**Maven**

<https://www.youtube.com/watch?v=IAbZVA4tK6M> – часовой курс основ

<https://www.youtube.com/watch?v=gv60974ebDY> – создание проекта в связке GitHub + Idea + Maven + Junit тесты. Кроме того она тут меняет разные начальные настройки и подгружает новые версии библиотек.

На выходе любого проекта Maven всегда один файл JAR(*в zip + manifest файлы*)/WAR(*структура папок для web приложений*)/EAR(*архив jar-ов и war-ов. Сейчас не используется. Вместо него используется war*).

Каждый проект в Maven называется артефакт.

При необходимости собрать один проект из нескольких существует два подхода:

- в <dependencies> своего проекта прописать ссылку на необходимый собранный JAR/WAR другого проекта

- изначально создаваться от одного родителя, прописывая все необходимые своему проекту (*который является дочерним*) зависимости в <dependencies> родительского проекта.

Eclipse не может создавать подпроекты (модули). Все они будут созданы на одном уровне.

Если какое-то доп.ПО, прописанное в <dependencies> не находится Maven-ом, то его с 90% вероятностью нужно удалить и не использовать в дальнейшем. Т.к. все нормальные (постоянно используемые и “не кривые”) дополнения есть в Maven.

Папка “target” создается в Maven-проекте когда мы создаем/собираем общий файл всего проекта jar или war. И в нее помещается весь проект.

При создании Git-репозитория нужно также сразу на том-же листе выбрать - создать файл “.gitignore” для Maven + поставить галочку **Initialize this repository with a README.**

В IDEA выбрать “Get from Version Control”, вставить http ссылку из GitHub, на вопрос о создании типового проекта IntelliJ IDEA ответить НЕТ, после чего выбрать пункт “Create New Project”, там Maven + “Create from archetype”, потом в Artifact Coordinates в ArtifactId набрать название как у репозитория GitHub. В открывшемся проекте на предложенных внизу сообщениях выбираю Enable Auto-Import и Always Add.

Для правильной работы механизмов Java необходимо проверить в File/Project Structure/закладках Project и Modules установленные версии, и если версии старые то выбрать и установить свои 10 версии. После чего

написать в командной строке “mvn clean install” и он по новой подгузит все зависимости. После чего в Idea находясь на листе “pom.xml” нажать пр.кн.мыши, выбрать пункт Maven, а там Reimport.

В файл “.gitignore” нужно добавить (к уже исключенным ресурсам) “.idea” и “\*.iml”.

В файле “pom.xml” изменить версию Junit на 12-ю + добавить плагины автоматической проверки maven-checkstyle-plugin и maven-pmd-plugin. которые запускаются командой “mvn checkstyle:check”

Блок с плагинами автоматической проверки:

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-pmd-plugin</artifactId>  
 <version>2.5</version>  
 <configuration>  
 <targetJdk>1.6</targetJdk>  
 <verbose>true</verbose>  
 </configuration>  
 <executions>  
 <execution>  
 <phase>package</phase>  
 <goals><goal>check</goal></goals>  
 </execution>  
 </executions>  
</plugin>  
<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
 <version>2.7</version>  
</plugin>

которые запускаются командой “mvn checkstyle:check” и “mvn mvn:check” (на общие команды типа “mvn install” не реагирует ??). Положит в папку target/ в файл checkstyle-result.xml и папку target/site/ в файл pmd.html*(который запускать - пкм “open in browser”)*.

Блок для Javadoc:

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-javadoc-plugin</artifactId>  
 <version>3.2.0</version>  
 <executions>  
 <execution>  
 <phase>prepare-package</phase>  
 <goals>  
 <goal>javadoc</goal>  
 </goals>  
 </execution>  
 </executions>  
 <configuration>  
 <!-- optional -->  
 <javadocExecutable>${java.home}/bin/javadoc</javadocExecutable>  
 </configuration>  
</plugin>

запускается командой “**mvn javadoc:javadoc**” (или общей “mvn install”). Сложит все в папку target/site/apidocs/resources/. Запускать файл “***имя класса.html”*** из пкм “open in browser”.

Также возможно для нормальной работы по сборке проекта может понадобиться изменить/добавить блок:



Параметры генерации:

* **<project>** является оболочкой всего проекта
* **<parent> -** объявление родительского pom.xml из которого будут браться основные настройки
* **<groupId>**=com.mycompany.app указывает на разработчика ПО, там вы можете указать доменное имя своей компании (*пишется наоборот*).
* **<artifactId>**=my-app — название проекта. Maven оперирует так называемыми артефактами. Это приложения, плагины, архетипы и другие проекты. И ваша разработка также будет считаться артефактом.
* **<version>**  обычно генерируется и обновляется автоматически. После номера версии идёт суффикс -SNAPSHOT. Это означает, что проект находится в стадии разработки. В процессе выпуска ПО фреймворк уберёт этот суффикс, а если разработка продолжится — автоматически увеличит номер версии.

**groupId, artifactId и version однозначно идентифицируют проект.**

* **<archetypeArtifactId>**=maven-archetype-quickstart указывает, какой архетип использовать в качестве шаблона для создания приложения. Как видите, это тоже артефакт. Указанный архетип создаст простой проект, сгенерирует структуру каталогов и даже заботливо положит в папку исходников программу «Hello, world!», чтобы вам не пришлось самому писать её в двухсотый раз.
* **<archetypeVersion>**=1.4 указывает на версию артефакта «архетип».
* **<interactiveMode>**=false отключает создание проекта в интерактивном режиме. Вы можете запустить цель archetype:generate без параметров, и Maven предложит установить параметры в процессе генерации.
* **<name>** содержит отображаемое имя артефакта(имя всего проекта).
* **<url>** содержит ссылку на сайт.
* **<description>** содержит краткое описание

Предыдущие три тега часто используются при формировании документации.

* **<packaging>** указывает тип(расширение) файла в который будет собран весь проект.
* **<modules>** и вложенные в него тэги <**module>**, содержит информацию о добавленных к основному “родительскому” проекту с файлом pom.xml дочерних модулей(подпроектов) со своими pom.xml, для разделения проекта на структурные блоки (например на логику и визуальную часть) с целью улучшения читаемости всего проекта. Любая зависимость из парент проекта добавляется дочерние подпроекты. Причем наш основной pom.xml тоже имеет родительский pom.xml самого Maven, и в нем хранится большое количество всех основных зависимостей.
* **<properties>** указывают особые настройки, такие как кодировка файла или используемая версия или имя переменной :

где “my.filter.value” это имя переменной, а hello это ее значение.

Переменные прописываются в верху pom.xml, а используются так, например:

<outputDirectory>${my.filter.value}</outputDirectory>

* **<dependencies>** описывает все используемые в проекте зависимости. Каждую необходимо выделить тегом **dependency** и указать уникальные идентификационные данные: <**groupId>**, <**artifactId>** и <**version>**. Maven сам подгружает транзитивные зависимости*(зависимости зависимостей)*.

- С помощью тега <**scope>** можно указать этапы, на которых будет использован артефакт. При этом в итоговый JAR файл такая зависимость не попадет.

- Также **<dependencies>** можно прописать для конкретного плагина.

- Любая зависимость из парент проекта добавляется в дочерние подпроекты. Причем наш основной pom.xml тоже имеет родительский pom.xml самого Maven, и в нем хранится большое количество всех основных зависимостей.

- Maven не разрешает циклические зависимости.

- Если долго не подгружается какая-то зависимость, то можно написать в командной строке “*mvn clean install*” и он по новой подгузит все зависимости. После чего в Idea находясь на листе “pom.xml” нажать пр.кн.мыши, выбрать пункт Maven, а там Reimport.

* **<reporting>** <reporting><plugins><plugin>- - -</plugin></plugins></reporting> содержит информацию о необходимой отчетности.
* **<build>** содержит информацию по самой сборке: где находятся исходные файлы, где ресурсы, какие плагины используются. В нем могут содержаться следующие теги:
* **<sourceDirectory>** определяет, откуда maven будет брать файлы исходного кода. По умолчанию это src/main/java, но вы можете определить, где это вам удобно. Директория может быть только одна (без использования специальных плагинов)
* **<resources>** и вложенные в неё тэги **<resource>** определяют, одну или несколько директорий, где хранятся файлы ресурсов. Ресурсы в отличие от файлов исходного кода при сборке просто копируются . Директория по умолчанию src/main/resources. Если же по какой-то причине нужно переопределить каталог, используйте вложенные теги **resources**, **resource**, **directory** в секции **build**:

<build>

<resources>

<resource>

<directory>src/main/another\_resources\_directory</directory>

</resource>

</resources>

</build>

* **<outputDirectory>**определяет, в какую директорию компилятор будет сохранять результаты компиляции - \*.class файлы. Значение по умолчанию - target/classes
* **<finalName>**- имя результирующего jar (war, ear..) файла с соответствующим типу расширением, который создаётся на фазе package. Значение по умолчанию — artifactId-version.
* **<pluginManagement>** используется для указания, что такие плагины будут видны во всех модулях проекта.
* **<plugins>** и вложенные в неё тэги **<plugin>**, плагины это подгружаемое доп.ПО, которое позволяет задать дополнительные действия, которые будут выполняться при сборке. (как сказал лектор – плагин это обычный **JS** код).

Например для автоматической проверки кода на наличие "плохого" кода и потенциальных ошибок:

<plugins>

<plugin>

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

<artifactId>maven-pmd-plugin</artifactId>

<version>2.4</version>

</plugin>

</plugins>

После того как плагин объявлен, его можно настроить так, чтобы он автоматически запускался в нужный момент. Это делается с помощью привязки плагина к [фазе](https://www.apache-maven.ru/lifecycle.html) сборки проекта:

<plugin>

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

<artifactId>maven-checkstyle-plugin</artifactId>

<version>2.6</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>check</goal> //это значит, что “check” будет командой запуска этого плагина

</goals>

</execution>

</executions>

</plugin>

в данном примере плагин запустится в фазе проекта package.

Для работы большинства плагинов обычно требуются дополнительные настройки, которые специфичны для конкретного плагина. Настройки задаются в тэгах <configuration>. Например так настраивается tomcat - плагин:

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>tomcat-maven-plugin</artifactId>

<version>1.1</version>

<configuration>

<fork>false</fork>

<server>test-server</server>

<url>http://test-server/manager</url>

</configuration>

</plugin>

Содержимое в тэгах зависит от конкретного плагина и описывается в документации по плагину.

Три основных цикла (lifecycle):

clean – очистка

default – сборка

site – формирование отчетов

Основные фазы (phase) подциклов и команды(**goal**) по их запуску:

подготовка ресурсов – (**\*-resources**)

валидация (проверка правильности параметров) – **validate**

компиляция (собирает весь исходный код) – **compile**(компилирует +создает папку target и ее подпапки)

тестирование (unit и интеграционные) – **test**

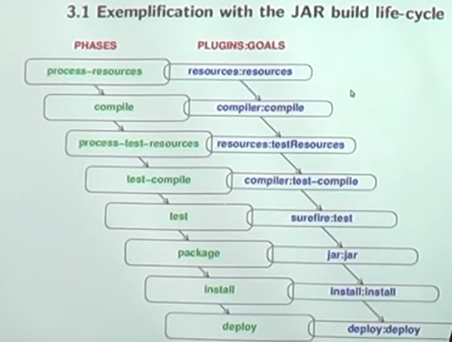
упаковка в JAR(по умолчанию)/WAR/EAR - **package**

установка (в репозиторий) – **install** *(если какой-то зависимости нет, то нужно писать mvn install -U “зависимость” и она вытянется).* Команда- положит pom.xml по такому адресу.

деплой на продакшин(развертывание) – **deploy**

*При наборе команды mvn compile выполняться вначале команды (\*-resources) потом validate и только потом compile*

Фазы (команды) при указании в <packaging> **jar**



Создается jar файл

Surefire запускаются Unit тесты и генеряться отчеты

Компилируются Unit тесты

Копируются конфигурационные файлы для Unit тестов

Можно выбрать какой версии Java будет компилироваться

jar файл записывается в remote репозиторий

jar файл записывается в \*\.m2\repository

Копируются файлы и фильтруются переменные

тут написано как можно запустить каждую фазу: mvn “goal” или mvn “plugin:goal”.

Команды (goal) например для цикла clean:

pre-clean (для выполнения действий перед очисткой при которой пропадет папка target )

clean

post-clean (для выполнения действий после очистки)

Для нормальной работы команд в Maven нужно в pom.xml обязательно добавить блоки со ссылками на новые блоки ПО *(т.к. те, что по умолчанию установлены, уже устарели и не работают)*:

<properties>  
 <maven.compiler.source>1.6</maven.compiler.source>  
 <maven.compiler.target>1.6</maven.compiler.target>  
</properties>  
  
<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-site-plugin</artifactId>  
 <version>3.7.1</version>  
 </plugin>  
 </plugins>  
</build>

А может нужно будет писать вот так (если проблемы на этапе компиляции):

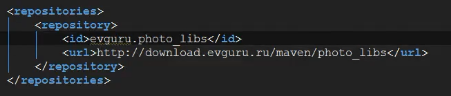


**Репозитории** (**repositories**)- это место где хранятся артефакты: jar файлы, pom -файлы, javadoc, исходники.

Local – локальный у меня на компе, папка C:\Users\Макс\.m2\repository

Central - центральный репозиторий всего Maven расположенный по адресу https://search.maven.org/#browse, в котором ищется необходимая зависимость с кодом для добавления в файл pom.xml.

Remote – удаленный репозиторий для хранения своих зависимостей, которых нет в Central репозитории. Такие репозитории указываются в файле pom.xml:

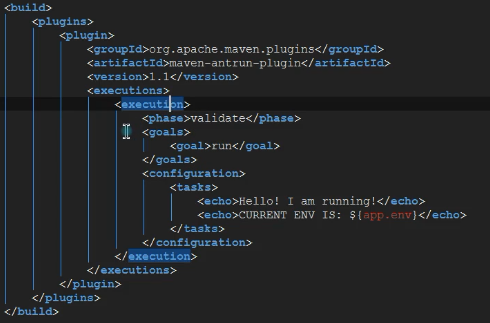


**Плагины (plugins)**

Это программы для выполнения команд типа “mvn имя плагина: команда” и д.р.

Плагины бывают 2-х типов:

- build –сборки, они должны быть размещены внутри тега <build><plugins><plugin>\_</plugin></plugins></build>



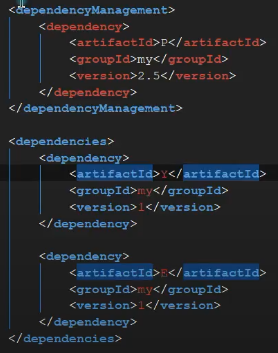
- report – отчетов. Тоже, что и build только тег <build> меняется на тег <report>.

Зависимости(Dependencies)

Логика работы Moven при выборе зависимости



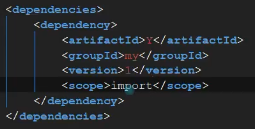
при такой ситуации Maven выберет и туда и туда Р версии 2.1 т.к. до нее ближе путь. А если нужно, чтобы он выбрал более новую версию Р 2.5, то в файле pom.xml нужно написать:



Скоупы(Scopes)(область видимости)

Они указывают на каком этапе (в каком случае) подключать зависимость.

Они указываются внутри тега <dependencies><dependency><scope>*import/runtime/compile/provided/test*</scope></dependency></dependencies>:



*compile* - по умолчанию;

*provided –* используется когда такая зависимость уже есть в общем файле проекта на сервере и класть ее туда еще раз не нужно.

*runtime –* указывает, что зависимость сразу компилировать не нужно, тесты пройдут без нее и она будет получена по запросу во время выполнения определенного блока кода

*test –* указывает, что эта зависимость необходима только для тестирования

*import –* указывает какая зависимость должна быть заменена списком других зависимостей

Модули(Modules)

Модули служат для отображения разных компонентов например бизнес логики и отображения.

Для создания модульного проекта необходимо, чтобы тип упаковки был не JAR/WAR/EAR, а “pom”.

Общие для всех модулей зависимости выносятся в верхний модуль и в остальные(внутренние) модули наследуются от него.