Інструменти збірки на Java. Основи Maven 2

**Системы сборки проектов**

Мавен - это инструмент для сборки Java проекта: компиляции, создания jar, создания дистрибутива программы, генерации документации. Простые проекты можно собрать в командной строке. Если собирать большие проекты с командной строки, то команда для сборки будет очень длинной, поэтому её иногда записывают в bat/sh скрипт. Но такие скрипты зависят от платформы. Для того чтобы избавиться от этой зависимости и упростить написание скрипта используют инструменты для сборки проекта.

Для платформы Java существуют два основных инструмента для сборки: [Ant](http://ant.apache.org/) и [Maven](http://maven.apache.org/).

**Основные преимущества Maven**

* **Независимость от OS.** Сборка проекта происходит в любой операционной системе. Файл проекта один и тот же.
* **Управление зависимостями.** Редко какие проекты пишутся без использования сторонних библиотек(зависимостей). Эти сторонние библиотеки зачастую тоже в свою очередь используют библиотеки разных версий. Мавен позволяет управлять такими сложными зависимостями. Что позволяет разрешать конфликты версий и в случае необходимости легко переходить на новые версии библиотек.
* **Возможна сборка из командной строки.** Такое часто необходимо для автоматической сборки проекта на сервере ([Continuous Integration](http://www.apache-maven.ru/continuous-integration.html)).
* **Хорошая интеграция со средами разработки.**Основные среды разработки на java легко открывают проекты которые собираются c помощью maven. При этом зачастую проект настраивать не нужно - он сразу готов к дальнейшей разработке.   
  Как следствие - если с проектом работают в разных средах разработки, то maven удобный способ хранения настроек. Настроечный файл среды разработки и для сборки один и тот же - меньше дублирования данных и соответственно ошибок.
* **Декларативное описание проекта.**

**Установка** **Maven**

 Зайдите на официальный сайт мавен в раздел [загрузка](http://maven.apache.org/download.html) и скачайте последнюю стабильную версию.

 Распакуйте архив в инсталляционную директорию. Например в C:\Program Files\maven\ в Windows или /opt/maven в Linux

 Установите переменную окружения M2\_HOME:   
В Windows кликните правой кнопкой мыши на "мой компьютер" ->свойства->дополнительные параметры->переменные среды->системные переменные и там добавьте "M2\_HOME" и "C:\Program Files\maven\" .   
В Linux можно добавить строку "export M2\_HOME=/opt/maven"в файл /etc/profile .

 Установите переменную окружения PATH В Windows в переменной PATH добавьте к списку директорий строку %M2\_HOME%\bin". В Linux можно добавить строку "export PATH=$PATH:$M2\_HOME/bin"в файл /etc/profile .

 Проверьте корректность установки, набрав в командной строке

mvn -version

Если результат примерно такой

dima@myhost ~ $ mvn -version

Apache Maven 3.0 (r1004208; 2010-10-04 15:50:56+0400)

Java version: 1.6.0\_22

Java home: /opt/sun-jdk-1.6.0.22/jre

Default locale: ru\_RU, platform encoding: UTF-8

OS name: "linux" version: "2.6.34-gentoo-r12" arch: "amd64" Family: "unix"

то поздравляю, вы успешно установили Maven.

\*Во многих дистрибутивах Linux, maven устанавливается автоматически, с помощью менеджера пакетов.

**Если что-то не работает**

* Проверьте, установлен ли у вас JDK.

Для этого наберите в консоли «java -version» ответ должен быть примерно таким:

java version "1.6.0\_22"

OpenJDK Runtime Environment (IcedTea6 1.10.5) (ArchLinux-6.b22\_1.10.5-1-x86\_64)

OpenJDK 64-Bit Server VM (build 19.0-b09, mixed mode)

* Проверьте установлена ли переменная окружения JAVA\_HOME

Если у вас Windows наберите в консоли:

echo %JAVA\_HOME%

Если у вас Linux наберите в консоли:

echo $JAVA\_HOME

команда должна вывести путь к JDK.

## Простой пример

Давайте создадим новый проект. Для этого выполним команду:

mvn archetype:generate

Выполнив эту команду, maven покажет список шаблонов(архетипов) для разных проектов. Выберите проект и его версию по умолчанию, нажав Enter, Enter. Далее команда спросит *groupId* и *artifactId*. Введите данные:   
*ru.apache\_maven*   
*testproject1*

В результате сгенерируется проект со стандартной структурой директорий

testproject1

|-- pom.xml

`-- src

|-- main

| `-- java

| `--ru

| `-- apache\_maven

| `-- App.java

`-- test

`-- java

`-- ru

`-- apache\_maven

`-- AppTest.java

с примером приложения и с JUnit тестом для этого приложения. pom.xml - это как раз файл для maven. Его мы будем рассматривать отдельно [более подробно](http://www.apache-maven.ru/project-file.html).

Чтобы скомпилировать, нужно перейти в директорию проекта testproject1 и набрать в консоли mvn compile. Если в консоль выведется

...

[INFO] BUILD SUCCESS

.....

то компиляция прошла успешно и в созданной директории target/classes будут class файлы c нашей программой.

Если вы наберёте mvn package, в директории target будет создан jar файл testproject1-1.0-SNAPSHOT.jar

Давайте запустим скомпилированную нами программу

[dima@myhost testproject1]$ java -cp ./target/classes ru.apache\_maven.App

Hello World!

Поздравляю, программа работает!

**Интеграция**

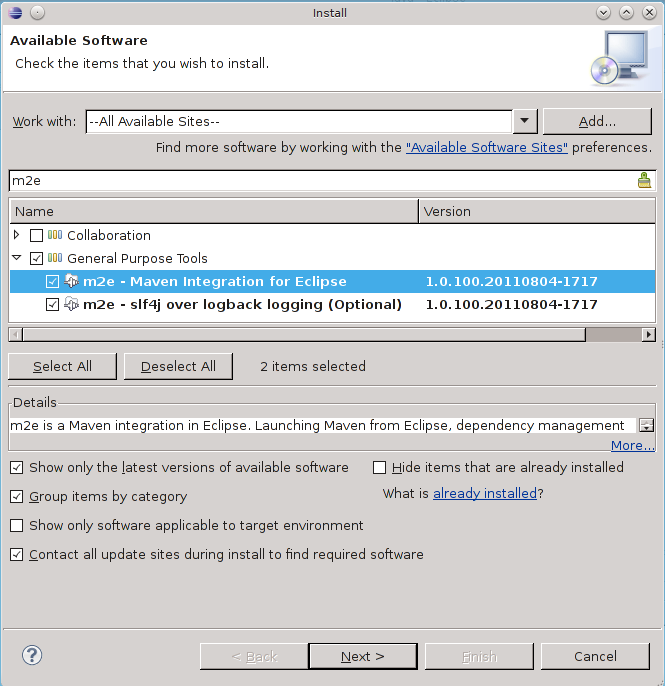
Хочу обратить внимание, что полученный проект можно сразу открывать средой разработки:

* для среды IntelliJ Idea проект открывается сразу ("File/OpenProject")
* для среды NetBeans проект открывается сразу ("Файл/Создать проект/Проект maven с существующим POM")
* для eclipse нужно установить соответствующий плагин.

**Установка плагина в eclipse**

Для установки

1. зайдите в меню Help->Install New Software
2. Выберите Work with --All Available Sites--
3. в фильтре наберите m2e и отметьте плагины как показано на рисунке



После установки плагина можно:

* Создавать проекты через File->New Progect->Maven->Maven Project
* Открыть существующий проект File->Import->Maven->Existing Maven Projects

**Что такое pom.xml**

pom.xml - это основной файл, который описывает проект. Вообще могут быть дополнительные файлы, но они играют второстепенную роль.

Давайте разберём из чего состоит файл pom.xml

**Корневой элемент и заголовок.**

<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>

Корневой элемент <project>, схема, которая облегчает редактирование и проверку, и версия POM.

Внутри тэга project содержится основная и обязательная информация о проекте:

<!-- The Basics -->

<groupId>...</groupId>

<artifactId>...</artifactId>

<version>...</version>

В Maven каждый проект идентифицируется парой groupId artifactId. Во избежание конфликта имён, groupId - наименование организации или подразделения и обычно действуют такие же правила как и при именовании пакетов в Java - записывают доменное имя организации или сайта проекта. artifactId - название проекта. Внутри тэга version, как можно догадаться хранится версия проекта. Тройкой groupId, artifactId, version (далее - GAV) можно однозначно идентифицировать jar файл приложения или библиотеки. Если состояние кода для проекта не зафиксировано, то в конце к имени версии добавляется "-SNAPSHOT" что обозначает, что версия в разработке и результирующий jar файл может меняться. <packaging>...</packaging> определяет какого типа файл будет создаваться как результат сборки. Возможные варианты pom, jar, war, ear

Давайте лучше рассмотрим на примере проекта [powermock-core](http://mirrors.ibiblio.org/pub/mirrors/maven2/org/powermock/powermock-core/1.4.6/powermock-core-1.4.6.pom) **groupId** - org.powermock, **artifactId** - powermock-core , **version** - 1.4.6

Также добавляется информация, которая не используется самим мавеном, но нужна для программиста, чтобы понять, о чём этот проект:

* <name>powermock-core</name> название проекта для человека
* <description>PowerMock core functionality.</description> Описание проекта
* <url>[http://www.powermock.org](http://www.powermock.org/)</url> сайт проекта.

**Зависимости**

Зависимости - следующая очень важная часть pom.xml - тут хранится список всех библиотек (зависимостей) которые используюся в проекте. Каждая библиотека идентифицируется также как и сам проект - тройкой groupId, artifactId, version (GAV). Объявление зависимостей заключено в тэг <dependencies>...</dependencies>.

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.4</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.powermock</groupId>

<artifactId>powermock-reflect</artifactId>

<version>${version}</version>

</dependency>

<dependency>

<groupId>org.javassist</groupId>

<artifactId>javassist</artifactId>

<version>3.13.0-GA</version>

<scope>compile</scope>

</dependency>

</dependencies>

Как вы могли заметить, кроме GAV при описании зависимости может присутствовать тэг <scope>. Scope задаёт, для чего библиотека используется. В данном примере говорится, что библиотека с GAV junit:junit:4.4 нужна только для выполнения тестов.

**Тэг <build>**

Тэг <build> не обязательный, т. к. существуют значения по умолчанию. Этот раздел содержит информацию по самой сборке: где находятся исходные файлы, где ресурсы, какие плагины используются. Например:

<build>

<outputDirectory>target2</outputDirectory>

<finalName>ROOT</finalName>

<sourceDirectory>src/java</sourceDirectory>

<resources>

<resource>

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

<includes>

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

</includes>

</resource>

</resources>

<plugins>

<plugin>

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

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

<version>2.4</version>

</plugin>

</plugins>

</build>

Давайте рассмотрим этот пример более подробно.

* <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

* <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.

Maven плагины позволяют задать дополнительные действия, которые будут выполняться при сборке. Например в приведённом примере добавлен плагин, который автоматически делает проверку кода на наличие "плохого" кода и потенциальных ошибок. См [ссылка](http://www.apache-maven.ru/plugins/maven-pmd-plugin.html).

Читайте более подробнее [о плагинах](http://www.apache-maven.ru/plugins/index.html)

Требования к сборке сильно зависят от проекта. Плагины - это способ расширить функциональность maven в больших диапазонах.

**Использование плагина**

В простейшем случае запустить плагин просто, например:

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

В данном примере вызывается плагин с

* groupId "org.apache.maven.plugins"
* artifactId "maven-checkstyle-plugin"
* последней версией
* и целью (goal) "check"

Цель - это действие, которое плагин может выполнить. Целей может быть несколько.

плагины с groupId "org.apache.maven.plugins" можно запустить в более краткой форме:

mvn maven-checkstyle-plugin:check

или даже так:

mvn checkstyle:check

**Объявление плагина в pom.xml**

Объявление плагина похоже на объявление зависимости. Также, как и зависимости плагины идентифицируется с помощью GAV(groupId,artifactId,version). Например:

<plugin>

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

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

<version>2.6</version>

</plugin>

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

**Привязка к фазам сборки проекта**

После того как плагин объявлен, его можно настроить так, чтобы он автоматически запускался в нужный момент. Это делается с помощью привязки плагина к [фазе](http://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>

</goals>

</execution>

</executions>

</plugin>

в данном примере плагин запустится в фазе проекта package

**Настройки**

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.

Репозитории - это место где хранятся артефакты: jar файлы, pom -файлы, javadoc, исходники. Существуют:

* Локальный репозиторий по умолчанию он расположен в <home директория>/.m2/repository - персональный для каждого пользователя.
* центральный репозиторий который расположен в <http://repo1.maven.org/maven2/> и доступен на чтение для всех пользователей в интернете.
* Внутренний "Корпоративный" репозиторий- дополнительный репозиторий, один на несколько пользователей.

**Локальный репозиторий**

Локальный репозиторий по умолчанию расположен в <home директория>/.m2/repository. Здесь лежат артефакты которые были скачаны из центрального репозитория либо добавлены другим способом. Например если вы наберёте команду

mvn install

в текущем проекте, то соберётся jar (или war, pom в зависимости от содержимого тэга packaging) который установится в локальный репозиторий. Найти его можно в <home директория>/.m2/repository/<groupIdPath>/<artifactId>/<version>/<artifactId>-<version>.jar где groupIdPath получается заменой всех точек на слеши. Например для проекта

<groupId>ru.apache-maven</groupId>

<artifactId>site</artifactId>

<version>1.0-SNAPSHOT</version>

jar файл будет лежать по пути: <home директория>/.m2/repository/ru/apache-maven/site/1.0-SNAPSHOT/site-1.0-SNAPSHOT.jar

**Центральный репозиторий**

Чтобы самому каждый раз не создавать репозиторий, сообщество для Вас поддерживает центральный репозиторий. Если для сборки вашего проекта не хватает зависимостей, то они по умолчанию автоматически скачиваются с <http://repo1.maven.org/maven2>. В этом репозитории лежат практически все опенсорсные фреймворки и библиотеки.

Самому в центральный репозиторий [положить](http://maven.apache.org/guides/mini/guide-central-repository-upload.html) 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/>

**Корпоративный репозиторий**

Если вы хотите создать свой репозиторий, содержимое которого вы можете полностью контролировать(как локальный), и сделать так, чтобы он был доступен для нескольких человек, вам будет полезен корпоративный репозиторий. Доступ к артефактам можно ограничивать настройками безопасности сервера так, что код ваших проектов не будет доступен извне.

Чтобы добавить репозиторий в список, откуда будут скачиваться зависимости, нужно добавить секцию repositories в pom.xml, например:

<project>

...

<repositories>

<repository>

<id>my-company-repo</id>

<url><http://my-company-site.ru/repo></url>

</repository>

</repositories>

...

</project>

Существуют несколько реализаций серверов - репозиториев maven. Наиболее известные это [artifactory](http://www.jfrog.org/products.php), [continuum](http://continuum.apache.org/), [nexus](http://nexus.sonatype.org/).

**Основные фазы сборки проекта**

1. compile

Компилирование проекта

1. test

Тестирование с помощью JUnit тестов

1. package

Создание .jar файла или war, ear в зависимости от типа проекта

1. integration-test

Запуск интеграционных тестов

1. install

Копирование .jar (war , ear) в локальный репозиторий

1. deploy

публикация файла в удалённый репозиторий

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 (javadoc+сайт описания проекта)

Т. к . команда mvn понимает когда ему передают несколько фаз то для сборки проекта создания документации "с нуля" выполняют:

mvn clean package site

С полным списком фаз и их описанием (на английском языке) можно ознакомиться здесь <http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference>

**Основные сведения**

Мавен изначально создавался , принимая во внимание портируемость. Но довольно часто приложение приходится запускать в разном окружении: например, для разработки используется одна база данных, в рабочем сервере используется другая. при этом могут понадобиться разные настройки, разные зависимости и плагины. Для этих целей в maven используются профайлы.

Давайте определим два профайла: один для разработки, другой для производственного сервера. Для разработки вполне подойдёт база hsqldb, которая хранит все данные в памяти. На производственном сервере же используется база данных postgres, которая сохраняет все данные на диск. В профайлах для каждой конфигурации определены свои проперти database.url и зависимости для разных jdbc драйверов.

Ниже приведён пример объявления таких профайлов.

<?xml version="1.0" encoding="UTF-8"?>

<project>

<profiles>

(1) <profile>

<id>development</id>

<properties>

<database.url>jdbc:hsqldb:mem:testdb</database.url>

</properties>

<dependencies>

<dependency>

<groupId>org.hsqldb</groupId>

<artifactId>hsqldb</artifactId>

<version>2.0.0</version>

</dependency>

</dependencies>

</profile>

(2) <profile>

<id>productionServer</id>

<properties>

<database.url>jdbc:postgresql://databseserver/database</database.url>

</properties>

<dependencies>

<dependency>

<groupId>postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>9.0-801.jdbc4</version>

</dependency>

</dependencies>

</profile>

</profiles>

.....

</project>

цифрами 1 и 2 обозначены начала объявления профайлов. каждый профайл имеет идентификатор в данном случае development и productionServer.

Внутри тэга <profile> содержатся все те же объявления что и внутри <project>: properties, dependencies, и др. Вот полный список тегов которые могут содержаться внутри профайлов:

\* <repositories>

\* <pluginRepositories>

\* <dependencies>

\* <plugins>

\* <properties>

\* <modules>

\* <reporting>

\* <dependencyManagement>

\* <distributionManagement>

\* <build> тэг, который может содержать

o <defaultGoal>

o <resources>

o <testResources>

o <finalName>

**Активация профайла**

Чтобы содержимое тэга профайла "работало", нужно профайл активировать. Когда профайл активирован, его содержимое объединяется с основной частью pom.xml. Нужно заметить, что активных профилей одновременно может быть несколько.   
Активировать профайл можно несколькими способами:  
во первых, это можно задать вручную в командной строке запуска maven, например: mvn package -P production   
Во вторых, при объявлении самого профайла можно задать тэг <activation>, который определяет какой профайл будет активирован: в нашем примере профайл development активный по умолчанию: <activation><activeByDefault>true</activeByDefault></activation>. Кроме активации по умолчанию можно задать активацию на основе операционной системы, установленных переменных окружения, версии JDK.

В командной строке можно задать, какие профили будут деактивированы: mvn goal -P !profile-1,!profile-2 //приоритет командной строки выше

Активные профайлы можно также задать в ~/.m2/settings.xml

<settings>

...

<profiles>

<profile>

<id>appserverConfig</id>

<properties>

<appserver.home>/path/to/appserver</appserver.home>

</properties>

</profile>

</profiles>

<activeProfiles>

<activeProfile>appserverConfig</activeProfile>

</activeProfiles>

...

</settings>

**Отладка**

Чтобы проверить работу и, возможно, найти ошибки, полезны следующие плагины:

* для того, чтобы показать какие профайлы сейчас активны, можно набрать:

mvn help:active-profiles

причём опции командной строки принимаются во внимание. команда

mvn help:active-profiles -P productionServer,development

выведет, как и положено

The following profiles are active:

- development (source: pom)

- productionServer (source: pom)

* Также можно посмотреть pom, полученный после объединения основной части и активных профайлов:

mvn help:effective-pom -P productionServer

**Разработка веб приложений**

**Создание с помощью archetype:generate**

Давайте сгенерируем простое web приложение с помощью [плагина](http://www.apache-maven.ru/plugins/index.html) maven-archetype:plugin.

mvn archetype:generate

В результате нам вывалится очень большой список архетипов, и maven предложит вам выбрать. Выбирать в таком списке довольно сложно, поэтому будет удобно если вы отфильтруете этот список. Наберите

maven-archetype-webapp

и maven предложит вам список из нескольких архетипов.

Choose archetype:

1: internal -> org.apache.maven.archetypes:maven-archetype-webapp (A simple Java web application)

Выберите

**[1]** org.apache.maven.archetypes:maven-archetype-webapp.

Напишите   
groupId : **ru.apache-maven**  
artifactId : **webapptest**  
а остальные значения можно оставить по умолчанию.  
в результате получится проект с такой структурой директорий:

├── pom.xml

└── src

└── main

├── resources

└── webapp

├── index.jsp

└── WEB-INF

└── web.xml

Поздравляю, простейший веб-проект готов. Теперь можно:

* собрать с помошью mvn package

в результате в директории target образуется testwebapp.war, который можно деплоить в сервлет контейнер, например в [Apache Tomcat](http://tomcat.apache.org/)

* запустить напрямую: **mvn tomcat:run**

В этом случае запустится tomcat и приложение будет сразу доступно по адресу <http://localhost:8080/webapptest/>

**Создание веб проекта с нуля**

Разработка Веб-приложений Создать веб-проект довольно просто- нужно добавить тэг <packaging> внутри <project> и задать в качестве значения значение war:

<project xmlns="<http://maven.apache.org/POM/4.0.0>"

xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"

xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0>

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

<modelVersion>4.0.0</modelVersion>

<groupId>ru.apache-maven</groupId>

<artifactId>parent-project</artifactId>

<version>1.0-SNAPSHOT</version>

**<packaging>war</packaging>**

</project>

Всё, веб проект практически готов , осталось создать правильную структуру директорий и файл web.xml. Здесь будет приведена структура директорий по умолчанию, хотя её легко изменить под ваши нужды.

├── pom.xml

└── src

└── main

├── java

├── resources

└── webapp

├── index.jsp

└── WEB-INF

└── web.xml

**Ссылки**

Для дальнейшей работы с веб-приложениями советую прочитать:

* [maven-war-plugin](http://maven.apache.org/plugins/maven-war-plugin/)
* [maven-tomcat-plugin](http://tomcat.apache.org/maven-plugin-2.2/index.html)
* [maven-jetty-plugin](http://docs.codehaus.org/display/JETTY/Maven+Jetty+Plugin)

Переходим на Maven

В связи с тем, что мы подошли к важному рубежу — программирование на Java для web — есть смысл перейти на новый уровень построения и сборки нашего проекта. Раньше dct было достаточно просто — у нас обычное приложение, которое использовало по сути только одну дополнительную библиотеку — JDBC драйвер для PostgreSQL. Мы могли его подключить в виде JAR-файла, указав в CLASSPATH и это не должно было вызывать большого количество проблем. Теперь же наше приложение резко усложняется. Во-первых — нам надо будет подключать больше библиотек. Причем надо будет не просто брать определенные JAR-файлы — нам надо точно знать версию подключаемого файла. Кроме этого, для компиляции нам потребуется один набор библиотек, а для работы нашего приложения — другой. Думаю, что вы уже догадываетесь, что в случае сложного приложения управление набором подключаемых JAR-файлов — очень утомительная задача, особенно, если все делать в ручном режиме.  
Также мы должны получить не просто набор скомпилированных class-файлов — нам надо собрать их в один архив — WAR (Web ARchive), структура которого уже не такая простая, как у JAR-файла, Опять же — можно собирать его вручную, но это сложно и неудобно.  
Тот, кто знаком с IDE, может сказать — а зачем нам это все надо ? Тот же NetBeans достаточно хорошо умеет делать сборки WAR. Или Eclipse. (Что касается Intellij IDEA, то здесь несколько сложнее — web-программирование поддерживает только платная версия).  
Но тут же возникает неудобство — тогда надо всем использовать только определенную IDE, что весьма неудобно в большом коллективе разработчиков. Если мы говорим о сообществе OpenSource, то сборка с помощью определенной IDE принимает совсем жалкий вид. К тому же даже в самой продвинутой IDE управление набором подключаемых JAR-файлов весьма непростая задача.  
В итоге мы приходим к очевидному выводу — нужен какой-то унифицированный и простой продукт. Желательно, чтобы он мог запускаться вообще из командной строки (наподобие javac) и использовал только возможности JDK.  
Для решения всего этого клубка проблем существуют специальные программные продукты — build tools — средства сборки. Появились они еще в 70-х года прошлого века. На сегодня я бы выделил три продукта для Java-программистов:

1. Ant
2. Maven
3. Gradle

**Ant** хоть и используется, но на мой взгляд, это больше дань огромному количеству унаследованных возможностей, которые были созданы для него.  
Что же касается **Maven** и **Gradle**, то это два мощных инструмента, которые умеют позволяют управлять проектами в достаточно удобной форме.  
В этой статье речь пойдет о **Maven** — это не значит, что **Gradle** хуже (В Интернете можно найти огромное количество холиваров на тему «что лучше — Maven или Gradle». Мое мнение — выбирайте сами, руководствуясь здравым смыслом). Возможно, что для некоторых случаев **Gradle** даже лучше, но по моим наблюдениям, на данный момент **Maven** используется чаще. Так что давайте начнем предварительное знакомство с этим продуктом. В нашем путешествии по технологиям Java он будет нам хорошим помощником и сполна обеспечит нас необходимой функциональностью.  
Также необходимо отметить, что все распространенные IDE очень неплохо интегрируют Maven внутрь себя, что не может не радовать. Так что начнем.

Установка Maven

Установка **Maven** достаточно простая. Это касается очень большого количества различных программных продуктов OpenSource для Java. Установка заключается в том, чтобы скачать архив и распаковать его в какой-то каталог. Дальше бывает удобно прописать путь до нужной программы в переменную окружения **Path** (этот пункт был описан в разделе [Установка JDK](http://java-course.ru/begin/install-jdk/)). И еще бывают случаи, когда необходимо добавить переменные среды (мы уже с ними сталкивались при установке JDK — добавляли JAVA\_HOME). Вот собственно и вся установка. Что касается **Maven**, то мы по сути это и сделаем.

1. Скачаем архив Maven с официального сайта. Пока он такой [Download Apache Maven](https://maven.apache.org/download.cgi). Но проще набрать в поисковой строке «maven download» и первой же строкой вы получите искомое
2. Распакуйте архив в определенную директорию. Как я уже неоднократно упоминал, я устанавливаю все продукты в отдельную директорию **java**
3. В архиве есть каталог **bin**. Путь до этого каталога (для Windows это может быть что-то вроде **C:\java\apache-maven-3.5.0\bin**) я прописываю в переменной **Path**
4. **ВАЖНО** Убедитесь, что установлена переменная среды **JAVA\_HOME**. Без нее работа Maven просто невозможна
5. Некоторые программы используют Maven и им нужна информация о каталоге, в котором он установлен. Для этого используются переменные среды. Я рекомендую добавить переменную среды **M2\_HOME**, которая будет указывать на каталог, где установлен Maven. На данный момент этой переменной бывает достаточно, но в будущем ситуация может измениться. Кроме этой переменной можно использовать **M3\_HOME** или **MAVEN\_HOME**. В данном случае многое зависит от той программы, которая хочет использовать Maven. Что она захочет — то в общем-то и надо установить. Этот шаг я бы не назвал обязательным, но чтобы не испытывать проблем, можно это сделать

После окончания наших действий самое простое — это запустить Maven из командной строки. Можно просто набрать команду



|  |  |
| --- | --- |
| 1 | mvn -h |

В ответе вы должны увидеть длинный список опций для Maven. Собственно, на этом первоначальную установку можно считать законченной.

Настройка Maven

На первых порах я наверно не буду вдаваться в тонкости настройки Maven. Пока укажу следующее — в корневом каталоге Maven существует каталог **conf**. В нем находится очень важный файл **settings.xml**. Этот файл является весьма уважаемым — в нем устанавливается много настроек, которые могут радикально повлиять на поведение Maven. Но не пугайтесь — «может» совсем не означает, что это обязательно надо делать. Для начала работы вы можете его совсем не трогать — что не может не радовать. Настроек по умолчанию вам должно хватить. При более глубоком изучении Maven вы сможете вносить изменения и тем самым влиять на его поведение.

Простое понимание архитектуры Maven

Maven может решать очень сложные задачи по управлению проектом. В общем его позиционируют не как средство сборки, а как средство управления проектом. Что я считаю вполне справедливым. Учитывая мощные возможности расширения, Maven может выполнять очень сложные задачи. Но для начала (на мой взгляд) будет правильно взглянуть на него достаточно упрощенно.  
Что представляет собой процесс разработки ? В самом начале я описал эти шаги, но чтобы не гонять вас туда-сюда, да и для уточнения списка этих шагов, я перечислю их здесь. Итак:

1. Набрать текст программы для исправления ошибок или добавления новой функциональности
2. Скомпилировать программу
3. Запустить тестирующую программу для проверки
4. Собрать нужные классы в определенный архив или набор файлов, которые можно устанавливать
5. Установить готовый архив в среду исполнения и проверить его работоспособность
6. Если обнаружены ошибки — вернуться к первому пункту

И так постоянно. Редактируем, компилируем, проверяем, собираем, запускаем для тестирования.  
Каждый из этих шагов требует достаточно понятных действий. Для первого шага нам нужен редактор, потом компилятор, потом — программа тестирования.  
Давайте еще раз выделим этот момент — у нас есть определенные этапы (список которых достаточно четкий) и под каждый этап у нас есть (или должна быть) программа (возможно несколько программ), которая его выполняет. Совсем просто — есть этап и есть программа для его выполнения. И есть программа, которая выполняет эти этапы один за другим.  
В этом и есть вся идея Maven. Работа над обработкой проекта делится на определенный набор этапов — в Maven это называется “фаза” (phase). Под каждую фазу существует уже готовая программа — плагин (plugin). Каждый плагин обладает рядом настроек, которые можно менять под свои нужды. Опять же — только в случае необходимости.  
Что еще важно учесть — последовательность фаз четко определена. Если вы хотите проект собрать в архив, то перед этим вы пройдете фазу компиляции и фазу запуска тестов. Конечно же, вы можете управлять выполнением фаз, но это требует определенных усилий. К тому же надо учесть очень простой факт — в подавляющем большинстве проектов пропускать какие-то фазы или менять их порядок — это настолько экзотично, что тратить сейчас на это наше время — да ни за что..  
Чтобы я еще хотел отметить — вы также можете просто запустить отдельный плагин. В этом случае вы указываете плагин и цель (goal) через двоеточие. Большинство плагинов умеют выполнять несколько целей (несколько функций), одна из которых может быть указана по умолчанию. Каждая задача по сути ассоциирована с целью. Вот вы и указываете — плагин+цель. Пока об этом все.

Создаем Maven-проект

Прежде чем запускать maven, убедитесь, что ваш компьютер подключен к интернету — позже мы увидим, почему это важно. А пока приступим к созданию нашего первого проекта.  
Большинство руководств предложат вам запустить специальный Maven-плагин — **archetype** с целью **generate**. Выглядит это вот так:.



|  |  |
| --- | --- |
| 1 | mvn archetype:generate -DgroupId=edu.javacourse -DartifactId=sample -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false |

Не задавайтесь вопросом о всех параметрах — ниже вы о них прочитаете.  
Проверили переменную JAVA\_HOME и доступ к Интернет ? Тогда запускайте командную строку, переходите в каталог с вашими проектами (надеюсь, вы не просто читали мои статьи и у вас есть такой каталог) и запускайте команду. В процессе у вас будет выводиться много разной информации, в частности, вы увидите достаточно много строк, похожих на такие:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Downloading: https://repo.maven.apache.org/maven2/asm/asm-commons/3.2/asm-commons-3.2.jar  Downloaded: https://repo.maven.apache.org/maven2/asm/asm/3.2/asm-3.2.jar (43 KB at 27.5 KB/sec)  Downloading: https://repo.maven.apache.org/maven2/asm/asm-util/3.2/asm-util-3.2.jar  Downloaded: https://repo.maven.apache.org/maven2/asm/asm-commons/3.2/asm-commons-3.2.jar (33 KB at 20.3 KB/sec)  Downloading: https://repo.maven.apache.org/maven2/asm/asm-analysis/3.2/asm-analysis-3.2.jar  Downloaded: https://repo.maven.apache.org/maven2/org/apache/velocity/velocity/1.7/velocity-1.7.jar (439 KB at 267.2 KB/sec) |

Важно, чтобы в конце вы увидели вот такие строки:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [INFO] ------------------------------------------------------------------------  [INFO] BUILD SUCCESS  [INFO] ------------------------------------------------------------------------  [INFO] Total time: 1.576 s  [INFO] Finished at: 2017-09-04T18:01:15+03:00  [INFO] Final Memory: 14M/214M  [INFO] ------------------------------------------------------------------------ |

Это сообщение говорит о том, что наш проект создан и мы можем начать изучение Maven. Если же вы увидите слова **BUILD FAILURE**, то вам будет сложно. Я не смогу увидеть ваши ошибки и указать причину. Надо читать сообщения, которые maven выводит на экран.  
Если же все закончилось успешно (а это при правильной установке и настройке наиболее вероятно), у вас будет создан каталог **sample**, который на некоторое время станет для вас сосредоточением всего, что касается Maven. Перебираемся в него.

Что такое Maven-проект

Прежде чем мы с головой окунемся в команды Maven я бы хотел, чтобы вы посмотрели на те файлы и каталоги, которые сгенерировал maven.  
Не могу не отметить важное — даже самый лаконичный Maven-проект (который в своем описании содержит пару строк) уже умеет многое — он способен пройти через все основные фазы, используя настройки плагинов по умолчанию и вы получите уже готовый к установке набор файлов.  
Итак, в первую очередь мы должны посмотреть на файл **pom.xml**. Это наиважнейший файл, без которого вообще ничего не будет. Именно он в ответе за конфигурацию проекта. Сейчас он выглядит вот так:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <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>edu.javacourse</groupId>    <artifactId>sample</artifactId>    <packaging>jar</packaging>    <version>1.0-SNAPSHOT</version>    <name>sample</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> |

Мы еще вернемся к этому замечательному файлу, а пока посмотрим на директорию **src**, которая находится в этом же каталоге. Я хочу, чтобы вы зашли внутрь и увидели еще две директории — **main** и **test**. Мы поговорим об их предназначении, а пока давайте удалим директорию **test** и вернемся к нашему файлу **pom.xml**. Откройте его обычным редактором — проще всего наверно Notepad — и сделайте из него вот такой:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <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>edu.javacourse</groupId>    <artifactId>sample</artifactId>    <version>1.0-SNAPSHOT</version>    </project> |

Как видите, в нашем файле очень немного информации и тем не менее мы можем использовать его для нашей работы. Прежде чем мы начнем изучать команды Maven я бы хотел указать важные элементы файла — тем более, что их всего четыре.

1. modelVersion
2. groupId
3. artifactId
4. version

**modelVersion** — этот элемент говорит о версии pom.xml. Уже много лет он остается неизменным, так что воспринимайте его как необходимость. Он просто должен быть.  
Следующие три элемента определяют ваш проект как уникальный объект-артефакт.  
**groupId** — это обобщающее имя группы проектов, которые могут быть связаны неким общим функционалом. Но это именно группа  
**artifactId** — уникальный идентификатор проекта в данной группе  
**version** — версия проекта. Ваш проект развивается, наполняется функциональностью и постоянно версии все новые и новые выходят. Их же надо как-то отслеживать. Вот отсюда и появился этот атрибут. У нас версия имеет возможно непривычное значение **1.0-SNAPSHOT**, но такое именование имеет свое объяснение и я расскажу вам об этом в свое время.

Теперь, если вы вернетесь к нашей первой команде, то увидите, что практически все вышеназванные параметры мы указали, предварив их опцией **-D**. Единственный неизвестный — это **interactiveMode=false**. Вкратце — он отключает кучу вопросов, которые задавал бы плагин при своей работе. Когда вы не очень понимаете, о чем разговор это только раздражает. Вот я и подумал — пусть помолчит.

Итак, по сути все, что вам надо для работы, уже есть — наш минимальный **pom.xml** готов. Давайте учиться работать с Maven используя этот лаконичный вариант.

Начальные команды Maven

Не уверен, что в случае отсутствия опыта работы с командной строкой, вам понравится предлагаемый мной вариант работы с maven, но все-таки давайте привыкать.  
Я достаточно много работаю с Unix (не как администратор, а как программист) и командная строка для меня весьма эффективный инструмент. Это часто распространяется и на работу в среде Windows.  
В качестве рекомендации — УЧИТЕСЬ РАБОТАТЬ В КОМАНДНОЙ СТРОКЕ.  
Достаточно часто это очень эффективно и очень надежно в плане понимания, что конкретно вы делаете. Как я уже говорил, многие IDE неплохо интегрированы с Maven, но я наталкивался на ситуации, когда они скрывают некоторые моменты и вы не очень понимаете, что вы делаете и можете совершить ошибку. В командной строке вы гораздо полнее контролируете процесс и это помогает решать проблемы.быстрее.  
Сейчас мы будем знакомиться с командами Maven, которые помогут нам более полно понять, как работает этот инструмент.  
К тому же, современные процессы управления проектами включают такие понятия как Continues Integration, ночные сборки. И они пользуются запуском Maven именно из командной строки. Когда вы владеете командной строкой — вы быстрее сможете разобраться в проблеме.

Итак, как я уже говорил ранее, Maven по сути идет по указанному списку фаз и под каждую фазу (ну под основные точно) у него есть обработчик — плагин. Полный список фаз вы можете посмотреть здесь: [Lifecycles Reference](http://maven.apache.org/ref/3.5.0/maven-core/lifecycles.html). Но есть несколько основных, о которых мы и будем говорить.  
Их можно разделить на три группы. Первая и самая большая группа — это фазы сборки. Вот они:

1. validate
2. compile
3. test
4. package
5. verify
6. install
7. deploy

Фазы иду одна за другой — т.е. если указать фазу **test**, то сначала будет выполнена фаза **validate**, затем **compile** и только после этого **test**. Немного слов о каждой фазе я скажу прямо сейчас.  
Итак:  
**validate** — фаза, на которой происходит проверка **pom.xml**. Maven должен убедиться, что он корректный и его можно принять к обработке,  
**compile** — думаю, что вы уже догадались. Это фаза компиляции файлов java.  
**test** — запуск модульных тестов. О тестах мы пока еще не говорили, но обязательно поговорим.  
**package** — сборка файлов проекта в архив. По умолчанию это JAR-файл. Также можно сделать WAR или EAR. Мы пока о таких архивах не говорили, но основная идея этой фазы — сделать готовый к использованию набор файлов.  
**verify** — проверка через интеграционные тесты. Это в какой-то степени подобно модульным тестам, но несколько иное.  
**install** — установка в локальный репозиторий.  
**deploy** — установка на удаленный репозиторий. Назначение двух последних фаз мы раскроем позже.  
Вторая группа состоит по сути только из одной команды:

1. **clean** — очистка проекта

Ну и наконец третья — тоже одна команда:

1. **site** — создание документации на проект

Запуск Maven из командной строки достаточно простой. Для примера запустим цикл до фазы **package**. Выглядит это так:



|  |  |
| --- | --- |
| 1 | mvn package |

В результате вы можете увидеть опять много строк, которые что-то загружают:.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Downloading: https://repo.maven.apache.org/maven2/asm/asm-commons/3.2/asm-commons-3.2.jar  Downloaded: https://repo.maven.apache.org/maven2/asm/asm/3.2/asm-3.2.jar (43 KB at 27.5 KB/sec)  Downloading: https://repo.maven.apache.org/maven2/asm/asm-util/3.2/asm-util-3.2.jar  Downloaded: https://repo.maven.apache.org/maven2/asm/asm-commons/3.2/asm-commons-3.2.jar (33 KB at 20.3 KB/sec)  Downloading: https://repo.maven.apache.org/maven2/asm/asm-analysis/3.2/asm-analysis-3.2.jar  Downloaded: https://repo.maven.apache.org/maven2/org/apache/velocity/velocity/1.7/velocity-1.7.jar (439 KB at 267.2 KB/sec) |

Если вы все сделали правильно, то у вас должно появиться сообщение об успешном завершении сборки. Вот такое:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [INFO] ------------------------------------------------------------------------  [INFO] BUILD SUCCESS  [INFO] ------------------------------------------------------------------------  [INFO] Total time: 0.851 s  [INFO] Finished at: 2017-09-04T21:59:24+03:00  [INFO] Final Memory: 18M/492M  [INFO] ------------------------------------------------------------------------ |

Давайте разбираться, что же мы тут такое натворили.  
Во-первых — посмотрим на наш каталог с проектом. Вы можете увидеть уже не только директорию **src**, но и новую — **target**.  
Самое время рассказать, что это такое за директории.  
**src** — это директория, где находятся файлы проекта — в основном это файлы java, но не только. Еще узнаем, какого рода файлы там могут быть, но не сейчас. Мы уже заходили внутрь этой директории и даже стерли там директорию **test**. Обычно там две директории: **main** и **test**. В первой (main) находятся файлы проекта. Если зайти внутрь, то мы увидим там директорию **java**. А дальше — это обычная структура директорий для пакетов. В нашем случае это **edu/javacourse/App,java**. Класс **App.java** создался автоматически при создании проекта, как пример. Он нас пока мало интересует.  
Вторая директория (test) тоже содержит директорию **java**, но в ней располагаются файлы, предназначенные для тестирования. Мы про них пока не думаем, но обязательно вернемся при первой возможности (но не в этой главе).  
Если заглянуть в директорию **target**, то там можно будет увидеть вот такой набор директорий:



|  |  |
| --- | --- |
| 1  2  3 | classes  maven-archiver  maven-status |

а также файл **sample-1.0-SNAPSHOT.jar**.

Для начала остановим наше внимание только на директории **classes** и файле **sample-1.0-SNAPSHOT.jar**  
В директории мы увидим скомпилированный класс **edu/javacourse/App.class**. Думаю, что вы догадались — это результат этапа компиляции (фаза compile). Файл — результат сборки нашего проекта (фаза package). Можно увидеть, что его имя состоит из комбинации параметров нашего файла **pom.xml** — **artifactId** и **version**.  
Обращаю ваше внимание — не из имени директории, а именно из **artifactId**. Чуть позже мы с вами поэкспериментируем и увидим, что это так.  
Давайте научимся очищать наш проект — фаза **clean**.



|  |  |
| --- | --- |
| 1 | mvn clean |

Посмотрите на состав директории нашего проекта — директория **target** исчезла. Теперь наш проект снова готов к сборке. В принципе он всегда готов к сборке, просто после фазы **clean** мы все делаем заново — компилируем и собираем. Если же не “чистить” проект, то директория **target** не удаляется. Вот теперь самое время провести эксперимент. Давайте еще раз соберем наш проект командой



|  |  |
| --- | --- |
| 1 | mvn package |

Убедитесь, что файл **sample-1.0-SNAPSHOT.jar** появился в директории **target**. Теперь откройте на редактирование файл **pom.xml** и исправьте строку с тэгом **artifactId** с **sample** на **hello**. Снова запустите команду



|  |  |
| --- | --- |
| 1 | mvn package |

После сборки зайдите в директорию **target** и вы увидите файл **hello-1.0-SNAPSHOT.jar**.  
Фазы можно совмещать — не надо набирать сначала **mvn clean** и потом **mvn package**. Можно сразу набрать вот так:



|  |  |
| --- | --- |
| 1 | mvn clean package |

Поиграйте с командами — попробуйте сделать только **mvn clean compile** и вы увидите, что компиляция прошла (появилась директория **target/classes**), но JAR-файл не появился.  
Команды **install** и **deploy** пока нет смысла выполнять — мы как раз подходим к тому моменту, когда вам станет понятно, что такое репозитории.

Репозитории Maven

Если вы несколько раз запускали команды Maven, то могли заметить, что строки, которые начинаются с **Downloading …** уже не столь часто встречаются. Или совсем не встречаются. В чем же тут разгадка.  
Одним из больших преимуществ Maven заключается в том, что это не просто средство сборки — это также и целая среда для управления проектами.  
Для работы Maven постоянно нуждается в сторонних библиотеках. Те же плагины — это сторонние библиотеки. Но откуда же они появляются? Вопрос совершенно правильный — мы же ничего, кроме JDK и Maven не устанавливали (да, еще IDE было, но там тоже не все есть). И если вы посмотрите на состав библиотек в директории Maven, то вряд ли найдете там все, что необходимо. Но тогда откуда? Ответ очень простой — из внешних хранилищ — репозиториев. Еще раз приведу часть строки, которые много раз должны были появляться на экране:



|  |  |
| --- | --- |
| 1 | Downloading: https://repo.maven.apache.org/... |

Вот оно, чудо. Maven “ходит” во внешний репозиторий по указанному адресу и оттуда скачивает необходимые библиотеки. Но сразу приходит в голову мысль: “Если он все время что-то скачивает, то это же крайне неэффективно”. Да и складывать все это скачанное куда-то надо. Если вы уже так подумали — то вы совершенно правы.  
Штука в том, что у вас на компьютере Maven размещает ЛОКАЛЬНЫЙ репозиторий.  
И скачивает необходимые файлы только один раз. После этого размещает их в локальном репозитории, откуда и берет для работы. Так где же этот локальный репозиторий размещается на вашем диске? Самое время посмотреть файл **settings.xml** — я выше писал о нем. Располагается он в директории **conf** в каталоге, где установлен Maven.  
Если посмотреть внимательно, то можно найти вот такой кусочек:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <!-- localRepository     | The path to the local repository maven will use to store artifacts.     |     | Default: ${user.home}/.m2/repository    <localRepository>/path/to/local/repo</localRepository>    --> |

В комментариях нам говорят, что тэг **localRepository** позволяет указать директорию, в которую будут складываться все скачанные файлы. На данный момент этот тэг закомментирован, поэтому используется значение по умолчанию. А именно — ваш домашний каталог + **.m2/repository**.  
Для пользователей Windows это достаточно часто каталог **C:\Users\[ваш логин]**. Для русскоязычной персии каталог **Users** называется **Пользователи**. Что же касается пользователей Unix-систем, то это ваша home-директория (те, кто пользуется Unix должны быть в курсе).  
Давайте попробуем посмотреть, что же там есть. Все начинается с каталога **repository**, в котором уже много чего есть. Все эти директории — результат скачивания. Если вы еще не запускали из любопытства команду



|  |  |
| --- | --- |
| 1 | mvn install |

из каталога нашего проекта — самое время сделать это. Запустили ? Все закончилось успешно ? Тогда смотрим в наш локальный репозиторий. Что, уже улыбаетесь ? Не знаю, как вы — я улыбался (про себя конечно) от понимания простоты.  
Итак, в каталоге с локальным репозиторием появилось дерево каталогов вот такой структуры:



|  |  |
| --- | --- |
| 1 | edu/javacourse/sample/1.0-SNAPSHOT |

Организация достаточно простая — каталоги верхнего уровня соответствуют (вы удивились этой простоте?) тэгу **groupId** из уже знакомого нам файла **pom.xml**. Далее следует **artifactId** и под конец **version**. Очень просто и надежно.  
Можно попробовать поменять номер версии на **1.0** и снова запустить команду **mvn install**. Увидели новый каталог ? Молодцы.

Итак, что мы в конечном итоге должны понять ? У нас есть место на диске, куда Maven складывает нужные ему библиотеки/плагины/и прочая. После скачивания из удаленного репозитория он их использует локально.  
Проекты, которые мы разрабатываем сами, легко собираются и размещаются в нашем локальном репозитории. Еще увидим, насколько это удобно и важно.

Что лично мне очень нравится — вы можете удалить все, что есть в локальном репозитории — и вам за это ничего не будет. При запуске Maven опять скачает все необходимое. Вам надо просто набраться терпения.

Подключение внешних библиотек

Как мы уже смогли увидеть, Maven делает за нас очень важную работу — собирает наш проект и в процессе сборки скачивает из удаленного репозитория (https://repo.maven.apache.org) нужные библиотеки. Как он понимает, какие именно нам нужны ? Да очень просто — по **groupId**, **artifactId** и **version**. Например, библиотека JUnit для тестирования имеет вот такой URI: [JUnit 4.12](https://repo.maven.apache.org/maven2/junit/junit/4.12/)

Т.е. все, что нам надо для подключения нужной библиотеки — просто указать нужные **groupId**, **artifactId** и **version** в нашем проекте. Просто ? Да не то слово.

Что же для этого нам надо сделать ? Я думаю, что для нас будет просто отличным примером следующее:

1. Перевод нашего проекта “Список контактов” на Maven
2. Подключение библиотеки, которая реализует пул соединений с базой данных

Переезжаем на Maven

Конечный результат вы можете скачать по этой ссылке: [Список контактов](http://java-course.ru/first/sources/ContactProject_04.zip).  
Открыть проект можно прямо в NetBeans. Что мне нравится в netBeans, так это то, что он открывает Maven-проекты без каких-либо дополнительных действий. Просто найдите каталог с проектом и открывайте. При этом не создает никаких дополнительных файлов и каталогов, в отличии от той же Intellij IDEA или Eclipse. Хотя и эти IDE прекрасно справляются.  
Переезд на Maven в нашем случае был достаточно простой — бОльшая часть работы заключается в копировании файлов в нужные каталоги.  
Но все-таки есть моменты, на которые надо обратить внимание.  
Во-первых, настало время узнать, что в каталоге **src/main** находится не только каталог **java**.  
Как вы можете видеть, у нас появился каталог **resources**. Этот каталог имеет специальное назначение — в него помещают файлы, которые нужны проекту, но не являются файлами с исходным кодом. Это наши старые знакомые — файлы properties. Причем я точно соблюдаю вложенность каталогов — они все также имеют структуру **edu/javacourse/contact/gui**. Пока о ресурсах все. Достаточно просто помнить — если у вас есть файлы ресурсов — properties, xml, картинки, какие-то тексты, которые вы хотите использовать как ресурсы проекта — складывайте их в каталог **resources**.  
Во-вторых, у нас несколько усложнился **pom.xml**. Ранее мы имели очень простой файл, а тут будет что-то новенькое. Давайте посмотрим на него:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0"           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">      <modelVersion>4.0.0</modelVersion>        <groupId>edu.javacourse</groupId>      <artifactId>contact</artifactId>      <version>1.0-SNAPSHOT</version>        <build>          <plugins>              <plugin>                  <groupId>org.apache.maven.plugins</groupId>                  <artifactId>maven-compiler-plugin</artifactId>                  <configuration>                      <source>1.8</source>                      <target>1.8</target>                  </configuration>              </plugin>          </plugins>      </build>    </project> |

Первая половина нашего проекта вам уже должна быть понятна. Но вот что это за слова внутри тэга **build**. Попробуем разобраться.  
Я думаю, что вы помните наш разговор о принципах работы Maven — на каждую фазу существует плагин, который .. ну и так далее. Так вот, штука в том, что для фазы **compile** мы используем плагин с именем (его artifactId) **maven-compiler-plugin**. Версию можно не указывать (на момент написания статьи была 3.7.0).  
Кстати, если зайти в наш локальный репозиторий, то можно найти этот плагин — внутри репозитория его путь будет такой: **org/apache/maven/plugins/maven-compiler-plugin**. Посмотрели ? Возвращайтесь обратно к проекту.  
По умолчанию наш плагин предполагает, что он работает с синтаксисом языка 1.5 (во всяком случае текущая версия именно так думает) . А наш замечательный проект использует синтаксис для Java версии 1.7 (минимум).  
Значит нам надо “рассказать” нашему плагину, что он должен работать с более высокой версией. Вот мы и написали, что при сборке проекта надо у плагине для компиляции установить параметры **source** и **target**. Первый параметр (source) говорит, что наша программа написана с использованием синтаксиса Java версии 1.8.Второе же (target) указывает, что результат компиляции будет запускаться на JVM версии 1.8 — следовательно компилятор генерирует байт-код, который может использовать возможности Java 1.8. Что мы с успехом и сделали.  
Теперь можем запустить команду



|  |  |
| --- | --- |
| 1 | mvn clean package |

и убедиться, что создается каталог **target**, в котором мы можем увидеть файл **contact-1.0-SNAPSHOT.jar**.

Запуск проекта из Maven

Перед тем, как начать запускать наш проект — ОБЯЗАТЕЛЬНО проверьте, что он скомпилировался.  
После сборки самое время запустить наше замечательное приложение. Но на этом пути нас подстерегает две проблемы.  
Первая — как запустить какой-нибудь класс из Maven. Она решается достаточно просто — надо использовать плагин. Вообще многие проблемы в Maven решаются именно таким таким способом — использованием плагина. Я уже говорил, что Maven можно расширять. Если конкретизировать, то существует возможность написания собственных плагинов для решения определенных задач. В Maven есть даже специальный архетип для этого. Т.е. можно написать свой плагин, встроить его в определенную фазу и он будет вызываться и исполнять порученную ему задачу. Не буду больше об этом рассказывать — вернемся к основной задаче — надо запустить нашу программу.  
Делается это достаточно просто — вот такая команда позволит нам запустить наше приложение:



|  |  |
| --- | --- |
| 1 | mvn exec:java -Dexec.mainClass=edu.javacourse.contact.test.ContactTest |

Если все до этого было в порядке, то наша программа запустится. Но нас постигнет разочарование — окошко не сможет показать данные из таблиц.  
Если мы посмотрим на те сообщения, которые нам вывелись в процессе выполнения команды, то увидим важное:



|  |  |
| --- | --- |
| 1  2  3  4 | java.lang.ClassNotFoundException: org.postgresql.Driver          at java.net.URLClassLoader.findClass(URLClassLoader.java:381)          at java.lang.ClassLoader.loadClass(ClassLoader.java:424)          at java.lang.ClassLoader.loadClass(ClassLoader.java:357) |

Ну конечно же, как мы могли забыть — нам нужен драйвер PostgreSQL. Но как же нам это сделать?  
Как обычно, решений может быть несколько, но мы воспользуемся возможностью, чтобы посмотреть, как можно подключить внешнюю библиотеку.  
В случае, когда вам надо просто подключить библиотеку, все очень просто — вам надо прописать эту библиотеку в качестве **зависимости (dependency)** в наш файл **pom.xml**. Вот как теперь он будет выглядеть:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0"           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">      <modelVersion>4.0.0</modelVersion>        <groupId>edu.javacourse</groupId>      <artifactId>contact</artifactId>      <version>1.0-SNAPSHOT</version>        <dependencies>          <dependency>              <groupId>postgresql</groupId>              <artifactId>postgresql</artifactId>              <version>9.1-901-1.jdbc4</version>          </dependency>      </dependencies>        <build>          <plugins>              <plugin>                  <groupId>org.apache.maven.plugins</groupId>                  <artifactId>maven-compiler-plugin</artifactId>                  <configuration>                      <source>1.8</source>                      <target>1.8</target>                  </configuration>              </plugin>          </plugins>      </build>    </project> |

Здесь надо обратить внимание на секцию **dependencies**. Все выглядит достаточно логично — я подключаю зависимость, которая однозначно определяется ее  
**groupId**, **artefactId** и **version**. Но у пытливого читателя (я очень надеюсь, что вы именно такие) может сразу возникнуть вопросы типа таких:  
**Откуда автор знает, что надо использовать такие значения ?**  
**Как Maven узнает, где брать эту библиотеку ?**  
Ответ на второй вопрос вы скорее всего поняли — конечно же в удаленном репозитории. Но вот как я нашел параметры? Здесь тоже очень несложно — я пользуюсь поиском Гугл. Просто набираем в поисковой строке что-то вроде такого



|  |  |
| --- | --- |
| 1 | postgresql maven repo |

и практически первая же ссылка будет вести нас на сайт центрального репозитория. Обычно это какие-то такие слова:  
**Maven Repository: postgresql » postgresql**

На момент написания статьи ссылка вот такая: [Maven Repository: postgresql » postgresql](https://mvnrepository.com/artifact/postgresql/postgresql)

На странице вы можете увидеть полный список версий артефакта **postgresql**. Я выбрал версию 9.3-1102.jdbc4 и зашел туда.И здесь вы можете увидеть подсказку, что вы должны вставить в ваш **pom.xml**. Причем подсказка не только для Maven — тут и Gradle, и Ivy, и много чего еще.  
Т.е. теперь вы можете подключать нужные вам библиотеки в проект достаточно просто и элегантно. Прописывайте зависимость в **pom.xml** и готово.  
На само деле подключения библиотек тоже изобилует всякими тонкостями, но пока я думаю, хватит того, что я рассказал.  
Но что важно — теперь мы еще раз можем запустить наше приложение командой



|  |  |
| --- | --- |
| 1 | mvn exec:java -Dexec.mainClass=edu.javacourse.contact.test.ContactTest |

И чудо должно получиться — если все правильно, вы увидите наше окошко со списком контактов.  
Полный код проекта можно скачать по этой ссылке:  
[Список контактов](http://java-course.ru/first/sources/ContactProject_05.zip)

Чуть не забыл, не хотелось вас оставлять в неведении по поводу этой команды. Полагаю, что вы догадались, что строка **-Dexec.mainClass=edu.javacourse.contact.test.ContactTest** как-то связана с нашим классом для запуска. Но вот что значит остальное. Я вкратце опишу для того, чтобы вы на примере увидели общие принципы.  
Итак, первая часть — **exec:java**. Это запуск конкретного плагина — его описание можно найти вот по этой [ссылке](http://www.mojohaus.org/exec-maven-plugin/index.html). Мы указываем его имя и через двоеточие — цель. В данном случае цель **java** запускает собранное приложение.  
И наконец завершает нашу команду указание параметра **exec.mainClass** с указанием нашего класса. Надо отметить, что передача параметров плагину происходит именно так — вы указываете **-D** за которым идет имя параметра и после знака “=” его значение.  
Таким образом можно передавать параметры всем плагинам.  
Но стоит отметить еще один способ установки параметров для плагина — это можно сделать сразу в файле **pom.xml**. Выглядеть это может вот так:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <plugin>      <groupId>org.codehaus.mojo</groupId>      <artifactId>exec-maven-plugin</artifactId>      <version>1.6.0</version>      <configuration>          <mainClass>edu.javacourse.contact.test.ContactTest</mainClass>      </configuration>  </plugin> |

Теперь проект можно будет запускать просто набрав команду



|  |  |
| --- | --- |
| 1 | mvn exec:java |

Корпоративный репозиторий

Так вот Maven позволяет вам добавить в свою конфигурацию ссылки на ваш корпоративный репозиторий. Конечно же существует несколько продуктов, которые позволяют вам организовать в компании такое хранилище. С ним как раз и можно работать через фазу **deploy**.