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). 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. 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

```
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 (точнее можно посмотреть в разделе Введение) 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 -I
```

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

Важно, чтобы в конце вы увидели вот такие строки:

```
[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:

```
<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:

```
<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. 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:

```
mvn package
```

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

```
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/
```

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

```
[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:

```
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**.

```
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

```
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

```
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:

```
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:

```
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:

```
  <!-- 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

```
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:

```
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

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.
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:

```
<?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

```
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:

```

```

```
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:

```
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:

```
<?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

```
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]

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

```
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]

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. 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:

```
<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

```
mvn exec:java
```

Корпоративный репозиторий

Прежде чем мы перейдем к решению задачи для пула коннектов, я бы хотел сказать несколько слов об еще одной важной составляющей Maven. Это возможность подключать свои репозитории в рамках компании. Не секрет, что внутри фирмы при разработки используются свои собственные библиотеки. Само собой, использовать центральный вседоступный репозиторий нет никакого желания.
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**. Но об этом как-нибудь в другой раз.
Собственно вот и все, что хотелось пока сказать, а теперь нас ждет пул коннектов.

Пул соединений

Как я уже говорил в статье Список контактов — работаем с БД, создание соединения дорогостоящая операция и делать это каждый раз для исполнения команды не самое лучшее решение.
Для решения этой задачи в JDBC используется такое понятие как пул соединений, Формально это специальный класс, который содержит в себе несколько уже готовых соединений с базой данных и когда вам требуется соединение, то вы просто получаете его и пула, используете и потом (что очень важно) возвращаете обратно в пул. Если быть точнее, то пул помечает соединение, как занятое и отдает его вам. После использования соединения, пул помечает его как свободное и может отдать кому-то другому. Если запросов на свободное соединение слишком много, то они выстраиваются в очередь и ожидают. При превышении времени ожидании будет генерироваться исключение.
Что крайне важно отметить в данной ситуации — вам ОБЯЗАТЕЛЬНО надо ОСВОБОЖДАТЬ соединения. Для этого крайне важно вызывать метод **close()**. Да-да, не ищите каких-то специальных методов — для освобождения соединения для пула нужно вызывать все тот же метод **close**. И новая конструкция с try для ресурсов становится очень удобным решением.
Для пула соединений это просто жизненно необходимо — освобождать соединения. В случае, если вы этого не будете делать, то количество свободных соединений в пуле очень быстро закончится и ваша система просто перестанет работать. Я регулярно сталкивался (и боюсь, что буду еще сталкиваться) с такой ситуацией — так что будьте очень внимательны.
Пул соединений чаще всего используется при работе серверов приложений. Их там можно настраивать и управлять их характеристиками. Что достаточно удобно.
В нашем случае для standalone-приложения (которое запускается само по себе) существует специальная библиотека, которая позволяет сделать практически то же самое.
Библиотека называется c3p0 (вы возможно подумали — да здравствуют “Звездные войны”. Не угадали — смотрите подробности здесь). Если вдруг сайт уже не существует — это Connection Pool для JDBC 3.0, Далее cp30 и еще далее c3p0. Вот такой вот “пердимонокль”.
Документация может быть получена по адресу c3p0 — JDBC3 Connection and Statement Pooling а сам проект находится здесь: c3p0:JDBC DataSources/Resource Pools

Но мы собираемся подключать это в виде зависимости Maven. Так и сделаем.
Если честно — написание всех этих крайне важных слов про пул коннектов занял у меня гораздо больше времени, чем кодирование. Да вы сейчас сами убедитесь. Я внес изменения в три файла и написал один. Итак, начал я с подключения библиотеки **c3p0** в файл **pom.xml** (само собой искал параметры я через Гугл — думаю, вы уже сами справитесь)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLoca
3      <modelVersion>4.0.0</modelVersion>
4
5      <groupId>edu.javacourse</groupId>
6      <artifactId>contact</artifactId>
7      <version>1.0-SNAPSHOT</version>
8
9      <dependencies>
10         <dependency>
11             <groupId>postgresql</groupId>
12             <artifactId>postgresql</artifactId>
13             <version>9.1-901-1.jdbc4</version>
```

```
        </dependency>

        <dependency>
            <groupId>com.mchange</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.5.1</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>
```

После этого я создал новую реализацию интерфейса **ConnectionBuilder** — класс **ComboConnectionBuilder**. Вот такой он у меня получился.

```
package edu.javacourse.contact.dao;

import com.mchange.v2.c3p0.ComboPooledDataSource;
import edu.javacourse.contact.config.GlobalConfig;

import java.beans.PropertyVetoException;
import java.sql.Connection;
import java.sql.SQLException;

public class ComboConnectionBuilder implements ConnectionBuilder
{
    private ComboPooledDataSource dataSource;

    public ComboConnectionBuilder() {
        try {
            dataSource = new ComboPooledDataSource();
            dataSource.setDriverClass(GlobalConfig.getProperty("db.driver.class"));
            dataSource.setJdbcUrl(GlobalConfig.getProperty("db.url"));
```

```
            dataSource.setUser(GlobalConfig.getProperty("db.login"));
            dataSource.setPassword(GlobalConfig.getProperty("db.password"));
            dataSource.setMaxPoolSize(20);
        } catch (PropertyVetoException e) {
            e.printStackTrace();
        }
    }

    @Override
    public Connection getConnection() throws SQLException {
        return dataSource.getConnection();
    }
}
```

Как можно видеть, ключевую роль тут играет класс **ComboPooledDataSource**. Это класс библиотеки. В конструкторе я его создал и проинициализировал. Думаю, что только последний параметр вам может быть непонятен — остальные все упоминались. Но тем не менее повторюсь — так уж и быть, для особо невнимательных.

dataSource.setDriverClass — это имя класса для драйвера JDBC. У нас PostgreSQL, вот и имя драйвера соответствует.
dataSource.setJdbcUrl — URL, параметры для подключения к POstgreSQL — IP-адрес, порт и имя базы данных
dataSource.setUser — имя пользователя
dataSource.setPassword — его пароль
dataSource.setMaxPoolSize — максимальное количество подключений
Параметров там гораздо больше, но это вам для самостоятельной работы — покопайтесь, наверняка что-нибудь интересное найдете.
Скажу несколько слов о классе **ComboPooledDataSource**. И даже больше не о нем, а о его самом главном предке — интерфейсе **javax.sql.DataSource**. Этот интерфейс был введен именно для решения задачи пула соединений. Как сказано в документации, это “фабрика соединений”.

Теперь нам осталось только исправить класс **ConnectionBuilderFactory** — теперь вместо **SimpleConnectionBuilder** мы будем создавать **ComboConnectionBuilder** (вы задумались о reflection вместо изменения кода?)

```
package edu.javacourse.contact.dao;

public class ConnectionBuilderFactory
{
    public static ConnectionBuilder getConnectionBuilder() {
        return new ComboConnectionBuilder();
    }
}
```

Ну и наконец в классе **ContactDbDAO** я оказывается забыл исправить одну строку — создание экземпляра для реализации интерфейса **ConnectionBuilder** было неправильное — мы напрямую создавали **SimpleConnectionBuilder**. Теперь эта строка должна выглядеть вот так (вы найдете старый вариант без труда):

```
1 private ConnectionBuilder builder = ConnectionBuilderFactory.getConnectionBuilder();
```

Полный код проекта можно скачать по этой ссылке:
Список контактов

Мы проделали в этот раз большую и важную работу — переехали на унифицированное средство сборки Maven, познакомились с его основами и заодно узнали про пул коннектов. Надеюсь, что дальше будет еще интереснее и вы продолжите чтение.

И теперь нас ждет следующая статья: Потоки ввода-вывода.

Домашнее задание

Самостоятельно сделать еще один вариант пула коннектов на основе библиотеки Apache Commons DBCP.
Координаты в Maven-репозитории можно посмотреть по этой ссылке: Maven — Apache Commons DBCP
Вот вам кусочек кода для подсказки (это вырвано из интернета и не является окончательным вариантом) — дальше включайте мозг самостоятельно.

```
1 BasicDataSource ds = new BasicDataSource();
2 ds.setDriverClassName("com.mysql.Driver");
3 ds.setUrl("jdbc:mysql://localhost/test");
4 ds.setUsername("root");
5 ds.setPassword("password");
6 ds.setMinIdle(5);
7 ds.setMaxIdle(10);
8 ds.setInitialSize(5);
9 ds.setMaxOpenPreparedStatements(100);
```

Удачи.

7 comments to *Переезжаем на Maven*

Октябрь 25, 2017 at 10:24
Renat says:

Отличный сайт! Статьи интересные и полезные.

Reply

Декабрь 28, 2017 at 11:10
Eugene says:

Долго не работало. После mvn package, mvn clean package, mvn compile получалось

BUILD FAILURE

Error injecting: org.apache.maven.plugin.resources.ResourcesMojo
java.lang.NoClassDefFoundError: org/apache/maven/shared/filtering/MavenFilteringException

Удалось победить добавлением в pom этого

org.apache.maven.plugins
maven-resources-plugin
3.0.2

Иначе использовалась версия 2.6
maven-resources-plugin:2.6:resources (default-resources)

Reply

Декабрь 28, 2017 at 12:32
admin says:

А можно уточнить, в каком месте случилась проблема ? Я нигде не использовал resource-plugin.

Reply

- 
  Февраль 6, 2018 at 11:58
  Евгений says:

  Спасибо за ваши уроки, очень познавательно.
  Немного переработал ваш урок на встраиваемую базу Apache Derby (Java DB), чтобы не возиться с настройкой PostgreSQL. Возможно кому-нибудь пригодится: https://github.com/golubtsoff/contact

  Reply

- 
  Сентябрь 5, 2018 at 17:33
  Владимир says:

  Здравствуйте. Собранный JAR не запускается и весит 21кб(IDEA собирает его на пару мегабайт). В нём нет Connector SQL, хотя в POM указано:

  mysql
  mysql-connector-java
  8.0.12

  Reply

  - 
    Сентябрь 5, 2018 at 18:06
    admin says:

    А можно показать строку запуска ? То, что в Maven указан JAR — это не значит, что не надо его указывать при запуске.
    Т.е. должно быть что-то вроде:

    java -cp .;mysql-connector-java.jar MyClassWithMain

Reply

Сентябрь 6, 2018 at 12:31
Владимир says:

Получилось уже)

Reply

Leave a reply

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя *

E-mail *

Сайт

девять × = девять

Add comment