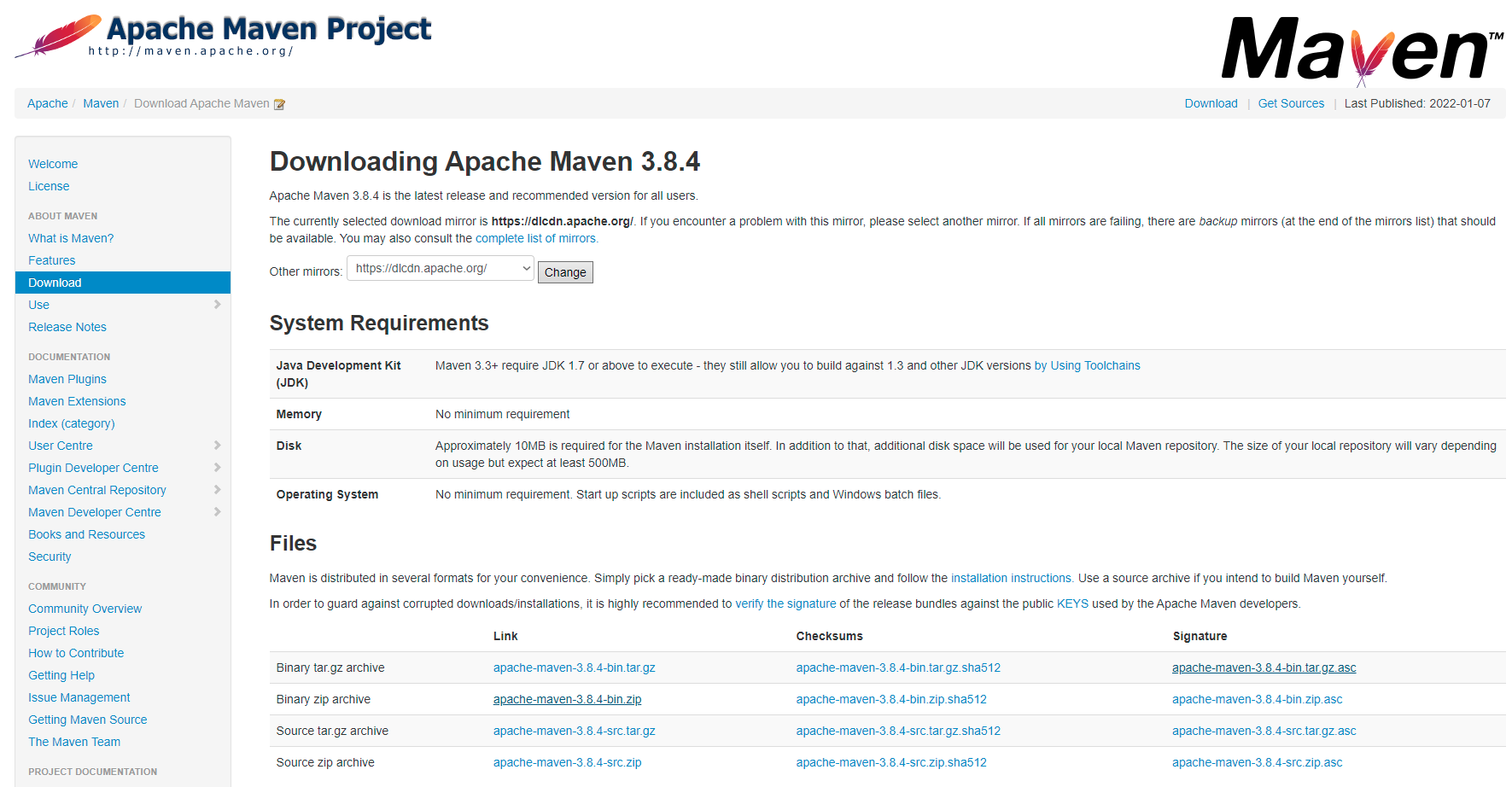
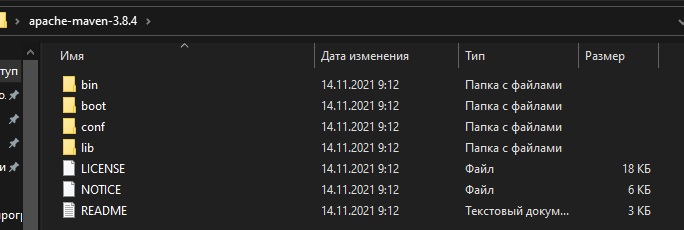
Задание №1 Сборка проекта на основе Maven

1. Ps создайте папочку maven, куда будем сгружать собранные примеры, что бы не потерялись. Так же можете сохранять их в этот файл в виде текста из конфигурационных файлов.
2. *Установите maven. Настройте environment variables и проверьте, что maven настроен корректно.*

Скачайте maven архив binary с <http://maven.apache.org/download.cgi>



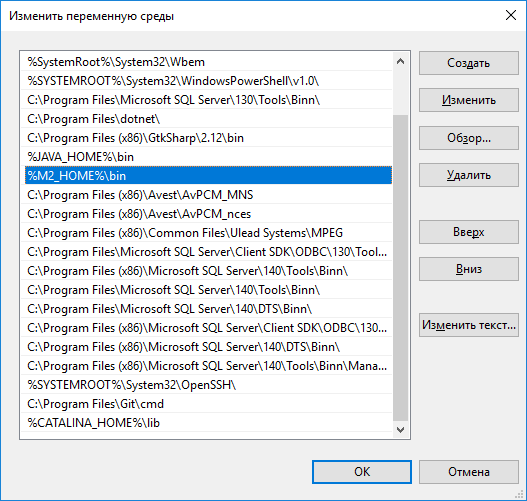
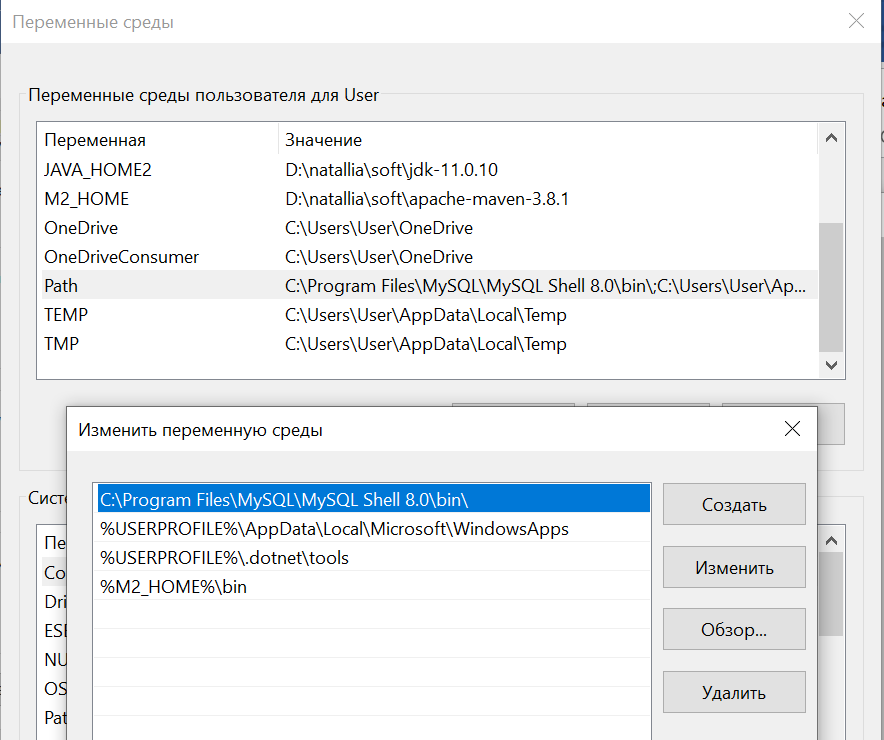
Обратите внимание что подбиться версия JDK 1.7 и выше. После загрузки вы получите zip архив около 10 Мб.



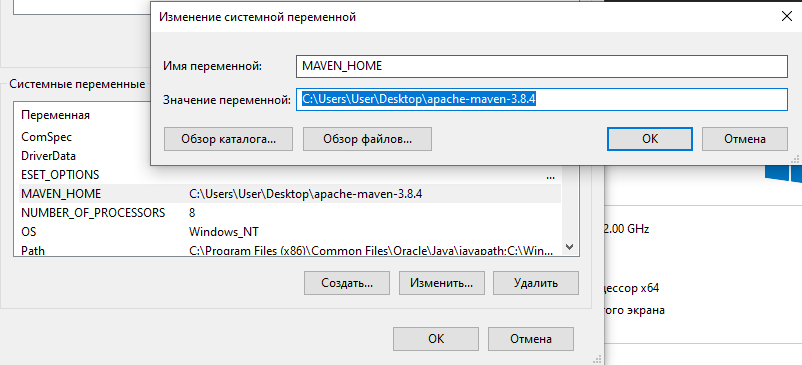
Maven устанавливается просто копированием в нужную директорию — никакого инсталлера нет. Распакуйте архив в любую папку. Для Windows, как правило, путь к папке не должен содержать пробелов.

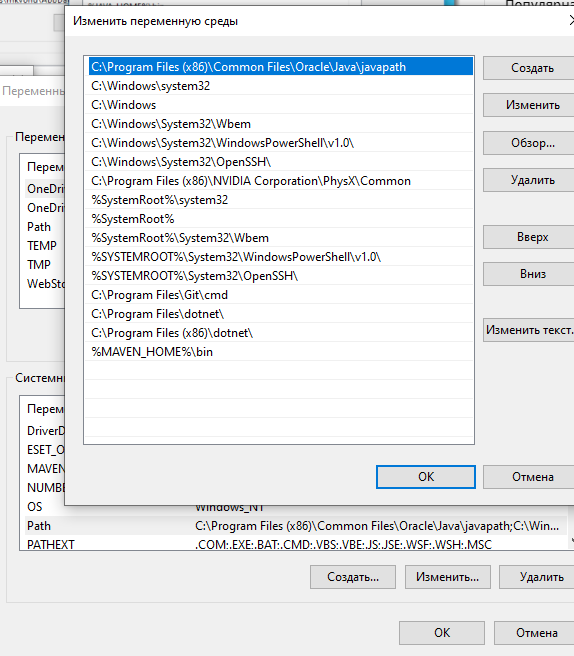
Для начала использования Maven необходимо настроить переменные среды Windows. MAVEN\_HOME (M2\_HOME) переменная должна быть установлена и

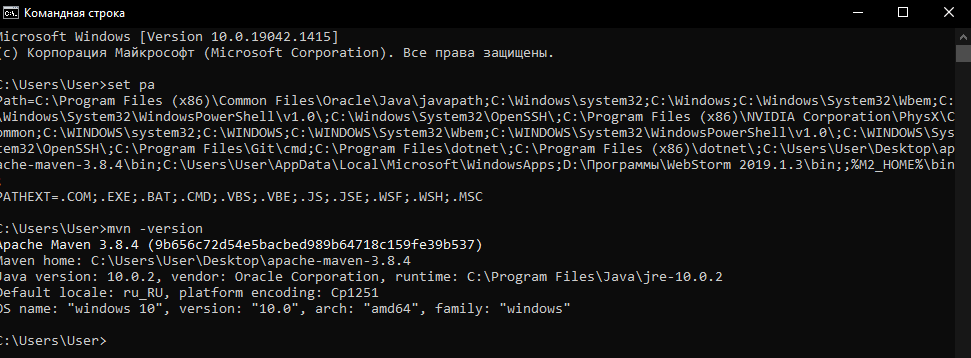
PATH переменная должна быть модифицирована для включения папки, откуда запускается Maven.



Убедитесь, что они установлены:



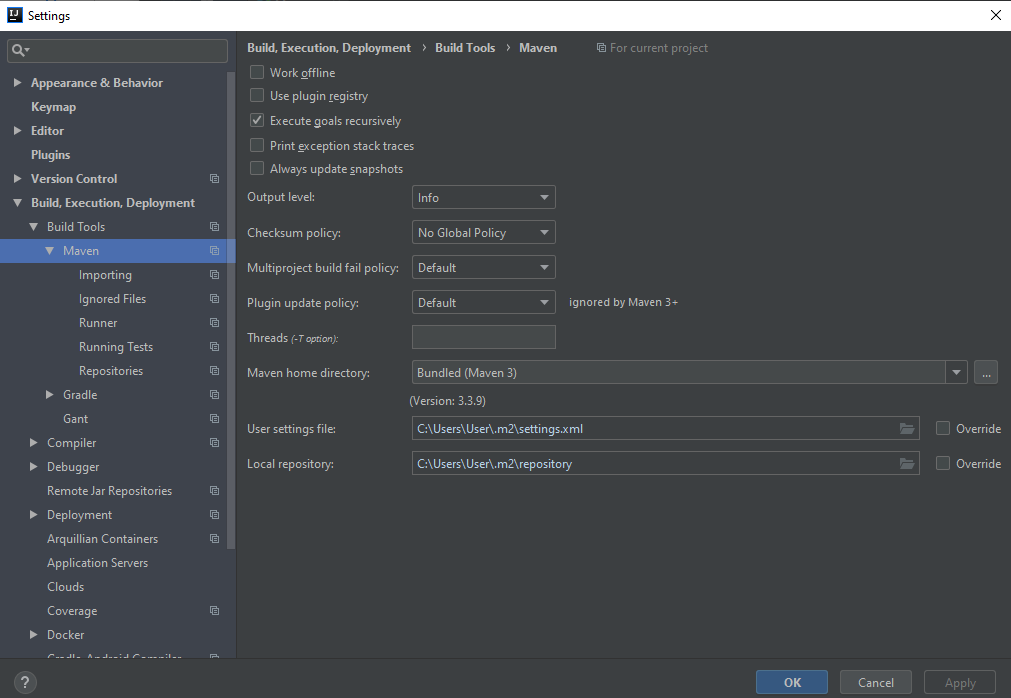




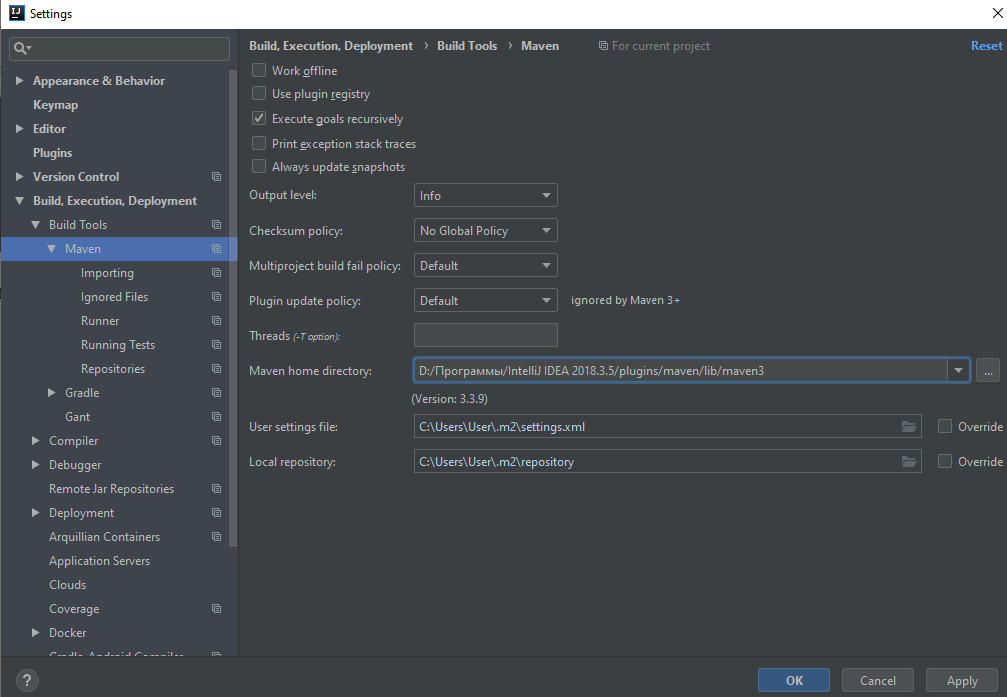
ApacheMaven, теперь, готов к использованию. Он, также, доступен для интеграции с IDE и другими программными средствами, предназначенными для разработки.mvn --version

1. *Проверка поддержки Maven*

Шаблон, на основе которого был создан проект включает поддержку maven

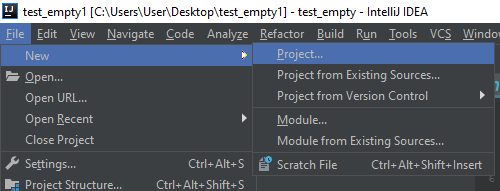


Если нужно будет поменять директорий maven или репозиторий это также можно сделать здесь.

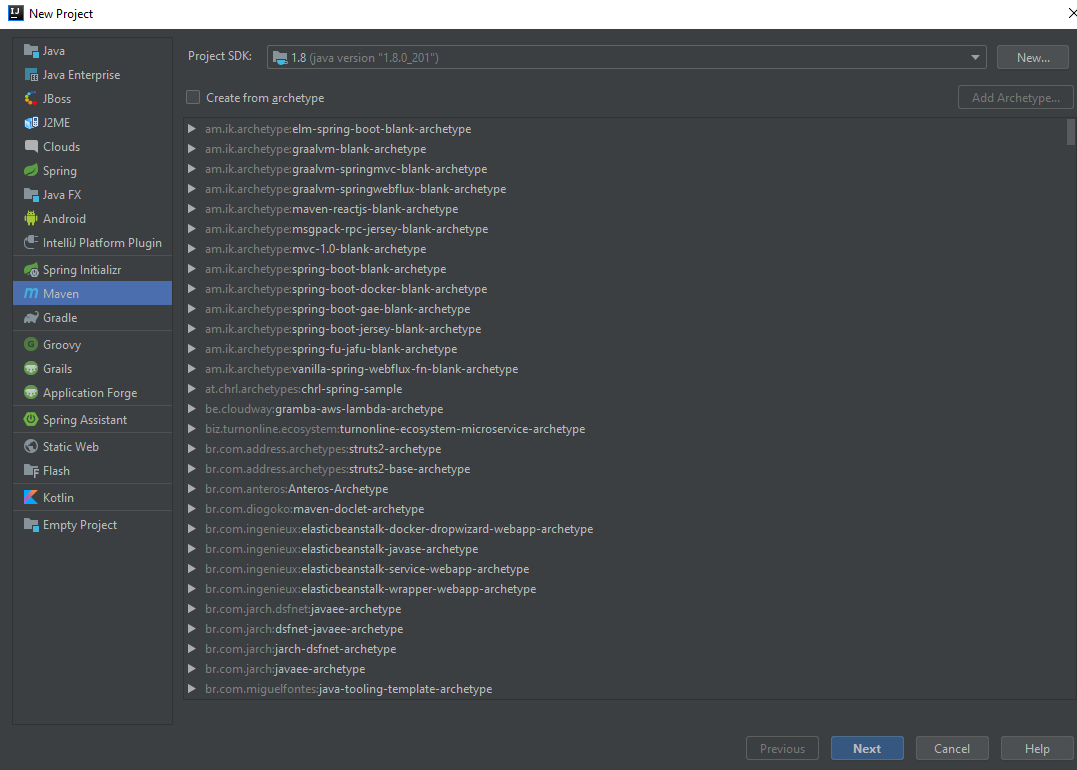


1. *Интеграция Maven*

Создайте проект с помощью Maven.



После чего



Выбираем Maven

Архетипы и прочее пока что не трогаем. Жмем далее.

Задаем имя проекта. А также спецификацию.

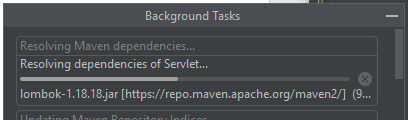
После чего должен быть автоматически создан проект с pom.xml.

<?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>test\_empty</groupId>  
 <artifactId>test\_empty</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <name>Servlet</name>  
 <packaging>war</packaging>  
   
 <properties>  
 <maven.compiler.target>1.8</maven.compiler.target>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <junit.version>5.7.0</junit.version>  
 </properties>  
  
</project>

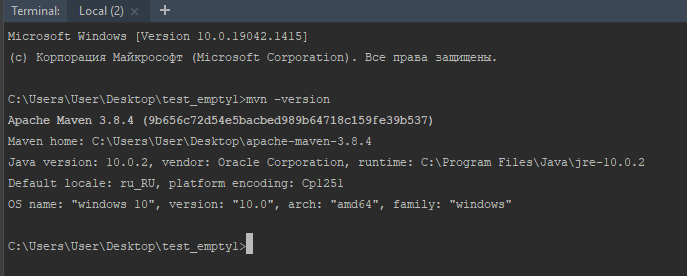
Изучите раздел c существующими зависимостями. У вас должно быть примерно так после добавления:

m

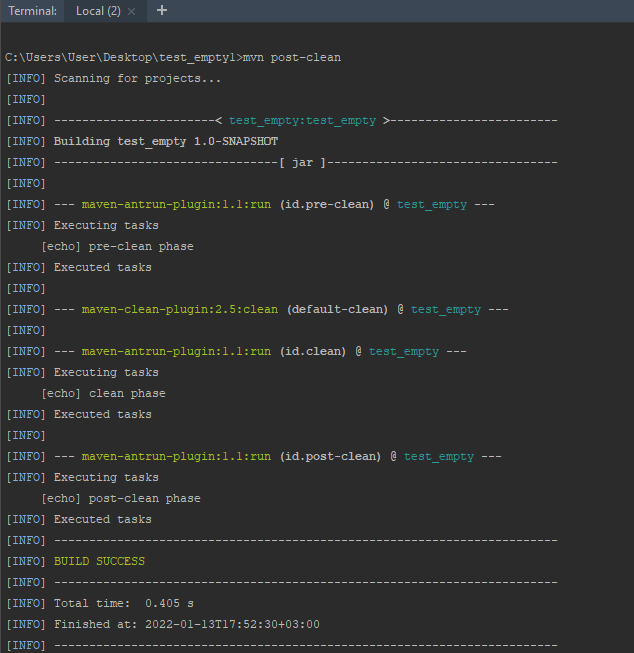
Далее вы сможете увидеть добавление зависимостей в проект.



Проверьте в консоли команду:



После чего выполните команду:



Maven начнет обработку и отображение всех фаз чистого жизненного цикла.

## **Жизненный цикл по умолчанию (или сборка)**

Это основной жизненный цикл Maven, который используется для создания приложения. Он имеет следующие 21 фазы.

|  |  |
| --- | --- |
| **Sr.No.** | **Фаза жизненного цикла и описание** |
| validate | Подтверждает, является ли проект корректным и вся ли необходимая информация доступа для завершения процесса сборки. |
| initialize | Инициализирует состояние сборки, например, различные настройки. |
| generate-sources | Включает любой исходный код в фазу компиляции. |
| process-sources | Обрабатывает исходный код (подготавливает). Например, фильтрует определённые значения. |
| generate-resources | Генерирует ресурсы, которые должны быть включены в пакет. |
| process-resources | Копирует и отправляет ресурсы в указанную директорию. Это фаза перед упаковкой. |
| compile | Комплирует исходный код проекта. |
| process-classes | Обработка файлов, полученных в результате компиляции. Например, оптимизация байт-кода Java классов. |
| generate-test-sources | Генерирует любые тестовые ресурсы, которые должны быть включены в фазу компиляции. |
| process-test-sources | Обрабатывает исходный код тестов. Например, фильтрует значения. |
| test-compile | Компилирует исходный код тестов в указанную директорию тестов. |
| process-test-classes | Обрабатывает файлы, полученные в результате компиляции исходного кода тестов. |
| test | Запускает тесты, используя приемлемый фреймворк юнит-тестирования (например, Junit). |
| prepare-package | Выполняет все необходимые операции для подготовки пакет, непосредственно перед упаковкой. |
| package | Преобразует скомпилированный код и пакет в дистрибутивный формат. Такие как JAR, WAR или EAR. |
| pre-integration-test | Выполняет необходимые действия перед выполнением интеграционных тестов. |
| integration-test | Обрабатывает и распаковывает пакет, если необходимо, в среду, где будут выполняться интеграционные тесты. |
| post-integration-test | Выполняет действия, необходимые  после выполнения интеграционных тестов. Например, освобождение ресурсов. |
| verify | Выполняет любые проверки для подтверждения того, что пакет пригоден и отвечает критериям качества. |
| install | Устанавливает пакет в локальный репозиторий, который может быть использован как зависимость в других локальных проектах. |
| deploy | Копирует финальный пакет (архив) в удалённый репозиторий для, того, чтобы сделать его доступным другим разработчикам и проектам. |

Теперь обновим pom файл и разберем, в чем он нам поможет сейчас:

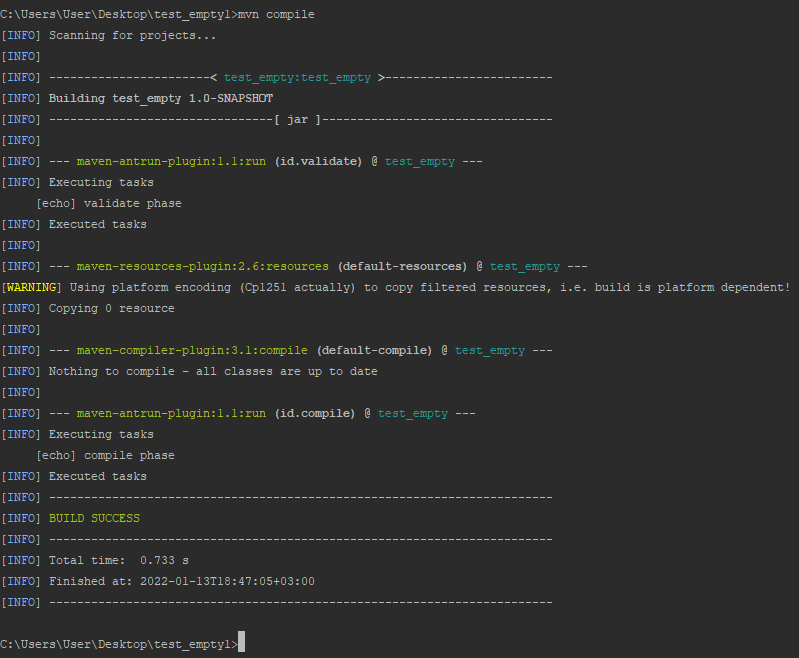
Необходимо уточнить два момента:

Когда мы выполняем команду Maven, например install, то будут выполнены фазы до install и фаза install.

Различные задачи Maven будут привязаны к различным фазам жизненного цикла Maven в зависимости от типа архива (JAR/WAR/EAR).

В следующем примере, мы привязываем задачу maven-antrun-plugin:run к нескольким фазам жизненного цикла сборки. Это также позволяет нам вызывать текстовые сообщения, отображая фазу жизненного цикла.

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-antrun-plugin</artifactId>  
 <version>1.1</version>  
 <executions>  
 <execution>  
 <id>id.validate</id>  
 <phase>validate</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <tasks>  
 <echo>validate phase</echo>  
 </tasks>  
 </configuration>  
 </execution>  
  
 <execution>  
 <id>id.compile</id>  
 <phase>compile</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <tasks>  
 <echo>compile phase</echo>  
 </tasks>  
 </configuration>  
 </execution>  
  
 <execution>  
 <id>id.test</id>  
 <phase>test</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <tasks>  
 <echo>test phase</echo>  
 </tasks>  
 </configuration>  
 </execution>  
  
 <execution>  
 <id>id.package</id>  
 <phase>package</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <tasks>  
 <echo>package phase</echo>  
 </tasks>  
 </configuration>  
 </execution>  
  
 <execution>  
 <id>id.deploy</id>  
 <phase>deploy</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <tasks>  
 <echo>deploy phase</echo>  
 </tasks>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>



## **Жизненный цикл Site**

Плагин Maven Site обычно используется для создания свежей документации, создания отчетов, развертывания сайта и т. Д. Он имеет следующие фазы:

Он включает в себя такие фазы:

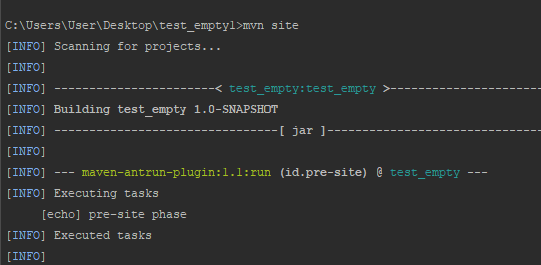
* pre-site
* site
* post-site
* site-deploy

Изменим наш pom файл:

В примере ниже мы прикрепляем задачу maven-antrun-plugin:run ко всем фазам жизненного цикла Site. Это позволяет нам вызывать текстовые сообщения для отображения фаз жизненного цикла.

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-antrun-plugin</artifactId>  
 <version>1.1</version>  
 <executions>  
 <execution>  
 <id>id.pre-site</id>  
 <phase>pre-site</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <tasks>  
 <echo>pre-site phase</echo>  
 </tasks>  
 </configuration>  
 </execution>  
  
 <execution>  
 (Добавить id.site)  
   
 </execution>  
  
 <execution>  
 (Добавить id.post-site)  
   
 </execution>  
  
 <execution>  
 (Добавить id.site-deploy)  
   
 </execution>  
  
 </executions>  
 </plugin>  
 </plugins>  
</build>

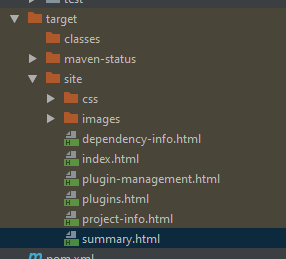
После чего выполним команду:



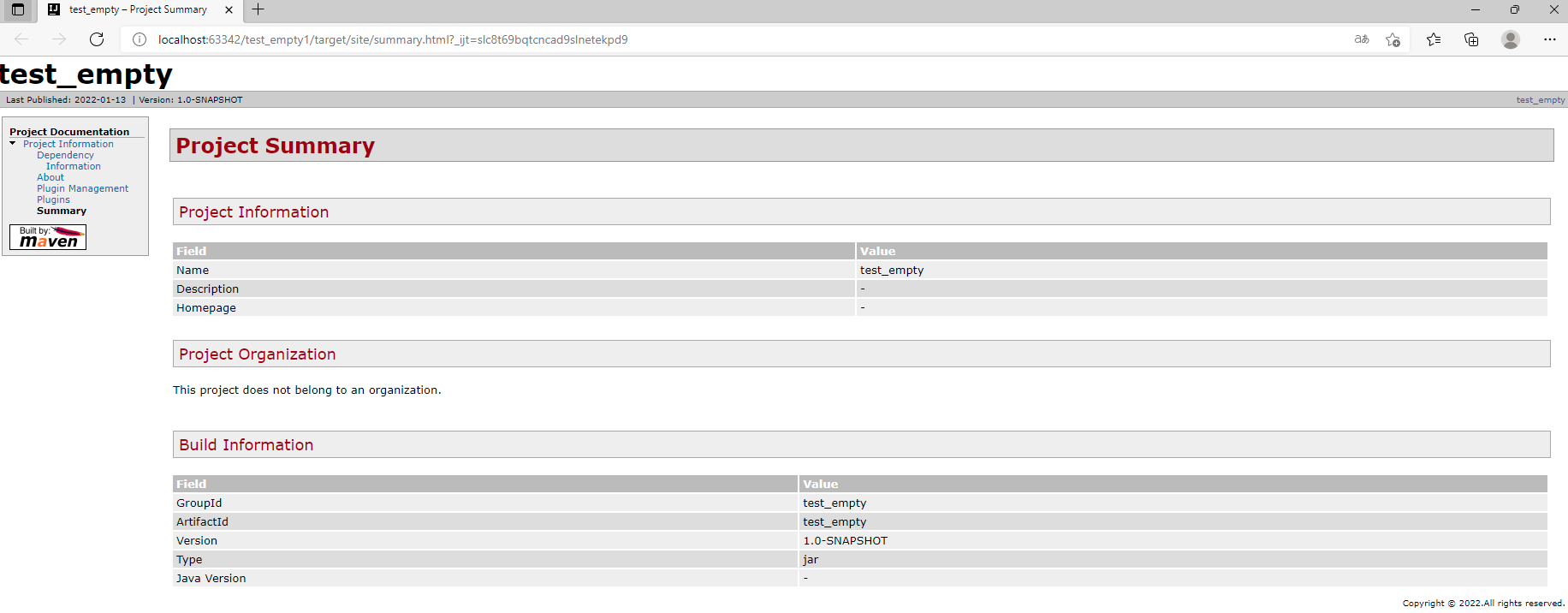
Если не работает, то исправим вот так:

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-site-plugin</artifactId>  
 <version>3.7.1</version>  
</plugin>  
<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-project-info-reports-plugin</artifactId>  
 <version>3.0.0</version>  
</plugin>

После чего в таргет выгрузилась информация по проекту:



Выбираем Нужную нам и жмякаем на просмотр:



Теперь можно просто и легко делать верификацию наших проектов.

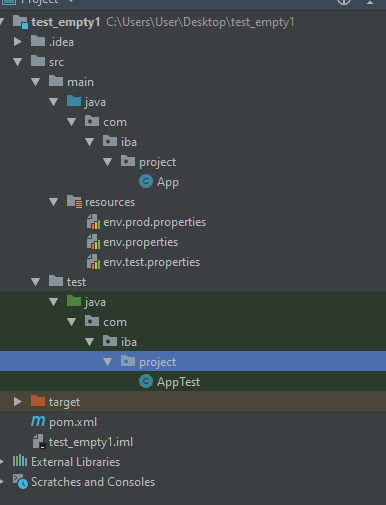
Перейдем к более сложным аспектам.

**Maven — Создание профилей**

Что такое профиль сборки?

Профиль сборки — это набор значений конфигурации, которые можно использовать для установки или переопределения значений по умолчанию сборки Maven. Используя профиль сборки, вы можете настроить сборку для различных сред.

Создадим структуру проекта:



Теперь в **каталоге src / main / resources** есть три специфичных для среды файла:

|  |  |
| --- | --- |
| **Sr.No.** | **Имя файла и описание** |
| 1 | **env.properties**  используется конфигурация по умолчанию, если профиль не указан. |
| 2 | **env.test.properties**  тестовая конфигурация при использовании тестового профиля. |
| 3 | **env.prod.properties**  производственная конфигурация при использовании профиля prod. |

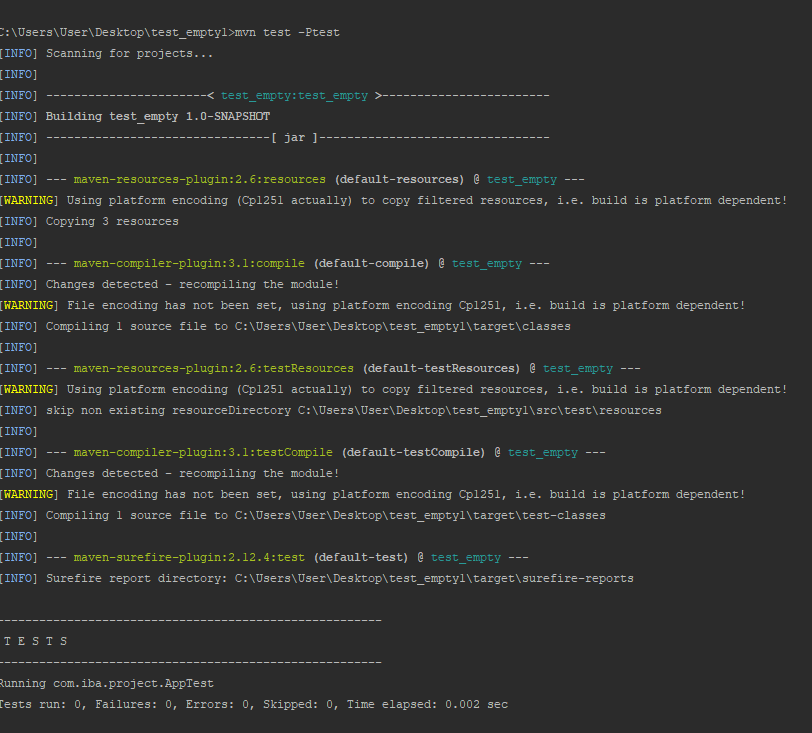
## Явная активация профиля

В следующем примере мы добавим maven-antrun-plugin: запустите цель, чтобы проверить фазу. Это позволит нам отображать текстовые сообщения для разных профилей. Мы будем использовать pom.xml для определения различных профилей и активировать профиль в командной консоли с помощью команды maven.

<profiles>  
 <profile>  
 <id>test</id>  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-antrun-plugin</artifactId>  
 <version>1.1</version>  
 <executions>  
 <execution>  
 <phase>test</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <tasks>  
 <echo>Using env.test.properties</echo>  
 <copy file="src/main/resources/env.test.properties"  
 tofile="${project.build.outputDirectory}/env.test.properties"/>  
 </tasks>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
 </build>  
 </profile>

В файл properties добавьте что-нибудь.

В результате наши в сбоку проекта будут включены данные настройки, указанные в тестовом файле или файле продакшина.



**Самостоятельно сделайте для продакшина и нормального билда проекта.**

**Для этого создайте ряд профилей: normal и prod.**

**Измените goal в каждом. Измените id. Измените echo.**

**Запуск каждого профиля начинается аналогично примеру выше, далее –P{id профиля}**

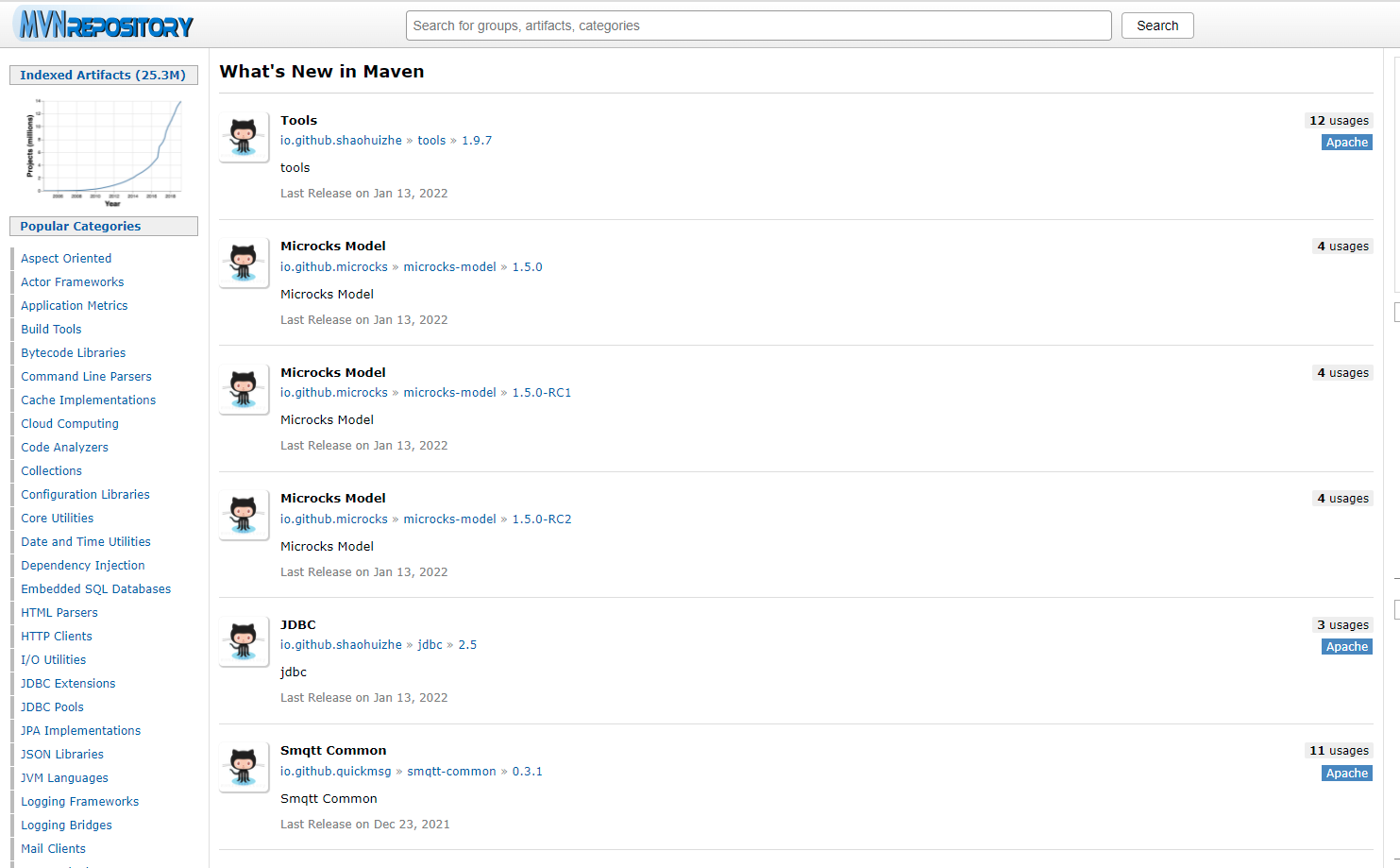
# Репозитории.

При работе с Maven под репозиторием мы понимаем директорию, где хранятся все JAR, библиотеки, плагины и любые артефакты, которыми Maven может воспользоваться.

Существует три типа репозиториев Maven:

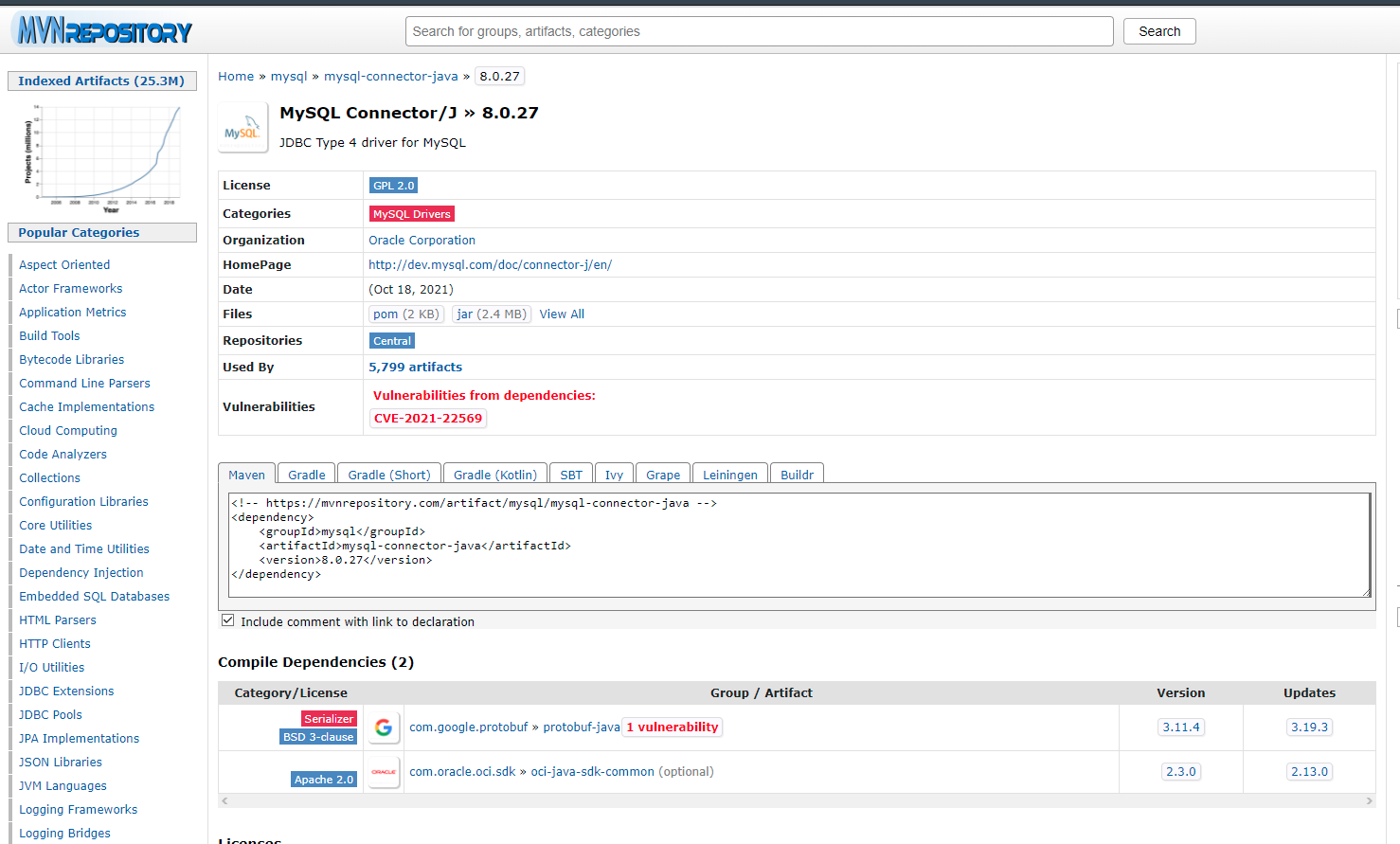
* **локальные (local)**
* **центральные (central)**
* **удалённые (remote)**

https://mvnrepository.com



Выбираем необходимую зависимость.

Далее:



Можем скопировать и использовать в своем проекте.

Все библиотеки будут доступны нам в коде автоматически.

## Последовательность поиска зависимостей Maven

Когда мы выполняем команды сборки Maven, Maven начинает поиск библиотек зависимостей в следующей последовательности:

* **Шаг 1** — Поиск зависимости в локальном репозитории, если не найден, перейти к шагу 2, иначе выполнить дальнейшую обработку.
* **Шаг 2** — Поиск зависимости в центральном репозитории, если не найдено и упоминается / упоминается удаленный репозиторий / репозитории, затем перейдите к шагу 4. В противном случае он загружается в локальный репозиторий для дальнейшего использования.
* **Шаг 3** — Если удаленный репозиторий не был упомянут, Maven просто останавливает обработку и выдает ошибку (Невозможно найти зависимость).
* **Шаг 4** — Поиск зависимости в удаленном репозитории или репозиториях, если он найден, он загружается в локальный репозиторий для дальнейшего использования. В противном случае Maven останавливает обработку и выдает ошибку (Невозможно найти зависимость).

# Плагины.

Если говорить в целом, то Maven – это фреймворк, который выполняет плагины. В этом фреймворке каждая задача, по сути своей, выполняется с помощью плагинов.

Плагины Maven использутся для:

* **создания jar – файла**
* **создания war – файла**
* **компиляции кода файлов**
* **юнит-тестирования кода**
* **создание отчётов проекта**
* **создание документации проекта**

**Типы плагинов**

Существует два типа плагинов в Maven:

* **Плагины сборки**Выполняются в процессе сборки и должны быть конфигурированны внутри блока <build></build> файла **pom.xml**
* **Плагины отчётов**Выполняются в процесса генерирования сайта и должны быть конфигурированны внутри блока <reporting></reporting> файла **pom.xml**.

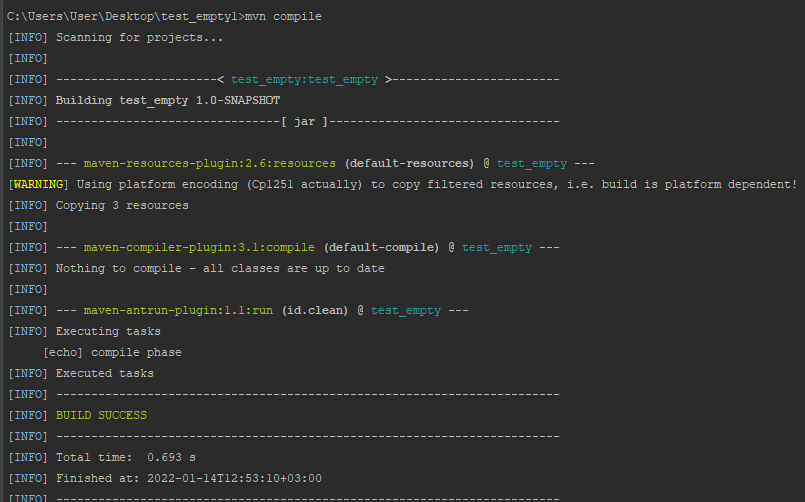
Вот список, наиболее используемых плагинов:

* **clean**Очищает цель после сборки. Удаляет директорию target.
* **compiler**Компилирует исходные Java файлы.
* **surefire**Запускает тесты JUnit. Создаёт отчёты о тестировании.
* **jar**Собирает JAR файл текущего проекта.
* **war**Собирает WAR файл текущего проекта.
* **javadoc**Генерирует Javadoc проекта.
* **antrun**Запускает набор задач **ant**из любой указанной фазы.

Для понимания того, как это работает на практике, рассмотрим следующий пример.

<profiles>  
 <profile>  
 <id>test</id>  
 <activation>  
 <file>  
 <missing>target/generated-sources/some/dir/com/iba/maven</missing>  
 </file>  
 </activation>  
 </profile>  
 </profiles>  
 <dependencies>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>3.8.1</version>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-antrun-plugin</artifactId>  
 <version>1.1</version>  
 <executions>  
 <execution>  
 <id>id.clean</id>  
 <phase>compile</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <tasks>  
 <echo>compile phase</echo>  
 </tasks>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
 </build>  
</project>

Выполняем команду:

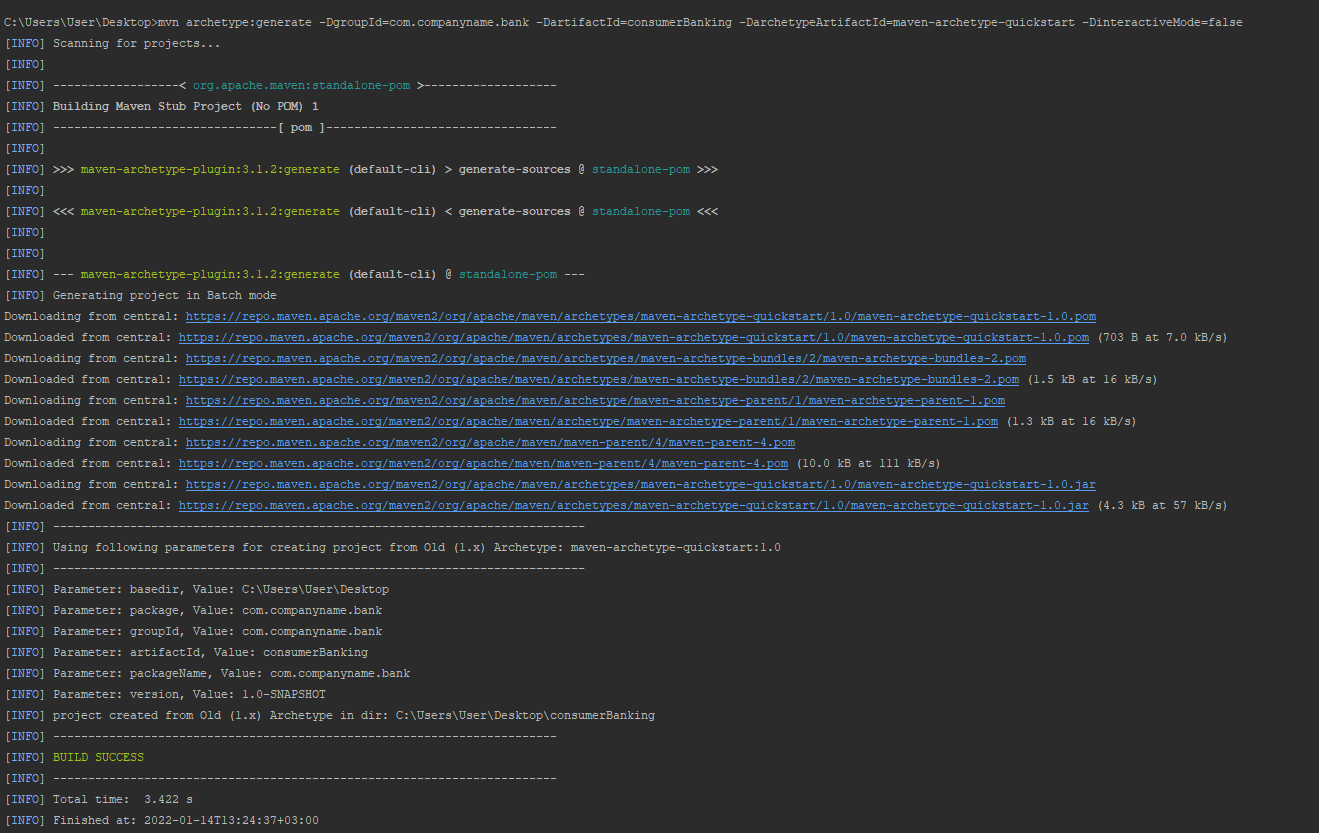


**Добавьте Фазу(executions) compile.**

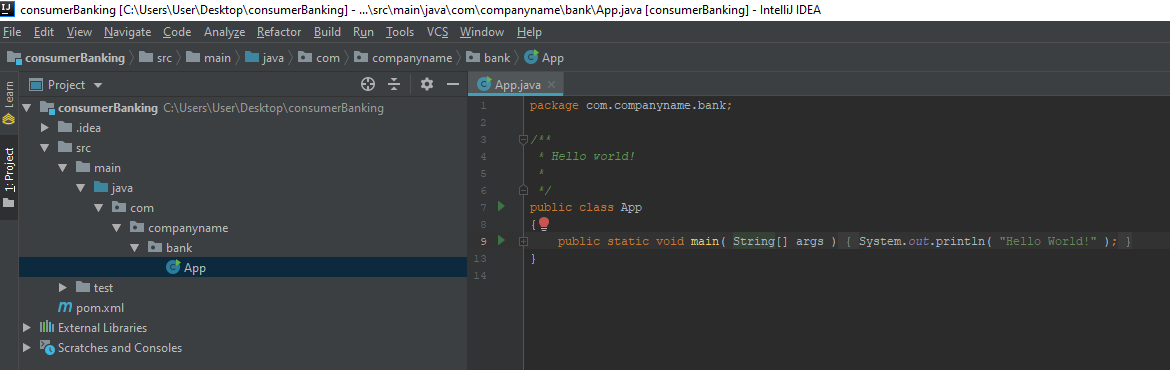
**Вызовите через консоль разные фазы и сравните результат в теге echo.**

**Создание проекта**

Создадим проект на основе основных зависимостей. Откроем консоль и используем следующую команду:



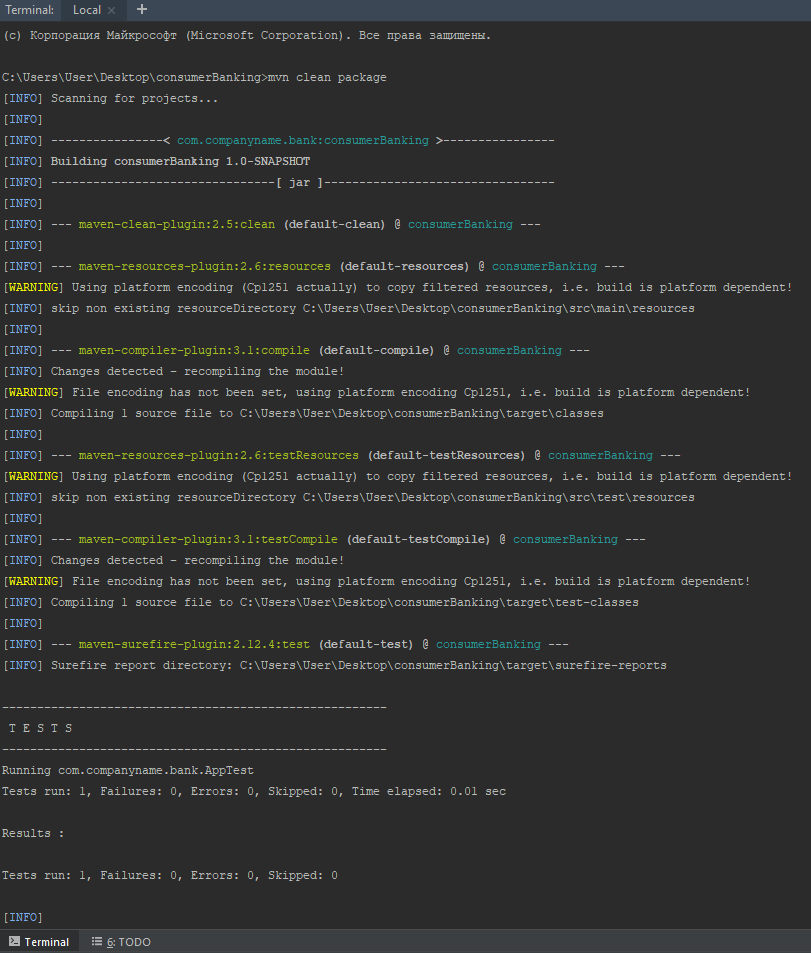
После чего перейдем в наш проект:



**Обратите внимание, что помимо класса, наш сборщик, создал файл тестирования проекта. Намекая нам о там, что все должно быть покрыто модульными тестами, о которых мы узнаем в середине нашего обучения.**

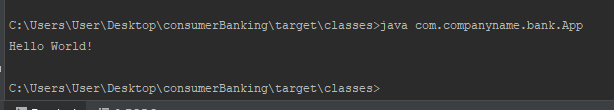
**Сборка проекта:**

Зайдем в консоль и выполним команду:



Обратите внимание, что тесты выполнены только потому, что в POM файле есть зависимость для тестирования!

Далее перейдите в папку classes и выполните команду:



Наше приложение запустилось.

Откройте билд проекта и посмотрите, что специфического вы сможете обнаружить внутри.

Что можете сказать про последний собранный архив?

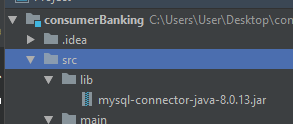
# Внешние зависимости.

Ранее мы уже изучали управление зависимостями с помощью репозиториев. Но что, если необходимые файлы не найдены ни в центральном, ни на удалённом репозитории? Для решения этой проблемы используются внешние зависимости.

Рассмотрим такой пример.

Добавим в нам проект в папку **src**директорию **lib**.

Добавьте в эту директорию любой jar файл.



После чего давайте изменим наш пом файл

<profiles>  
 <profile>  
 <id>test</id>  
 <activation>  
 <file>  
 <missing>target/generated-sources/some/dir/com/iba/maven</missing>  
 </file>  
 </activation>  
 </profile>  
 </profiles>  
 <dependencies>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>3.8.1</version>  
 <scope>test</scope>  
 </dependency>  
  
 <!-- External dependency -->  
 <dependency>  
 <groupId>mysql-connector-java-8.0.13.jar</groupId>  
 <artifactId>mysql-connector-java-8.0.13.jar</artifactId>  
 <scope>system</scope>  
 <version>1.0</version>  
 <systemPath>C:/Users/User/Desktop/consumerBanking/src/lib/mysql-connector-java-8.0.13.jar</systemPath>  
 </dependency>  
  
 </dependencies>  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-antrun-plugin</artifactId>  
 <version>1.1</version>  
 <executions>  
 <execution>  
 <id>id.clean</id>  
 <phase>compile</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <tasks>  
 <echo>compile phase</echo>  
 </tasks>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
 </build>  
</project>

После наших правок, проект сможет обработать нашу библиотеку.

# Шаблоны проектов.

С помощью концепции **Архитипов** Maven обеспечивает большое количество различных шаблонов проектов (более 600).

**Архитип**

Архитип – это плагин Maven, задача которого состоит в создании структуры проекта по определённому шаблону. Рассмотри архитип quickstart и создадим с его помощью java-приложение.

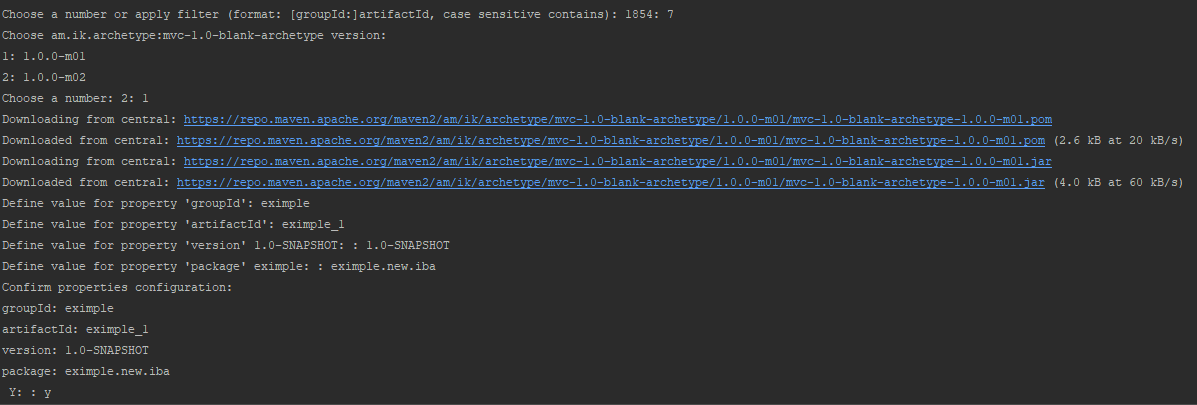
Попробуем еще вариант создания архетипа:

Перейдем в консоль и выполним команду mvn archetype:generate

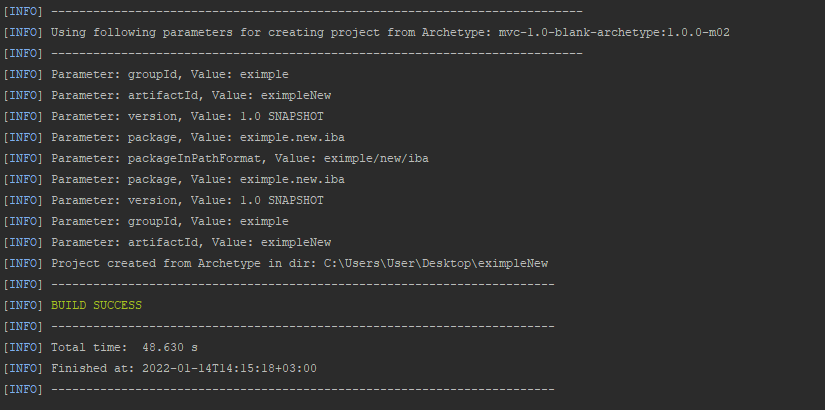
Далее нажимаем цифру 7.

После чего выбираем вид архетипа: 1.

Далее нужно дать метаданные нашему проекту.



В конце подтверждаем Y

В результате 

С глобального репозитория мы получили часть проекта:

