**Создаем Maven-проект**

Прежде чем запускать Maven, убедитесь, что ваш компьютер подключен к интернету.

Большинство руководств предложат вам запустить специальный Maven-плагин — **archetype** с целью **generate**.

Нужно перейти в директорию где будет создан проект и запустить команду:

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

Важные элементы файла pom.xml:

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>edu.javacourse</groupId>

<artifactId>sample</artifactId>

<version>1.0-SNAPSHOT</version>

</project>

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

**modelVersion** — этот элемент говорит о версии pom.xml.

**groupId** — это обобщающее имя группы проектов, которые могут быть связаны неким общим функционалом.

**artifactId** — уникальный идентификатор проекта в данной группе

**version** — версия проекта.

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.

### Начальные команды Maven

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 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** —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..  
**verify** — проверка через интеграционные тесты.

**install** — установка в локальный репозиторий.  
**deploy** — установка на удаленный репозиторий.

Вторая группа состоит 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, перейти в директорию полученного файла и для примера запустим цикл до фазы **package**.

mvn package

**Директории в созданном файле:**

**src** — это директория, где находятся файлы проекта — в основном это файлы java. 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.

Если заглянуть в директорию **target**, то там можно будет увидеть файлы:

classes

maven-archiver

maven-status

**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**.

Для очистки проекта — 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.

Фазы можно совмещать — не надо набирать сначала **mvn clean** и потом **mvn package**. Можно сразу набрать вот так:

**mvn clean package**

Для запуска установки: **mvn install**

После запуска смотрим в наш локальный репозиторий, к примеру:

**Installing** *здесь указывается путь к репозитории файла* **to** *здесь указывается путь к репозитории куда установился проект*

Так можно найти путь к проекту

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.

### Подключение внешних библиотек

### 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**.

В каталоге **src/main** может находится не только каталог **java**, можно добавить 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. 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**.

**Разбор файла pom.xml:**

<?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.  
Принцип работы Maven — на каждую фазу существует плагин, для фазы **compile** мы используем плагин с именем (его artifactId) **maven-compiler-plugin**. Версию можно не указывать.  
Если зайти в наш локальный репозиторий, то можно найти этот плагин — внутри репозитория его путь будет такой: **org/apache/maven/plugins/maven-compiler-plugin**.

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 плагина 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.

**Запуск проекта из Maven:**

Перед тем, как запустить 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 — надо использовать плагин. 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.

**Запустить приложение:**

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

**exec:java - э**то 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.

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**

Если все до этого было в порядке, то наша программа запустится. Но нас постигнет разочарование — окошко не сможет показать данные из таблиц, 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**, **artifactId** и **version**. Но 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 - в удаленном репозитории. Как найти параметры - 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 странице вы можете увидеть полный список версий артефакта **postgresql**. Выбрать нужную версию, 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.

**Корпоративный репозиторий**

Прежде чем перейти 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**.

**Пул соединений:**

Как я уже говорил в статье [Список контактов — работаем с БД](http://java-course.ru/begin/contact_db/), 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 пул, используете его 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()**, 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.   
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** - это Connection Pool для JDBC 3.0.Документация может быть получена по адресу [c3p0 — JDBC3 Connection and Statement Pooling](http://www.mchange.com/projects/c3p0/) а сам проект находится здесь: [c3p0:JDBC DataSources/Resource Pools](https://sourceforge.net/projects/c3p0/)

Но мы собираемся подключать это в виде зависимости Maven. Так и сделаем.  
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):

<?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>

        <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**:

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();

    }

}

Ключевую роль тут играет класс **ComboPooledDataSource, э**то класс библиотеки. В конструкторе я его создал и проинициализировал.

**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