Оглавление

[Объектная модель описания проекта (Project Object Model) 3](#_Toc59527884)

[Супер POM 3](#_Toc59527885)

[Пример файла pom.xml 3](#_Toc59527886)

[Корневой тег <project> 5](#_Toc59527887)

[Тег <modelVersion> 6](#_Toc59527888)

[Общая информация о проекте 6](#_Toc59527889)

[GAV параметры описания проекта 7](#_Toc59527890)

[Секция свойств <properties> 8](#_Toc59527891)

[Предопределённые переменные maven 9](#_Toc59527892)

[Кодировка maven проекта 10](#_Toc59527893)

[Секция родительских модулей <parent> 10](#_Toc59527894)

[Секция модулей <modules> 11](#_Toc59527895)

[Секция репозиториев <repositories> 11](#_Toc59527896)

[Локальный репозиторий 12](#_Toc59527897)

[Удалённый репозиторий 13](#_Toc59527898)

[Секция зависимостей <dependencies> 15](#_Toc59527899)

[Область действия зависимости, scope 16](#_Toc59527900)

[Транзитивные зависимости 17](#_Toc59527901)

[Секция репозиториев плагинов <pluginRepositories> 17](#_Toc59527902)

[Секция сборки проекта <build> 18](#_Toc59527903)

[Секция плагинов <plugins> 19](#_Toc59527904)

[Запуск плагина 20](#_Toc59527905)

[Полезные плагины 22](#_Toc59527906)

[Жизненный цикл сборки 25](#_Toc59527907)

[Фазы 25](#_Toc59527908)

[Выполнение фаз 26](#_Toc59527909)

[Цели 28](#_Toc59527910)

[Выполнение целей 28](#_Toc59527911)

[Секция отчетов <reporting> 29](#_Toc59527912)

[Конфигурационные файлы 31](#_Toc59527913)

[Структура конфигурационного файла 31](#_Toc59527914)

[Простые элементы верхнего уровня 31](#_Toc59527915)

[Серверы 32](#_Toc59527916)

[Зеркала 32](#_Toc59527917)

[Прокси-серверы 33](#_Toc59527918)

[Создание проекта 34](#_Toc59527919)

[Архетипы 34](#_Toc59527920)

[Параметры при создании проекта 34](#_Toc59527921)

[Создание проекта Maven в IntelliJIDEA 35](#_Toc59527922)

[Стандартная структура для Maven-проекта 36](#_Toc59527923)

# Объектная модель описания проекта (Project Object Model)

**POM файл (pom.xml)** – XML файл, который содержит декларативное описание проекта (и его сборки) и находится в корневой директории проекта. С помощью pom.xml конфигурируются зависимости от других проектов, индивидуальные фазы процесса построения проекта (build process), список плагинов, реализующих порядок сборки. POM файлы должны распространяться вместе с проектами.

При запуске Maven проверяет, содержит ли конфигурационный файл все необходимые данные и все ли данные синтаксически правильно записаны. Минимальная конфигурация включает версию конфигурационного файла, имя проекта, его автора и версию.

Крупные проекты могут быть поделены на несколько модулей, или подпроектов, каждый со своим собственным POM. Операции над модулями могут выполняться через общий корневой POM единой командой. POM-файлы подпроектов могут наследовать конфигурацию от других файлов конфигурации.

## Супер POM

В то же время все файлы конфигурации обязательно наследуются от **Супер POM** файла по умолчанию. Супер POM обеспечивает конфигурацию по умолчанию, например, стандартная структура каталогов, используемые по умолчанию плагины, привязка к фазам жизненного цикла и прочее.

Простой способ просмотреть настройки по умолчанию Super POM файла – это использование команды:

mvn help:effective-pom

## Пример файла pom.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

    http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

*<!-- Общая информация о проекте -->*

    <name>Simple Maven Project</name>

    <description>...</description>

    <url>...</url>

    <inceptionYear>...</inceptionYear>

    <licenses>...</licenses>

    <developers>...</developers>

    <contributors>...</contributors>

    <organization>...</organization>

    <issueManagement>...</issueManagement>    *<!-- Система отслеживания ошибок -->*

    <mailingLists>...</mailingLists>

*<!-- GAV параметры описания проекта -->*

    <groupId>...</groupId>

    <artifactId>...</artifactId>

    <version>...</version>

    <packaging>...</packaging>

*<!-- Секция свойств -->*

    <properties>

    ...

    </properties>

*<!-- Секция родительских модулей -->*

    <parent>

    ...

    </parent>

*<!-- Секция дочерних модулей -->*

    <modules>

    ...

    </modules>

*<!-- Секция репозиториев -->*

    <repositories>

    ...

    </repositories>

*<!-- Секция зависимостей -->*

    <dependencies>

    ...

    </dependencies>

*<!-- Секция репозиториев плагинов -->*

    <pluginRepositories>

    ...

    </pluginRepositories>

*<!-- Секция сборки -->*

    <build>

        <finalName>projectName</finalName>

        <sourceDirectory>${basedir}/src/java</sourceDirectory>

        <outputDirectory>${basedir}/targetDir</outputDirectory>

        <resources>

            <resource>

                <directory>${basedir}/src/java/resources</directory>

                <includes>

                    <include>\*\*/\*.properties</include>

                </includes>

            </resource>

        </resources>

*<!-- Секция плагинов -->*

        <plugins>

        ...

        </plugins>

    </build>

*<!-- Секция сборки -->*

    <reporting>

    ...

    </reporting>

    <distributionManagement>

    ...

    </distributionManagement>

</project>

Не все секции могут присутствовать в описании pom.xml. Так секции properties и repositories часто не используются. Параметры GAV проекта являются обязательными.

## Корневой тег <project>

Корневой элемент **<project>** является базовым и содержит всю информацию о проекте. В заголовке указана информация, необходимая Maven для понимания файла pom.xml (схема облегчающая редактирование и проверку, а также версия POM).

Этот тег обычно генерируются автоматически, менять его не нужно.

<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">

    ...

</project>

## Тег <modelVersion>

Тег **<modelVersion>** указывает на текущую версию POM.

Этот тег обычно генерируются автоматически, менять его не нужно.

<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>

...

</project>

## Общая информация о проекте

Общая информация о проекте используется для создания документации проекта и как правило не влияет на сборку проекта:

* name – человеческое имя проекта.

Тег **<name>** содержит отображаемое имя артефакта.

* url — ссылку на сайт.

Тег **<url>** отображает ссылку на сайт размещения проекта. Поскольку сайт не задан при генерации, pom.xml содержит напоминание об этом в виде комментария.

* description – краткое описание проекта.

Тег **<description>** содержит краткое описание.

Эти три тега зачастую используются при формировании документации.

<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">

...

*<!-- Общая информация о проекте -->*

    <name>Simple Maven Project</name>

    <description>...</description>

    <url>...</url>

    <inceptionYear>...</inceptionYear>

    <licenses>...</licenses>

    <developers>...</developers>

    <contributors>...</contributors>

    <organization>...</organization>

    <issueManagement>...</issueManagement>    *<!-- Система отслеживания ошибок -->*

    <mailingLists>...</mailingLists>

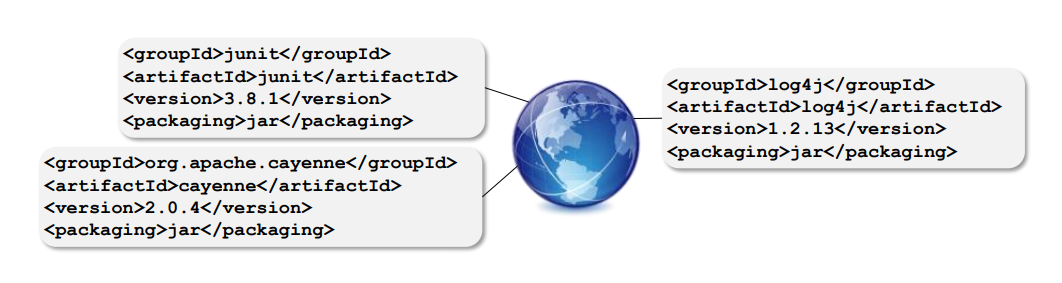
...

</project>

## GAV параметры описания проекта

**Артефакт** – любая библиотека, хранящаяся в репозитории, результат сборки проекта. Могут быть разных типов jar, war, zip, swf. Это может быть какая-то зависимость или плагин.

**Координаты артефакта (GAV параметры)** – уникальный набор параметров, состоящий из идентификатора группы, идентификатора артефакта, версии и упаковки.



**Полное имя артефакта (координата)** представляет четыре слова, разделенные знаком двоеточия в следующем порядке **groupId:artifactId:packaging:version** :

* groupId – идентификатор группы (организации, компании, команды);

Тег **<groupId>** определяет наименование организации или подразделения производителя объекта. Как правило идентификатор группы начинается с обратного доменного имени организации которая создала проект, то есть обычно действуют такие же правила, как и при именовании пакетов в Java.

* artifactId – идентификатор артефакта;

Тег **<artifactId>** обозначает название проекта. Это уникальный идентификатор среди проектов с одинаковыми идентификаторами группы groupId который определяет один проект.

* version – версия проекта;

Тег **<version>** хранит версию проекта. Он обычно генерируется и обновляется автоматически.

Проекты, которые были выпущены имеют конкретную версию.

Проекты, которые активно разрабатываются могут использовать специальный идентификатор SNAPSHOT. В процессе выпуска ПО фреймворк уберёт этот суффикс, а если разработка продолжится — автоматически увеличит номер версии.

Тройкой groupId, artifactId, version (далее - **GAV**) можно однозначно идентифицировать файл приложения или библиотеки.

* packaging – тип упаковки проекта.

Тег **<packaging>** определяет какого типа файл будет создаваться как результат сборки. Возможные варианты pom, jar, war, ear. Тег является необязательным. Если его нет, используется значение по умолчанию тип – jar. Проект с упаковкой jar собирается в JAR архив. Проект с упаковкой war собирается в WAR архив (веб приложение).

<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">

...

*<!-- GAV параметры описания проекта -->*

    <groupId>...</groupId>

    <artifactId>...</artifactId>

    <version>...</version>

    <packaging>...</packaging>

...

</project>

Назначение SNAPSHOT

Артефакты делятся на стабильные артефакты и артефакты с версией SNAPSHOT. Артефакты со стабильными версиями не должны изменяться. Артефакты с версиями SNAPSHOT могут изменяться. Стабильные артефакты не должны зависеть от артефактов с версиями SNAPSHOT.

***Пример:***

<groupId>com.examclouds</groupId>

<artifactId>courses</artifactId>

<version>1.0-SNAPSHOT</version>

## Секция свойств <properties>

Тег **<properties>** содержит особые настройки, такие как кодировка файла или используемая версия компилятора Java. Без этого блока можно обойтись, положившись на настройки по умолчанию.

<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">

    ...

*<!-- Секция свойств -->*

    <properties>

    ...

    </properties>

    ...

</project>

Отдельные настройки проекта можно определить в переменных. Это может быть связанно, к примеру, с тем, что требуется использовать семейство библиотек определенной версии. Для этого в проектном файле используется секция **<properties>**, в которой объявляются переменные. Обращение к переменной выглядит следующим образом:

${имя переменной}

***Пример:***

<properties>

    <junit.version>4.11</junit.version>

    <maven.compiler.source>1.4</maven.compiler.source>

    <maven.compiler.target>1.6</maven.compiler.target>

</properties>

...

<dependencies>

    <dependency>

        <groupId>junit</groupId>

        <artifactId>junit</artifactId>

        <version>${junit.version}</version>

        <scope>test</scope>

    </dependency>

</dependencies>

...

<build>

    <finalName>${project.artifactId}</finalName>

    <plugin>

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

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

        <version>2.3.2</version>

        <configuration>

            <source>${maven.compiler.source}</source>

            <target>${maven.compiler.target}</target>

        </configuration>

    </plugin>

</build>

### Предопределённые переменные maven

При описании проекта в pom-файле можно использовать предопределенные переменные. Их можно условно разделить на несколько групп:

Встроенные свойства проекта :

* ${basedir} – корневой каталог проекта, где располагается pom.xml;
* ${version} – версия артефакта; можно использовать ${project.version} или ${pom.version};

Свойства проекта (с помощью префиксов project или pom) :

* ${project.build.directory} – target директория (можно ${pom.build.directory});
* ${project.build.outputDirectory} - путь к директории, куда компилятор складывает файлы (по умолчанию target/classes);
* ${project.name} – наименование проекта (можно ${pom.name});
* ${project.version} – версия проекта (можно ${pom.version}).

Настройки settings.xml (с помощью префикса settings)

* ${settings.localRepository} путь к локальному репозиторию.

### Кодировка maven проекта

При выполнении отдельных команд Maven, связанных с копированием ресурсов или компиляцией, могут «выплыть» предупреждения о кодировке:

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ ...

[WARNING] Using platform encoding (Cp1251 actually) to copy filtered

                             resources, i.e. build is platform dependent!

[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ ...

[INFO] Changes detected - recompiling the module!

[WARNING] File encoding has not been set, using platform encoding Cp1251,

                                        i.e. build is platform dependent!

Чтобы обойти эти сообщения, необходимо включить в секцию **<properties>** следующий код с указанием требуемой кодировки :

<properties>

    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

## Секция родительских модулей <parent>

Maven позволяет собирать проект из нескольких модулей. Каждый программный модуль включает свой проектный файл pom.xml. Каждый модульный (дочерний) pom.xml должен включать параметры GAV (groupId, artifactId, version) корневого (родительского) pom.xml в теге **<parent>**.

<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">

    ...

*<!-- Секция родительских модулей -->*

    <parent>

    ...

    </parent>

    ...

</project>

***Пример:***

<parent>

    <groupId>com.example</groupId>

    <artifactId>project</artifactId>

    <version>0.0.1</version>

</parent>

## Секция модулей <modules>

Maven позволяет собирать проект из нескольких модулей. Каждый программный модуль включает свой проектный файл pom.xml. Один из проектных pom.xml файлов является корневым. Корневой pom.xml позволяет объединить все модули в единый проект. При этом в корневой проектный файл можно вынести общие для всех модулей свойства. В pom.xml родительского проекта необходимо определить параметры GAV (groupId, artifactId, version) и в теге <packaging> указать значение «pom». В секции **<modules>** перечисляются все дочерние проекты.

<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">

    ...

*<!-- Секция дочерних модулей -->*

    <modules>

    ...

    </modules>

    ...

</project>

***Пример:***

<groupId>com.example</groupId>

<artifactId>project</artifactId>

<version>0.0.1</version>

<packaging>pom</packaging>

...

<modules>

    <module>project1</module>

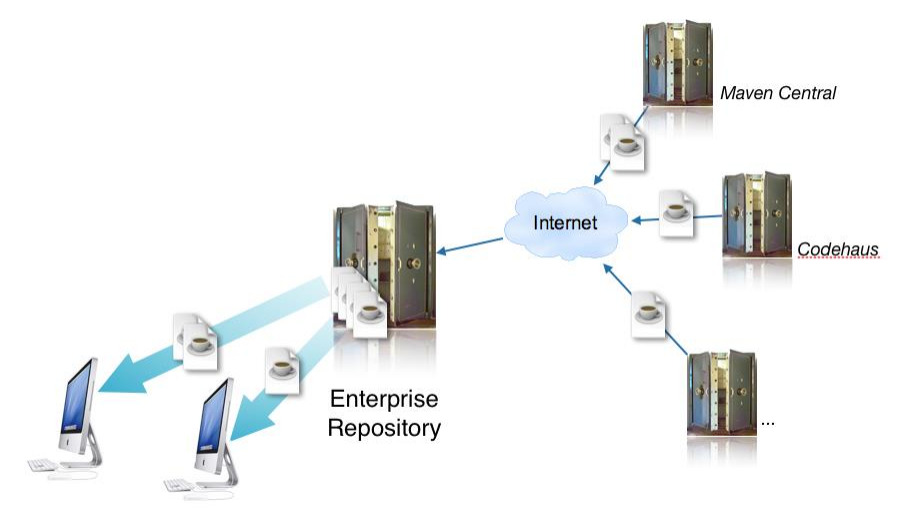
    <module>project2</module>

</modules>

## Секция репозиториев <repositories>

**Репозиторий** – место, где хранятся артефакты: jar файлы, pom файлы, javadoc, исходники.

В теге **<repositories>** можно определить дополнительные репозитории проекта.



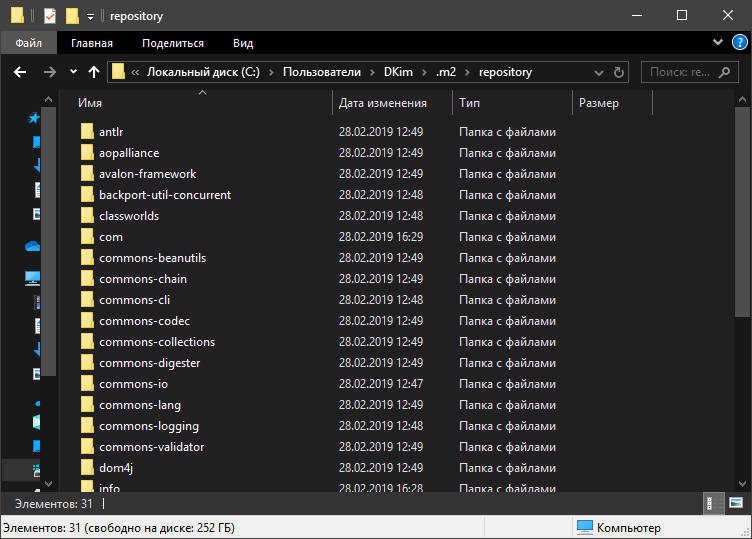
Существуют следующие виды репозиториев:

1. Локальный репозиторий
2. Удаленный репозиторий
   * Центральный репозиторий
   * Корпоративный репозиторий

### Локальный репозиторий

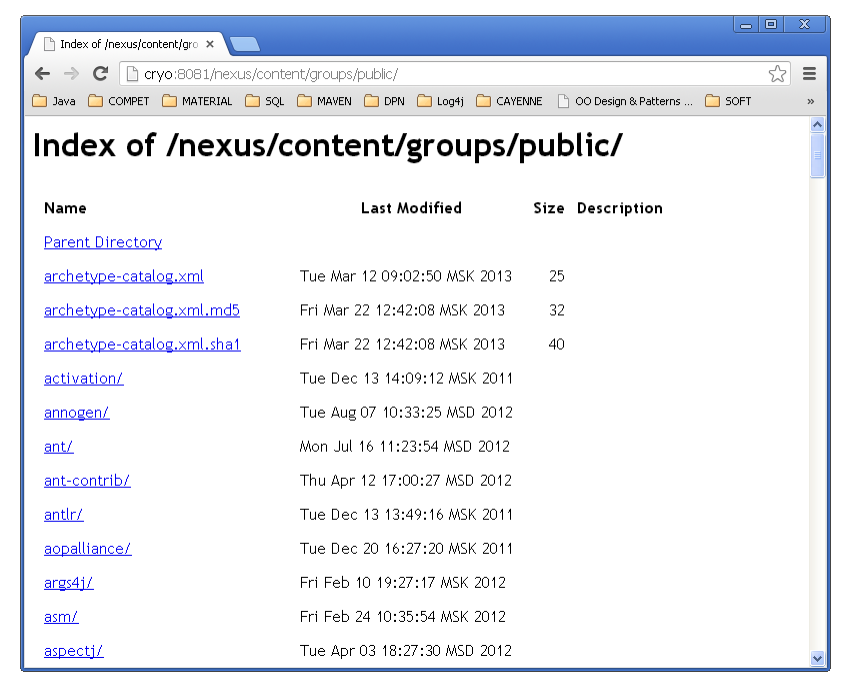
**Локальный репозиторий** – директория на диске содержащая дерево каталогов артефактов и персональный для каждого пользователя. По умолчанию это папка .m2/repository в домашней директории пользователя. Здесь лежат артефакты, которые были скачаны из центрального репозитория либо добавлены другим способом.

Фактически, это некоторый кэш основной части центрального репозитория. Во время сборки проекта все необходимые артефакты должны быть в локальном репозитории. Если какого-то необходимого артефакта нет он загружается из других репозиториев. В процессе работы над проектами, разработчики могут помещать в свой локальный репозиторий свои собственные артефакты для возможности их повторного использования.



### Удалённый репозиторий

**Удалённый репозиторий** – репозиторий на веб-сервере, который широко используется для размещения внутренних проектов организации. Когда Maven хочет загрузить зависимость, она переходит в удаленный репозиторий.

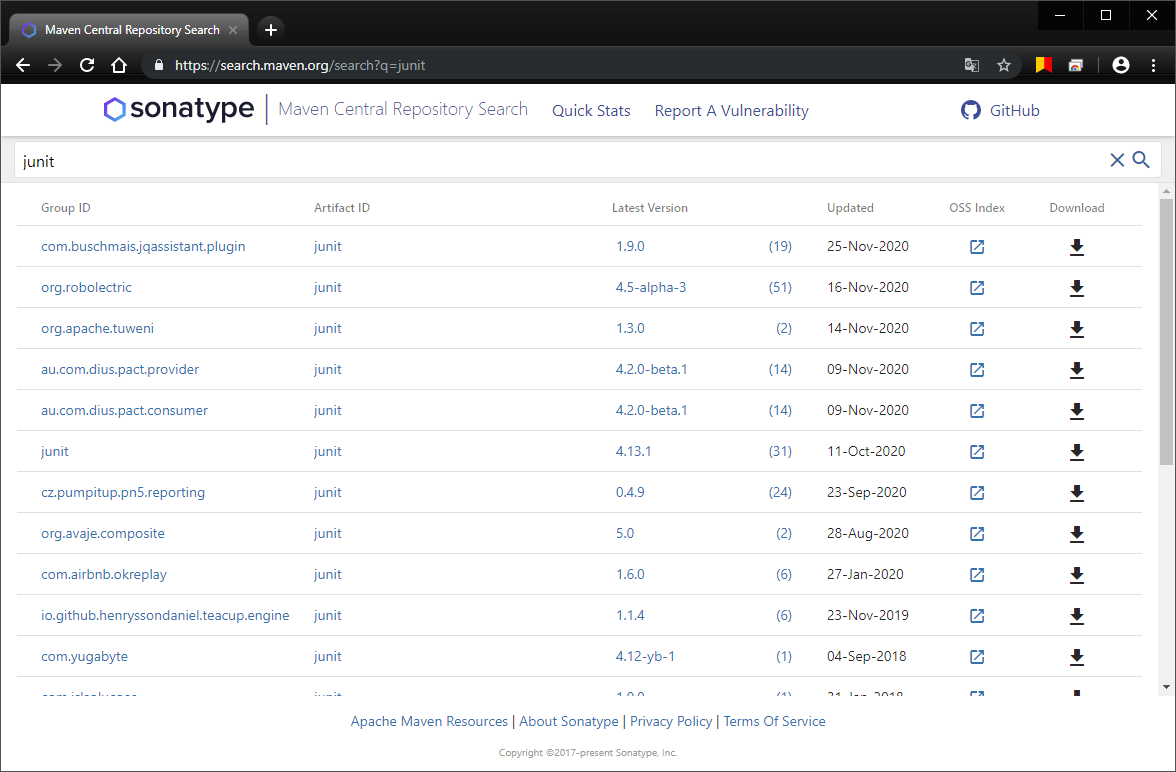


#### Центральный репозиторий

**Центральный репозиторий** поддерживается группой разработчиков Maven. Центральный репозиторий можно считать частным случаем удалённого репозитория. Центральный репозиторий находится по адресу <http://repo1.maven.org/maven2>. Но для просмотра и загрузки его содержимого через браузер следует использовать <http://search.maven.org/>. В этом репозитории лежат практически все опенсорсные фреймворки и библиотеки.

Самому в центральный репозиторий положить нельзя. Т.к. этот репозиторий используют все, то перед тем как туда попадают артефакты, они проверяются, тем более что если артефакт однажды попал в репозиторий, то по правилам изменить его нельзя.

Для поиска нужной библиотеки очень удобно пользоваться сайтами <http://mavenrepository.com/> и <http://findjar.com/>.



Если какая-либо зависимость не найдена в локальном репозитории, Maven проходит через центральный репозиторий. Затем Maven загружает эти зависимости в ваш локальный репозиторий.

#### Корпоративный репозиторий

**Корпоративный репозиторий** организуется для групповой работы над проектом. Сервер, на котором располагается корпоративный репозиторий называют production-сервером. Доступ к артефактам можно ограничивать настройками безопасности сервера так, что код ваших проектов не будет доступен извне.

Как правило, большинство популярных библиотек находятся в центральном репозитории, поэтому их можно прописывать сразу в раздел dependencies вашего pom-файла. Например чтобы подключить Spring Framework необходимо определить следующую зависимость:

<dependencies>

    ...

    <dependency>

        <groupId>org.springframework</groupId>

        <artifactId>spring</artifactId>

        <version>2.5.5</version>

    </dependency>

</dependencies>

Специфические вещи обычно не находятся в центральном репозитории и тогда вам придется указать репозиторий производителя вручную. Для примера добавим зависимость JSF-фреймворка ajax-компонентов JBoss RichFaces.

А вот репозиторий JBoss теперь необходимо прописать ручками либо в файле settings.xml, который лежит в корне вашего локального репозитория, либо в самом файле pom.xml, в зависимости от того, нужен ли данный репозиторий во всех проектах, либо в каком-то одном конкретном, соответственно:

*<!-- JBoss RichFaces Repository -->*

<repositories>

    <repository>

        <releases>

            <enabled>true</enabled>

        </releases>

        <snapshots>

            <enabled>false</enabled>

            <updatePolicy>never</updatePolicy>

        </snapshots>

        <id>repository.jboss.com</id>

        <name>Jboss Repository for Maven</name>

        <url>http://repository.jboss.com/maven2/</url>

        <layout>default</layout>

    </repository>

</repositories>

## Секция зависимостей <dependencies>

**Зависимости** – библиотеки, которые непосредственно используются в вашем проекте для компиляции кода или его тестирования.

Каждая библиотека идентифицируется так же, как и сам проект - тройкой groupId, artifactId, version (GAV). Как правило, большинство популярных библиотек находятся в центральном репозитории, поэтому их можно прописывать сразу в раздел dependencies вашего pom-файла.

### Область действия зависимости, scope

Кроме GAV при описании зависимости может присутствовать тег **<scope>**. Область действия scope определяет этап жизненного цикла проекта, в котором эта зависимость будет использоваться.

Существуют следующие области действия:

* test;

Если зависимость junit имеет область действия **test**, то эта зависимость будет использована maven'ом при выполнении компиляции той части проекта, которая содержит тесты, а также при запуске тестов на выполнение и построении отчета с результатами тестирования кода. Попытка сослаться на какой-либо класс или функцию библиотеки junit в основной части приложения (каталог src/main) вызовет ошибку.

* compile;

К наиболее часто используемой зависимости относится **compile** (используется по умолчанию). Т.е. dependency, помеченная как **compile**, или для которой не указано scope, будет доступна как для компиляции основного приложения и его тестов, так и на стадиях запуска основного приложения или тестов. Чтобы инициировать запуск тестов из управляемого maven-проекта можно выполнив команду "mvn test", а для запуска приложения используется плагин exec.

* provided;

Область действия зависимости **provided** аналогична compile, за исключением того, что артефакт используется на этапе компиляции и тестирования, а в сборку не включается. Предполагается, что среда исполнения (JDK или WEB-контейнер) предоставят данный артефакт во время выполнения программы. Наглядным примером подобных артефактов являются такие библиотеки, как hibernate или jsf, которые необходимы на этапе разработки приложения.

* runtime;

Область действия зависимости **runtime** не нужна для компиляции проекта и используется только на стадии выполнения приложения.

* system

Область действия зависимости **system** аналогична provided за исключением того, что содержащий зависимость артефакт указывается явно в виде абсолютного пути к файлу, определенному в теге systemPath. Обычно к таким артефактам относятся собственные наработки, и искать их в центральном репозитории, куда Вы его не размещали, не имеет смысла:

<dependencies>

    <dependency>

        <groupId>ru.carousel</groupId>

        <artifactId>carousel-lib</artifactId>

        <version>1.0</version>

        <scope>system</scope>

        <systemPath>/projects/libs/carousel-lib.jar</systemPath>

    </dependency>

</dependencies>

Как правило на сайтах крупных проектов пишут всю информацию, необходимую для встраивания их библиотеки в проект на основе Maven, но бывают случаи, когда артефакт приходится искать очень и очень долго. Здесь нам очень сильно может помочь <http://mvnrepository.com/> — он вам всегда подскажет где может находиться искомая библиотечка.

Бывает так, что проекты уже давно закрыты и в репозитории не положены потому что разработчики уже не заботятся об их распространении. Тогда остается один единственный способ – добавить файл в репозиторий вручную командой:

mvn install:install-file

    -Dfile=<path-to-file>

    -DgroupId=<group-id>

    -DartifactId=<artifact-id>

    -Dversion=<version>

    -Dpackaging=<packaging>

Последний параметр чаще всего имеет значение jar.

### Транзитивные зависимости

Начиная со второй версии фреймворка Maven были введены транзитивные зависимости, которые позволяет избегать необходимости изучения и определения библиотек, которые требуются для самой зависимости. Maven включает их автоматически.

В общем случае, все зависимости, используемые в проекте, наследуются от родителей. Ограничений по уровню наследований не существует, что, в свою очередь, может вызвать их сильный рост. В качестве примера можно рассмотреть создание проекта «A», который зависит от проекта «B». Но проект «B», в свою очередь, зависит от проекта «C». Подобная цепочка зависимостей может быть сколь угодно длинной. Как в этом случае поступает maven и как связан проект «A» и c проектом «C».

В следующей табличке, позаимствованной с сайта maven, представлен набор правил переноса области scope. К примеру, если scope артефакта «B» compile, а он, в свою очередь, подключает библиотеку «C» как provided, то наш проект «A» будет зависеть от «C» так как указано в ячейке находящейся на пересечении строки «compile» и столбца «provided».

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Compile | Provided | Runtime | Test |
| Compile | Compile | - | Runtime | - |
| Provided | Provided | Provided | Provided | - |
| Runtime | Runtime | - | Runtime | - |
| Test | Test | Test | Test | - |

## Секция репозиториев плагинов <pluginRepositories>

В теге **<pluginRepositories>** можно определить дополнительные репозитории плагинов проекта. По умолчанию включает информацию о репозитории central. Структура блока элемента pluginRepositories аналогична структуре элемента repositories. Каждый элемент pluginRepository указывает удаленное местоположение, где Maven может найти новые плагины.

***Пример:***

<pluginRepositories>

    <pluginRepository>

        <id>spb-maven-repo</id>

        <name>spb-maven-repo</name>

        <releases>

            <enabled>true</enabled>

            <updatePolicy>always</updatePolicy>

            <checksumPolicy>fail</checksumPolicy>

        </releases>

        <snapshots>

            <enabled>true</enabled>

            <updatePolicy>always</updatePolicy>

            <checksumPolicy>fail</checksumPolicy>

        </snapshots>

        <url>http://spb-maven-repo</url>

        <layout>default</layout>

    </pluginRepository>

</pluginRepositories>

## Секция сборки проекта <build>

Тег **<build>**  не обязательный, так как существуют значения по умолчанию.

<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">

*<!-- Секция сборки -->*

    <build>

        <finalName>projectName</finalName>

        <sourceDirectory>${basedir}/src/java</sourceDirectory>

        <outputDirectory>${basedir}/targetDir</outputDirectory>

        <resources>

            <resource>

                <directory>${basedir}/src/java/resources</directory>

                <includes>

                    <include>\*\*/\*.properties</include>

                </includes>

            </resource>

        </resources>

*<!-- Секция плагинов -->*

        <plugins>

        ...

        </plugins>

    </build>

</project>

Этот раздел содержит информацию по самой сборке:

* finalName – имя результирующего jar (war, ear ...) файла с соответствующим типу расширением, который создаётся на фазе package;

Значение по умолчанию — artifactId-version.

* sourceDirectory – директория, в которой расположены файлы исходного код;

По умолчанию это src/main/java, но вы можете определить, где это вам удобно. Директория может быть только одна (без использования специальных плагинов).

* outputDirectory – директория, в которую компилятор будет сохранять результаты компиляции - \*.class файлы;

Значение по умолчанию – target/classes.

* resourcesи вложенные в неё теги resource – одна или несколько директорий, где хранятся файлы ресурсов;

Ресурсы в отличие от файлов исходного кода при сборке просто копируются. Директория по умолчанию src/main/resources.

## Секция плагинов <plugins>

Maven базируется на plugin-архитектуре, которая позволяет применять плагины для различных задач (англ. compile, test, build, deploy, checkstyle, pmd, scp-transfer) для данного проекта, без необходимости их в явном виде инсталлировать.

**Плагин** – программный модуль, используемый Maven для выполнения действий над проектом, способ расширить функциональность Maven в больших диапазонах.

Иными словами, Maven запускает определенные плагины, которые выполняют всю работу. То есть, если мы хотим научить maven особенным сборкам проекта, то необходимо добавить в pom.xml указание на запуск нужного плагина в нужную фазу и с нужными параметрами.

Плагины хранятся в репозиториях плагинов которые как правило совпадают с репозиториями артефактов.

Плагины перечисляются и конфигурируются в секции **<plugins>** файла pom.xml. Некоторая базовая группа плагинов включается в каждый проект по умолчанию.

<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">

...

*<!-- Секция сборки -->*

    <build>

        ...

*<!-- Секция плагинов -->*

        <plugins>

        ...

        </plugins>

    </build>

    ...

</project>

Плагин является артефактом. Объявление плагина в проекте похоже на объявление зависимости. Плагины также идентифицируется с помощью GAV (groupId, artifactId, version).

***Пример:***

<plugins>

    <plugin>

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

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

        <version>2.6</version>

    </plugin>

</plugins>

Объявление плагина в pom.xml позволяет зафиксировать версию плагина, задать ему необходимые параметры, привязать к фазам.

В проектном файле pom.xml можно настроить для каждого из плагинов жизненного цикла набор конфигурационных переменных в **<configuration>.**

***Пример:***

<build>

    <plugins>

        <plugin>

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

            <configuration>

                <verbose>true</verbose>

                <executable>C:/Program\_Files/Java/jdk1.7.0\_67/bin/javac.exe</executable>

                <source>1.7</source>

                <target>1.7</target>

            </configuration>

        </plugin>

    </plugins>

</build>

### Запуск плагина

В каждой фазе жизненного цикла проекта вызывается определенный плагин (jar-библиотека), который включает некоторое количество целей (goal). Например, плагин maven-compiler-plugin содержит две цели: compiler:compile для компиляции основного исходного кода проекта и compiler:testCompile.

После того как плагин объявлен, его можно настроить так, чтобы он автоматически запускался в нужный момент. Это делается с помощью привязки плагина к фазе сборки проекта.

***Пример:***

<plugin>

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

    <artifactId>имя-плагина</artifactId>

    <executions>

        <execution>

            <id>customTask</id>

            <phase>generate-sources</phase>

            <goals>

                <goal>pluginGoal</goal>

            </goals>

...

Самое важное в данном случае – это определить для плагина наименование фазы execution/phase в которую нужно встроить вызов цели плагина goal.

В простейшем случае запустить плагин просто – для этого необходимо выполнить команду в определенном формате. Плагин обеспечивает достижение ряда целей с помощью следующего синтаксиса:

mvn [имя плагина]:[имя цели]

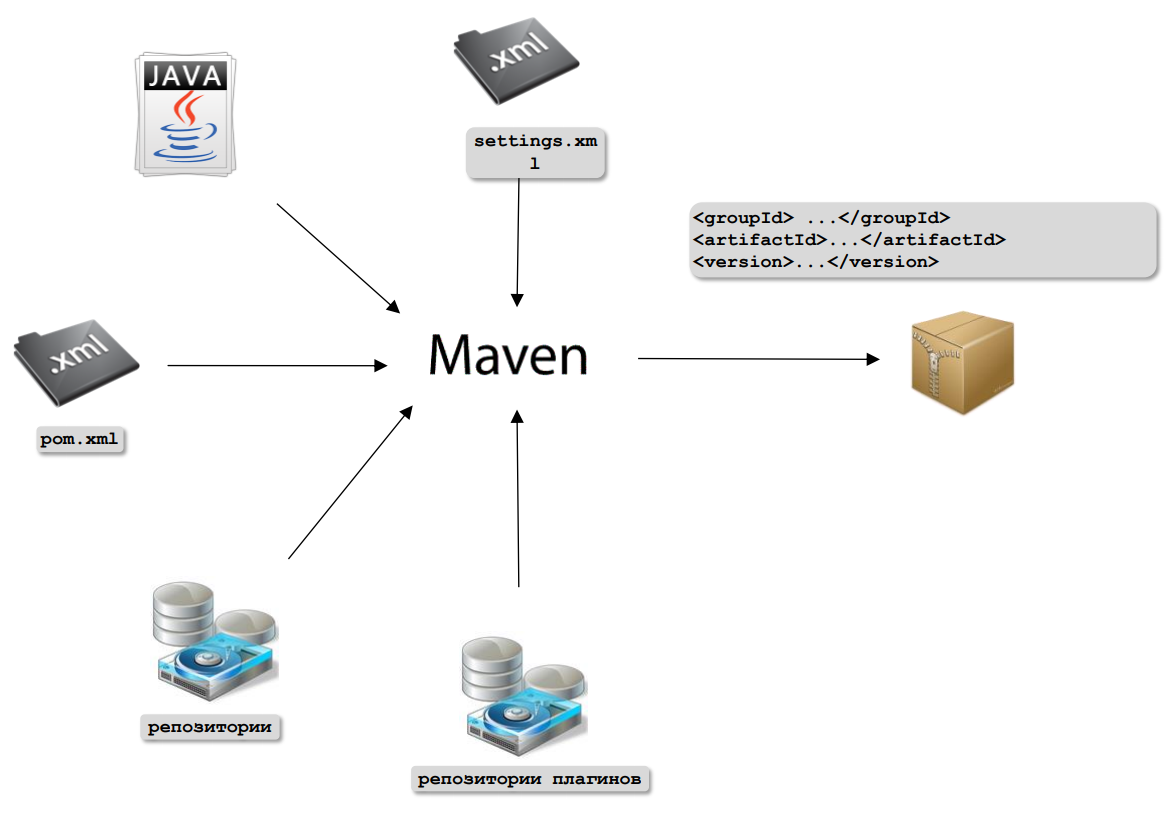
Например, чтобы запустить плагин maven-checkstyle-plugin (artifactId) с groupId равным org.apache.maven.plugins необходимо выполнить следующую команду :

mvn org.apache.maven.plugins:maven-checkstyle-plugin:check

Можно запустить в более краткой форме:

mvn maven-checkstyle-plugin:check

Целью (goal) выполнения данного плагина является проверка check.



### Полезные плагины

Количество доступных плагинов в настоящее время очень велико и включает, в том числе, плагины, позволяющие непосредственно из Maven запускать web-приложение для тестирования его в браузере; плагины, позволяющие тестировать или создавать банки данных; плагины, позволяющие генерировать Web Services.

Задачей разработчика в такой ситуации является найти и применить наиболее подходящий набор плагинов.

#### Плагин указания версии компилятора Java

Плагин, который позволяет указать версию компилятора Java.

***Пример:***

<plugin>

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

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

    <version>3.1</version>

    <configuration>

        <source>1.7</source>

        <target>1.7</target>

    </configuration>

</plugin>

#### Плагин указания исполняемого класса в библиотеке

Если один из ваших проектов собирается в Java-архив, который вы предполагаете запускать с помощью команды java -jar <имя файла Java-архива>, то такой архив, в своем файле манифеста должен иметь указание на класс со статической функцией main() используемой для запуска Java-приложений.

Плагин с указанием запускаемого класса com.visit7.test-1.MyApp.

***Пример:***

<plugin>

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

    <artifactId>maven-jar-plugin</artifactId>

    <configuration>

        <archive>

            <manifest>

                <mainClass>com.visit7.test-1.MyApp</mainClass>

            </manifest>

        </archive>

    </configuration>

</plugin>

#### Плагин maven-checkstyle-plugin

Плагин проверяет стиль и качество исходного кода и генерирует файл checkstyle-result.xml. Из наиболее часто используемых и простых проверок:

* наличие комментариев;
* размер класса не более N строк;
* в конструкции в try-catch, блок catch не пуст.

***Пример:***

<plugin>

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

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

     <version>2.17</version>

     <configuration>

         <suppressionsLocation>suppressions.xml</suppressionsLocation>

     </configuration>

 </plugin>

Запуск с помощью команды:

mvn checkstyle:check

Т.к. почти каждые проекты пишутся немного по-разному, рекомендуется создать свой набор правил. Или подавить проверку некоторых правил. Для этого используется файл suppressions.xml.

***Пример:***

<?xml*version*="1.0"?>

<!DOCTYPE suppressions PUBLIC "-//Puppy Crawl//DTD Suppressions 1.1//EN"

        "http://www.puppycrawl.com/dtds/suppressions\_1\_1.dtd">

<suppressions>

    <suppress *files*="." *checks*="HideUtilityClassConstructorCheck"/>

    <suppress *files*="." *checks*="JavadocPackageCheck"/>

    <suppress *files*="." *checks*="MagicNumberCheck"/>

    <suppress *files*="." *checks*="JavadocMethodCheck"/>

    <suppress *files*="." *checks*="FinalParametersCheck"/>

    <suppress *files*="." *checks*="JavadocTypeCheck"/>

    <suppress *files*="." *checks*="JavadocVariableCheck"/>

    <suppress *files*="." *checks*="DesignForExtensionCheck"/>

    <suppress *files*="." *checks*="HiddenFieldCheck"/>

    <suppress *files*="." *checks*="AvoidInlineConditionalsCheck"/>

    <suppress *files*="NoBody.java" *checks*="NeedBracesCheck"/>

    <suppress *files*="NoBody.java" *checks*="EmptyStatementCheck"/>

    <suppress *files*="inheritance[\\/](A|C|D|Figure|Figure2|M|N).java" *checks*="VisibilityModifierCheck" />

    <suppress *files*="oop[\\/](Box[1-6]|HeavyBox|HeavyBox1|StaticVars|Test4|UseStatic|ColorBox|Shipment|Test2|Test3|Toy).java"

*checks*="VisibilityModifierCheck" />

    <suppress *files*="introduction[\\/]VarDemo.java" *checks*="VisibilityModifierCheck" />

    <suppress *files*="introduction[\\/]VarDemo.java" *checks*="AvoidNestedBlocksCheck" />

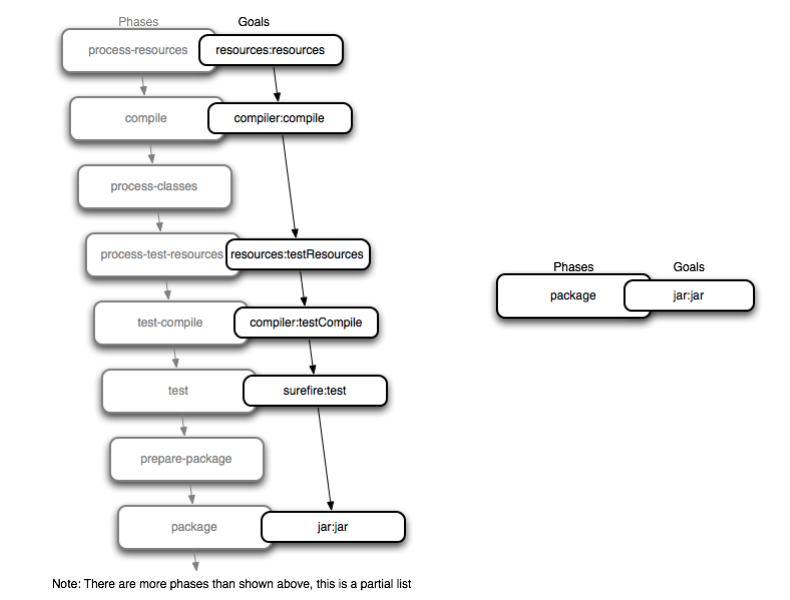
    <suppress *files*="enumerations[\\/](Coffee|Coffee3|Coffee2).java" *checks*="VisibilityModifierCheck" />

</suppressions>

# Жизненный цикл сборки

**Цикл сборки** – чётко определённая последовательность фаз.

У фазы могут быть зарегистрированы задачи. Собрать проект - пройти через последовательность фаз и для каждой фазы выполнить задачи зарегистрированные у этой фазы. По умолчанию у некоторых фаз уже есть зарегистрированные задачи. Кроме того, используя POM файл можно зарегистрировать новые задачи у фазы или изменить конфигурацию задач.



## Фазы

**Фаза** – набор действий, который связан с 2 или 3 целями.

Жизненный цикл Maven содержит три независимых порядка выполнения

* clean — жизненный цикл для очистки проекта.

Содержит следующие фазы:

1. pre-clean;
2. clean;
3. post-clean;

* default — основной жизненный цикл.

Содержит следующие фазы:

1. ***validate*** — проверка корректности проекта и обеспечение доступности необходимых зависимостей;
2. initialize — подготовка окружения сборки, вычисление значения переменных, создание необходимых директории и так далее;
3. generate-sources – генерация исходного кода для компиляции с кодом проекта;
4. process-sources – обработка исходного кода проекта (включая сгенерированный ранее) перед компиляцией;
5. generate-resources – генерация ресурсов проекта;
6. process-resources – обработка ресурсов проекта (включая сгенерированные) перед компиляцией;
7. ***compile*** – компиляция исходных файлов проекта;
8. process-classes – дополнительная обработка скомпилированных классов;
9. generate-test-sources — генерация исходного кода для компиляции совместно с тестами;
10. process-test-sources – обработка исходного кода тестов проекта (включая сгенерированный ранее) перед компиляцией;
11. generate-test-resources – генерация тестовых ресурсов проекта;
12. process-test-resources – обработка тестовых ресурсов проекта (включая сгенерированные) перед компиляцией;
13. test-compile – компиляция исходных кодов тестов;
14. process-test-classes – дополнительная обработка скомпилированных классов тестов;
15. ***test*** – проверка скомпилированных файлов (запуск юнит тестов);
16. prepare-package – выполнение процедур по подготовке к упаковке в пакет (обычно в этой фазе полностью формируется содержимое будущего пакета);
17. ***package*** – упаковка скомпилированных классов и прочих ресурсов (создание jar файла или war, ear в зависимости от типа проекта);
18. pre-integration-test – подготовка окружения к запуску интеграционных тестов;
19. integration-test – проверка взаимодействия между составными частями программного продукта (запуск интеграционных тестов);
20. post-integration-test – очистка окружения после интеграционных тестов;
21. ***verify*** – проверка для подтверждения готовности упакованного файла;
22. ***install*** – копирование jar (war, ear) в локальный репозиторий, чтобы сделать его доступным для других проектов текущего пользователя;
23. ***deploy*** – публикация файла в удалённый репозиторий, стабильная версия программного обеспечения распространяется на удаленный Maven-репозиторий, чтобы сделать его доступным для других пользователей

* site – жизненный цикл генерации проектной документации.

Содержит следующие фазы:

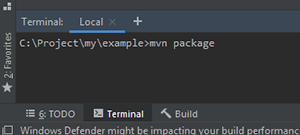
1. pre-site
2. site
3. post-site
4. site-deploy

### Выполнение фаз

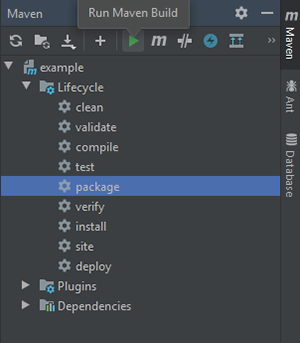
Выполнить запуск фазы можно двумя способами:

* через командную строку:

mvn package



* используя Intellij IDEA:



Все фазы выполняются последовательно: нельзя запустить, скажем, четвертую фазы, пока не завершены фазы 1-3. К примеру, нам нужно создать jar проекта. Чтобы его создать набираем:

mvn package

Но перед созданием jar-файла будут выполняться все предыдущие фазы compile и test, а фазы integration-test, install, deploy не выполнятся.

Если набрать,

mvn deploy

то выполнятся все приведённые выше фазы.

Особняком стоят фазы clean и site. Они не выполняются, если специально не указаны в строке запуска:

* clean – удаление всех созданных в процессе сборки артефактов: .class, .jar и др. файлов. В простейшем случае результат — просто удаление каталога target.
* site – предназначена для создания документации.

Также у каждой фазы есть пре- и пост-фазы: например, pre-deploy, post-deploy, pre-clean, post-clean, но используются они довольно редко.

Вызывать вручную можно все фазы, но, обычно, фазы с ‘-‘ в названии не принято вызывать явно, так как они могут оставить сборку в неоконченном виде. Например, вызвав integration-test вместо verify вы останетесь с неубранным мусором от интеграционных тестов.

Фазы можно перечислить через пробел. Т.к. команда mvn понимает, когда передают несколько фаз, то для сборки проекта создания документации с нуля выполняют:

mvn clean package site

## Цели

**Задача или цель (goal)** – специальная задача, которая относится к сборке проекта и его управлению. Она может привязываться как к нескольким фазам, так и ни к одной. Задача, которая не привязана ни к одной фазе, может быть запущена вне фаз сборки с помощью прямого вызова.

Несмотря на всё вышесказанное о предназначении фаз, на самом деле они не делают ничего. А в списке перечислено, для чего эти фазы проектировались и не более того. Всю настоящую работу выполняют те плагины, цели (goal) которых привязаны к фазам сборочного цикла.

Стандартные цели заложены по умолчанию, дополнительные добавляются Maven-плагинами. Плагины позволяют вставлять в стандартный цикл новые шаги (например, распределение на сервер приложений) или расширять существующие шаги. То есть Maven плагины позволяют задать дополнительные действия, которые будут выполняться при сборке.

Использование Maven часто сводится к выполнению одной из целей:

* validate – проверка корректности метаинформации о проекте;
* compile – компилиляция исходников;
* test – прогонка тестов классов;
* package – упаковка скомпилированнных классов в заданный формат (jar или war, к примеру);
* integration-test – отправка упакованных классов в среду интеграционного тестирования и прогонка тестов;
* verify – проверка корректности пакета и удовлетворение требованиям качества;
* install – отправка пакета в локальный репозиторий, где он будет доступен для использования как зависимость в других проектах;
* deploy – отправка пакета на удаленный production сервер, где доступ к нему будет открыт другим разработчикам.

### Выполнение целей

В общем случае для выполнения команды Maven необходимо выполнить следующий код:

mvn цель

В качестве параметров указываются не только имена фаз, но и имена и цели плагинов в формате

mvn [имя плагина]:[имя цели]

Например, вызов фазы цикла mvn clean эквивалентен вызову плагина mvn clean:clean.

Порядок выполнения команд Maven проекта зависит от порядка вызова целей и фаз. Следующая команда

mvn clean dependency:copy-dependencies package

выполнит фазу clean, после этого будет выполнена задача dependency:copy-dependencies, после чего будет выполнена фаза package. Аргументы clean и package являются фазами сборки, dependency:copy-dependencies является задачей.

## Секция отчетов <reporting>

Секция **<reporting>** содержит настройку плагинов для генерации отчетов.

Некоторые плагины Maven могут создавать отчеты, определенные и настраиваемые в элементе **<reporting>**, например: создание отчетов Javadoc.

<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">

    ...

*<!-- Секция отчетов -->*

    <reporting>

    ...

    </reporting>

    ...

</project>

***Пример:***

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

    ...

    <reporting>

        <outputDirectory>${basedir}/target/site</outputDirectory>

        <plugins>

            <plugin>

                <artifactId>maven-project-info-reports-plugin</artifactId>

                <version>2.0.1</version>

                <reportSets>

                    <reportSet></reportSet>

                </reportSets>

            </plugin>

        </plugins>

    </reporting>

    ...

</project>

# Конфигурационные файлы

**Конфигурационные файлы** содержат информацию применимую ко многим проектам и которую не следует распространять вместе с проектами. В них содержится информация о локальном репозитории, прокси-серверах, зеркалах, информация для аутентификации.

Конфигурационные файлы находятся в директории установки Maven **$M2\_HOME/conf/settings.xml** и в домашней директории пользователя **~/.m2/settings.xml**.

## Структура конфигурационного файла

Конфигурационные файлы имеют следующую структуру:

<settings>

    <localRepository/>

    <interactiveMode/>

    <usePluginRegistry/>

    <offline/>

    <pluginGroups/>

    <servers/>

    <mirrors/>

    <proxies/>

    <profiles/>

    <activeProfiles/>

</settings>

### Простые элементы верхнего уровня

Простые элементы верхнего уровня:

* localRepository – путь к локальному репозиторию;

Значение по умолчанию ${user.dir}/.m2/repository.

* interactiveMode – интерактивный режим;

Если Maven должен пытаться взаимодействовать с пользователем, должно быть true в противном случае false. Значение по умолчанию true.

* usePluginRegistry – параметр регистра плагинов;

Если Maven должен использовать ${user.dir}/.m2/plugin-registry.xml для управления версиями плагинов должно быть true. Значение по умолчанию false.

* offline – параметр офлайн;

Если Maven должен работать в отсоединённом состоянии должно быть true. Значение по умолчанию false.

* pluginGroups – список групп плагинов;

***Пример:***

<settings>

    <localRepository/>${user.dir}/.m2/repository</localRepository>

    <interactiveMode/>false</interactiveMode>

    <usePluginRegistry/>false</usePluginRegistry>

    <offline/>false</offline>

    <pluginGroups>

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

        <pluginGroup>org.codehaus.mojo</pluginGroup>

    </pluginGroups>

...

</settings>

Этот элемент содержит список элементов pluginGroup, каждый содержит groupId. Этот список будет использован при вызове Maven с командной строки без указания groupId. По умолчанию этот список включает org.apache.maven.plugins.

### Серверы

Элемент **<server>** содержит информацию для репозиториев и зеркал такую как имена пользователей и пароли. Элемент id это идентификатор репозитория или зеркала.

***Пример:***

<servers>

    <server>

        <id>server001</id>

        <username>my\_login</username>

        <password>my\_password</password>

        <privateKey>${user.home}/.ssh/id\_dsa</privateKey>

        <passphrase>some\_passphrase</passphrase>

        <filePermissions>664</filePermissions>

        <directoryPermissions>775</directoryPermissions>

        <configuration></configuration>

    </server>

</servers>

### Зеркала

Элемент **<mirror>** позволяет задавать зеркала для репозиториев. Элемент **<mirrorOf>** включает идентификатор репозитория. В идентификаторах репозиториев можно использовать \* и !.

***Пример:***

<mirrors>

    <mirror>

        <id>planetmirror.com</id>

        <name>PlanetMirror Australia</name>

        <url>http://downloads.planetmirror.com/pub/maven2</url>

        <mirrorOf>central</mirrorOf>

    </mirror>

</mirrors>

### Прокси-серверы

Элемент **<proxy>** позволяет задавать прокси сервер который может использоваться для взаимодействия с репозиториями и зеркалами репозиториев. Можно задать информацию для аутентификации и серсеры для взаимодействия с которыми прокси сервер использовать не следует.

***Пример:***

<proxies>

    <proxy>

        <id>myproxy</id>

        <active>true</active>

        <protocol>http</protocol>

        <host>proxy.somewhere.com</host>

        <port>8080</port>

        <username>proxyuser</username>

        <password>somepassword</password>

        <nonProxyHosts>\*.google.com|ibiblio.org</nonProxyHosts>

    </proxy>

</proxies>

# Создание проекта

## Архетипы

**Архетип** – шаблон для создания проекта, некая стандартная компоновка файлов и каталогов в проектах различного рода (веб, swing-проекты и прочие). Другими словами, Maven знает, как обычно строятся проекты и в соответствии с архетипом создает структуру каталогов.

Как правило, название артефакта состоит из названия группы, собственного названия и версии. К примеру Spring будет иметь вот такое название в среде Maven: org.springframework.spring:2.5.5. Последний домен означает всегда artifactId, все, что перед ним – groupId – хорошо это запомните!

## Параметры при создании проекта

Пример команды на создание проекта из архетипа:

mvn archetype:generate

    -DgroupId=com.mycompany.app

    -DartifactId=my-app

    -DarchetypeArtifactId=maven-archetype-quickstart

    -DarchetypeVersion=1.4

    -DinteractiveMode=false

Команда должна быть написана в одну строчку. Здесь, она разнесена по строкам для удобства чтения.

В команде используются следующие параметры:

* groupId – идентификатор группы создаваемого проекта.
* artifactId – идентификатор артефакта создаваемого проекта.

Maven оперирует так называемыми артефактами. Это приложения, плагины, архетипы и другие проекты. И ваша разработка также будет считаться артефактом.

* package – тип пакетирования создаваемого проекта.
* version – версия создаваемого проекта.
* archetypeGroupId – идентификатор группы архетипа создаваемого проекта.
* archetypeArtifactId – идентификатор артефакта архетипа создаваемого проекта.

Указанный архетип создаст простой проект, сгенерирует структуру каталогов

* archetypeVersion – версия артефакта «архетип».
* interactiveMode – интерактивный режим

Если Maven должен пытаться взаимодействовать с пользователем, должно быть

true в противном случае false. Значение по умолчанию true.

Если вы укажете все эти определения (-D), то выполнение цели archetype:generate запросит вас только подтвердить введенные данные. Для подтвержения введите Y и нажмите Enter, или просто нажмите Enter.

В результате выполнения команды создастся директория с именем артефакта внутри которой будет находится проектный файл pom.xml и начальная структура поддиректорий, характерная для проектов Maven.

***Пример сгенерированного файла pom.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

        http://maven.apache.org/maven-v4\_0\_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>ru.knzsoft</groupId>

    <artifactId>test-1</artifactId>

    <packaging>jar</packaging>

    <version>1.2.3</version>

    <name>test-1</name>

    <url>http://maven.apache.org</url>

    <dependencies>

        <dependency>

            <groupId>junit</groupId>

            <artifactId>junit</artifactId>

            <version>3.8.1</version>

            <scope>test</scope>

        </dependency>

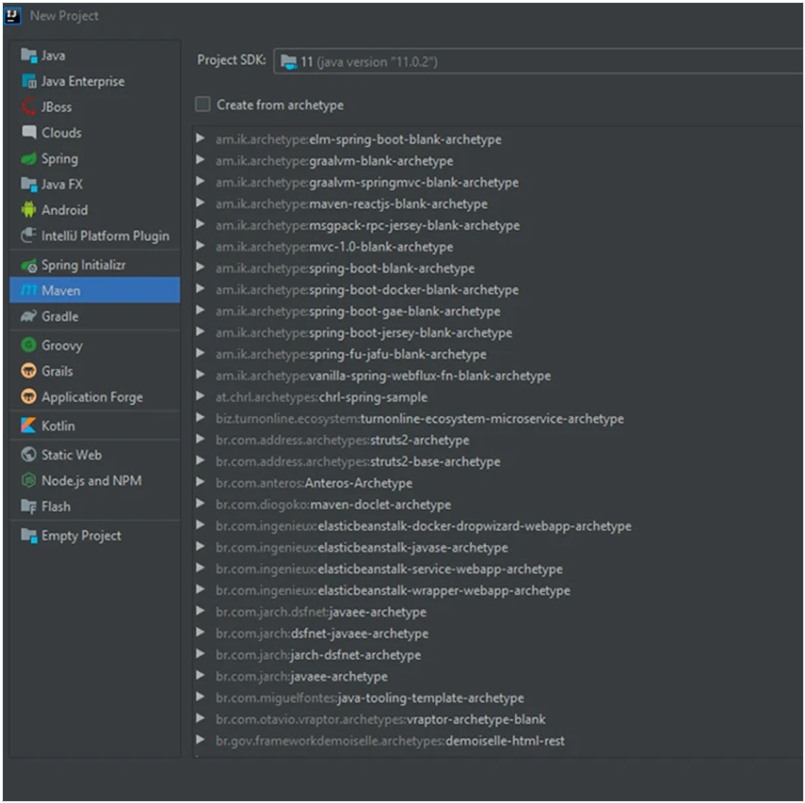
    </dependencies>

</project>

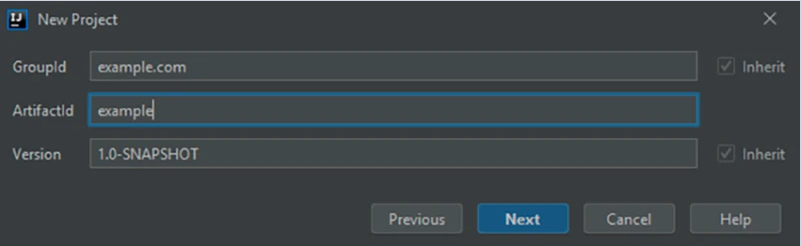
## Создание проекта Maven в IntelliJIDEA

Для того, чтобы создать проект Maven нужно выполнить следующие шаги:

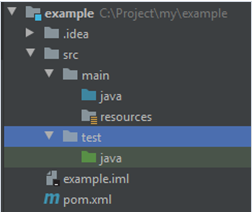
1. Открыть IntelliJIDEA и создать новый проект.
2. Выбрать тип проекта – Maven.



1. Нажать Next и заполнить поля на диалоговом окне.



1. Далее стандартно создать проект в необходимом месте.

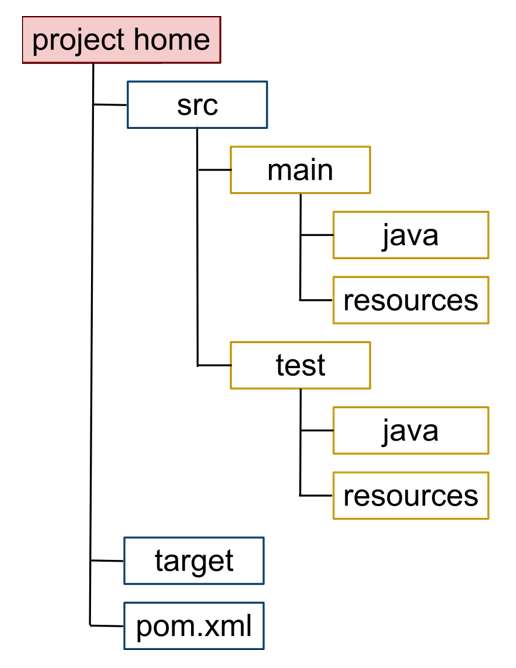


После того, как проект создался, обрати внимание на его структуру. Это стандартная структура для Maven-проекта.

## Стандартная структура для Maven-проекта

**Стандартная структура каталогов** — одна из реализаций принципа архетипов в Maven.

Следующая структура показывает важнейшие каталоги для проекта на Java:



Корневой каталог проекта:

* **файл pom.xml;**
* **src**: все исходные файлы;
  + **src/main**: исходные файлы собственно для продукта;
    - **src/main/java**: исходный текст на Java;
    - **src/main/resources**: другие файлы, которые используются при компиляции или исполнении, например properties-файлы;
  + **src/test**: исходные файлы, необходимые для организации автоматического тестирования;
    - **src/test/java**: JUnit-тест-задания для автоматического тестирования;
    - **src/test/resources**: другие файлы, которые используются при компиляции или исполнении, например properties-файлы;
* **target**: все создаваемые в процессе работы Мавена файлы;
  + **target/classes**: компилированные Java-классы;