, т.к. он занимается всем, что связано с проектом.

Его осн фазы:

Validate – валидирует все наши pom-ки в нашем проекте

Compile – компиляция проекта, те что лежат в main

Test – запускает unit тесты, для проверки исходных кодов

Package – упаковка проекта в соответствующий артефакт(war, jar)

Verify – запуск интеграционных тестов на основании того, что мы упаковали

Install – копирование артефактов с фазы package в локальный репозиторий(/m2/repository);

Deploy – берет проект из локального репозитория и пушит его в удаленный

С каждой фазой может быть связано одна, несколько или ни одной цели.

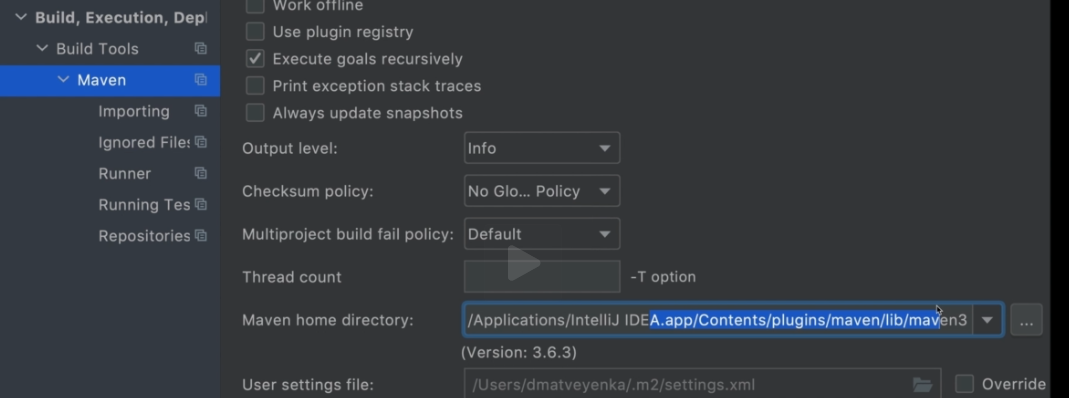
В случае дефолтного лайфсайкла набор целей зависит от того, во что мы собираем проект (package). Это будет jar/war или обычная pom-ка

Фаза site у ls Site занимается генерацией отчетов, репортов. (генерирует документацию по исходным файлам, отчеты по результатам тестов)

Можно самостоятельно связывать цели с определенными фазами.

Если открыть вкладку maven в Intellij, то можно увидеть основные цели lifecycle и плагины которые отвечают за эти цели, которые срабатывают в зависимости от того, какую из фаз мы выберем.

При вызове фаз используя модуль maven в Intellij, Мы используем не тот maven, что прописан в path, а тот что находится в Intellij



Чтобы использовать другие фазы в определенном lc то мы должны подключить свои голы у плагинов и использовать их в нужных фазах.(обо этом позже)

При фазе validate никаких голов не вызывается

Чтобы вызвать compile нужно пройти предыдущие фазы до compile. И если нам нужно вызвать какую либо фазу, вызываются все фазы до неё.

Так например если вызвать site-deploy вызовутся также pre-site – site – post-site и только потом site-deploy.

Разберем далее фазы lc Build:

Initialize – на этой фазе может быть привязан какой-то свой плагин и установлены свои проперти, создать директории, что-то для доп инициализации проекта

generate-sources – тоже можем привязать плагин, расчитана на то, что мы хотим сгенерировать какие-то исходники. Есть генераторы типа Querydsl, которые создают дополнительно классы на этой фазе(hibernate, spring)

process-sources – обрабатываем все исходники(фильтруем их, подставляем значения в них)

generate-resources – аналог generate-sources, но генерирует ресурсные файлы из нашей директории resources

process-resources – фаза, на которую уже привязана первая наша goal из плагина resources, которая обрабатывает наши ресурсные файлы. Т.е. берет директорию target и кладет туда все наши ресурсы из папки main-resources, также может дополнительно отфильтровать их, поставить какие-то значения и пр.(также как и process-sources, только для статических файлов из директории resources)

compile – к фазе привязан compiler-plugin и его goal -> compile. Он компилирует наши исходники из main-src и кладет в target-classes