Фреймворк Apache Maven

Apache Maven предназначен для автоматизации процесса сборки проектов на основе описания их структуры в файле на языке POM (Project Object Model), который является подмножеством формата XML. *maven* позволяет выполнять компиляцию кодов, создавать дистрибутив программы, архивные файлы jar/war и генерировать документацию. Простые проекты **maven** может собрать в командной строке. Название программы **maven** вышло из языка идиш, смысл которого можно выразить как «собиратель знания».

В отличие от ant с императивной сборкой проекта, **maven** обеспечивает декларативную сборку проекта. То есть, в файле описания проекта содержатся не отдельные команды выполнения, а спецификация проекта. Все задачи по обработке файлов *maven* выполняет посредством их обработки последовательностью встроенных и внешних плагинов.

Если описать *maven* на одной странице сайта, да еще привести примеры использования, то содержимое будет «нечитабельным» - это слишком большой объём информации для представления далеко не всех его возможностей. Поэтому описание разнесено по нескольким страницам. На этой странице представлено общее описание *maven*. На странице Примеры проектов maven описано использование *maven* для разнотипных проектов. Отдельными страницами представлено «применение maven в IDE Eclipse» (в разработке) и «плагины maven» (в разработке).

Инсталляция maven

Последнюю версию *maven* можно скачать в виде zip-архива со страницы загрузки на официальном сайте. После этого необходимо выполнить несколько шагов :

- распаковать архив в инсталляционную директорию. Например в директорию C:\maven-x.y.z в Windows или /opt/maven-x.y.z в Linux
- установить переменную окружения M2_HOME :
  - в Windows кликните правой кнопкой мыши на «Мой компьютер» и откройте окно Свойства/«Дополнительные параметры»/«Переменные среды»/«Системные переменные», в котором добавьте «M2_HOME» = "C:\maven-x.y.z\";
  - в Linux можно добавить строку «export M2_HOME=/opt/maven-x.y.z» в файл /etc/profile;
- Внесите изменения в переменную окружения PATH :
  - в Windows в переменную PATH добавьте строку %M2_HOME%\bin;
  - в Linux можно добавить строку «export PATH=$PATH:$M2_HOME/bin» в файл /etc/profile;

Чтобы убедиться, что **maven** установлен, необходимо в командной

строке ввести следующую команду :

mvn -–version

Должна появиться информация о версиях **maven**, *jre* и операционной системе типа :

Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T19:41:47+03:00)
Maven home: C:\apache-maven-3.3.9
Java version: 1.7.0_79, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.7.0_79\jre
Default locale: ru_RU, platform encoding: Cp1251
OS name: "windows 8.1", version: "6.3", arch: "amd64", family: "windows"

При инсталляции **maven**'а будет создан локальный репозиторий в вашей личной папке ${user.home}\.m2\repository. После этого можно считать, что **maven** готов к работе и можно приступать к созданию проектов сборки приложений.

Одной из привлекательных особенностей *maven*'а является справка online, работоспособность которой можно проверить после инсталляции. К примеру справку по фазе компиляции можно получить следующей командой :

mvn help:describe -Dcmd=compile

В результате Вы увидите следующую справочную информацию :

```
[INFO] 'compile' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-compiler-plugin:3.1:compile

It is a part of the lifecycle for the POM packaging 'jar'.
This lifecycle includes the following phases:
* validate: Not defined
* initialize: Not defined
* generate-sources: Not defined
* process-sources: Not defined
* generate-resources: Not defined
* process-resources: \
      org.apache.maven.plugins:maven-resources-
plugin:2.6:resources
*       compile:       org.apache.maven.plugins:maven-compiler-
plugin:3.1:compile
* process-classes: Not defined
* generate-test-sources: Not defined
* process-test-sources: Not defined
* generate-test-resources: Not defined
* process-test-resources: \
      org.apache.maven.plugins:maven-resources-
plugin:2.6:testResources
* test-compile: \
      org.apache.maven.plugins:maven-compiler-
plugin:3.1:testCompile
* process-test-classes: Not defined
```

```
      *         test:         org.apache.maven.plugins:maven-surefire-
plugin:2.12.4:test
      * prepare-package: Not defined
      * package: org.apache.maven.plugins:maven-jar-plugin:2.4:jar
      * pre-integration-test: Not defined
      * integration-test: Not defined
      * post-integration-test: Not defined
      * verify: Not defined
      *        install:        org.apache.maven.plugins:maven-install-
plugin:2.4:install
      * deploy: org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy

      [INFO] ------------------------------------------------------------
--------------
      [INFO] BUILD SUCCESS
      [INFO] ------------------------------------------------------------
--------------
      [INFO] Total time: 0.742 s
      [INFO] Finished at: 2016-09-19T22:41:26+04:00
      [INFO] Final Memory: 7M/18M
      [INFO] ------------------------------------------------------------
--------------
```

Репозитории проекта, repositories

Под репозиторием (*repository*) понимается, как правило, внешний центральный репозиторий, в котором собрано огромное количество наиболее популярных и востребованных библиотек, и локальный репозиторий, в котором хранятся копии используемых ранее библиотек.

Дополнительные репозитории, необходимые для сборки проекта, перечисляются в секции *<repositories>* проектного файла *pom.xml* :

```
<project>
    ...
    <repositories>
        <repository>
            <id>repo1.maven.org</id>
            <url>http://repo1.maven.org/maven2</url>
        </repository>
    </repositories>
    ...
</project>
```

Можно создать и подключать к проектам свой репозиторий, содержимое которого можно полностью контролировать и сделать его доступным для ограниченного количества пользователей. Доступ к содержимому репозитория можно ограничивать настройками безопасности сервера так, что код ваших проектов не будет доступен из вне. Существуют несколько реализаций серверов - репозиториев maven; к наиболее известным относятся artifactory, continuum, nexus.

Таким образом, репозиторий - это место, где хранятся файлы jar, pom, javadoc, исходники и т.д. В проекте могут быть использованы :

- центральный репозиторий, доступный на чтение для всех пользователей в интернете;

- внутренний «корпоративный» репозиторий - дополнительный репозиторий группы разработчиков;
- локальный репозиторий, по умолчанию расположен в ${user.home}/.m2/repository - персональный для каждого пользователя.

Для добавления, к примеру, библиотеки *carousel-lib.jar* в локальный репозиторий можно использовать команду *mvn install* (команда должна быть однострочной) :

mvn install:install-file \

```
-Dfile=${FILE_PATH}/carousel-lib.jar \
-DgroupId=ru.carousel \
-DartifactId=carousel-lib \
-Dversion=1.0 \
-Dpackaging=jar \
-DgeneratePom=true
```

В локальном репозитории «.m2» *maven* создаст директорию ru/carousel, в которой разместит данную библиотеку и создаст к ней описание в виде pom.xml.

Репозиторий можно также разместить внутри проекта. Описания процесса создания и размещения репозитория внутри проекта с примером можно прочитать здесь.

Собстенные наработки можно подключить как системную зависимость.

Терминология maven

В **maven** используется свой набор терминов и понятий. Ключевым понятием *maven* является артефакт (*artifact*) — это, по сути, любая библиотека, хранящаяся в репозитории, к которой можно отнести зависимость или плагин.

Зависимости (*dependencies*) представляют собой библиотеки, которые непосредственно используются в проекте для компиляции или тестирования кода.

При сборке проекта или для каких-то других целей (deploy, создание файлов проекта для Eclipse и др.) *maven* использует плагины (*plugin*).

Еще одним важным понятием **maven** проекта является архетип (*archetype*) - это некая стандартная компоновка каталогов и файлов в проектах различного типа (web, maven, swt/swing-проекты и прочие). Иными словами *maven* знает, как построить структуру проекта в соответствии с его архетипом.

Архетипы maven

Количество архетипов у *maven*'а огромно, «на разный вкус». Как правильно выбрать нужный, чтобы создать архитектуру будущего проекта? Просматривать в консоли не очень удобно, тем более что их количество переваливает за 1500 (к примеру для версии maven 3.3.9 на моем компьютере их 1665). Поэтому можно скачать их в отдельный файл, а потом со всем этим

хозяйством разбираться. Для этого необходимо выполнить следующую команду :

mvn archetype:generate > archetypes.txt

В результате в файле archetypes.txt можно увидеть, что-то подобное

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
---------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------
---------
[INFO]
[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) >
                    generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) <
                    generate-sources @ standalone-pom <<<
[INFO]
[INFO]   maven-archetype-plugin:2.4:generate   (default-cli)   @
standalone-pom
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart \
          (org.apache.maven.archetypes:maven-archetype-
quickstart:1.0)
Choose archetype:
1: remote -> am.ik.archetype:maven-reactjs-blank-archetype
                                    (Blank Project for React.js)
2: remote -> am.ik.archetype:msgpack-rpc-jersey-blank-archetype
                                    (Blank Project for Spring Boot +
Jersey)
3: remote -> am.ik.archetype:mvc-1.0-blank-archetype
                                    (MVC 1.0 Blank Project)
4: remote -> am.ik.archetype:spring-boot-blank-archetype
                                    (Blank Project for Spring Boot)
5: remote -> am.ik.archetype:spring-boot-docker-blank-archetype
                                    (Docker Blank Project for Spring
Boot)

...
```

При выполнении данной команды *maven*, после скачивания информации в файл, ожидает поступления команды пользователя. Т.е., находится в ожидании интерактивного ввода команд по созданию проекта определенного типа. Если файл уже создан, то прервите выполнение команды двойным нажатием клавишам Ctrl+C.

Для создания простенького *maven* проекта «carousel» (карусель) необходимо выполнить следующую команду :

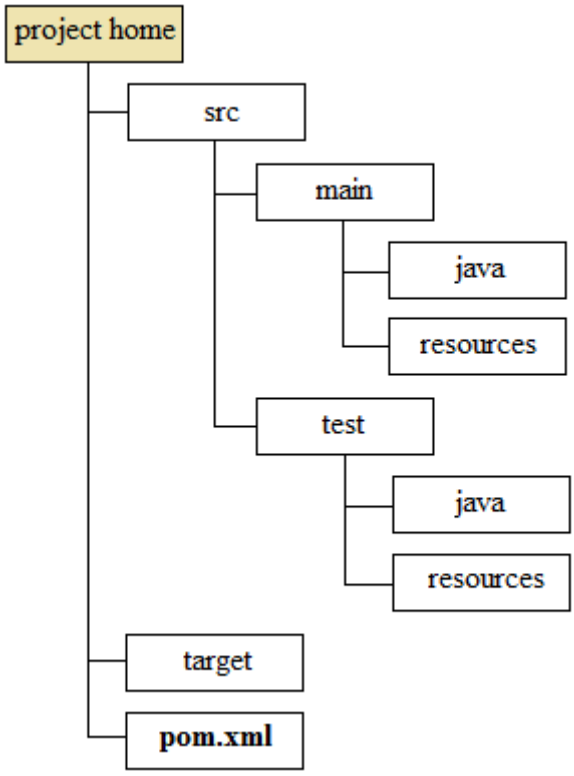
```
mvn archetype:generate \
    -DgroupId=ru.carousel \
    -DartifactId=carousel \
    -Dversion=1.0-SNAPSHOT \
    -DarchetypeArtifactId=maven-archetype-quickstart
```

На странице Примеры maven проектов подробно описываются

разнотипные **maven** проекты - проект консольного приложения без зависимостей, приложение с графическим интерфейсом и с зависимостями, web-приложение с фреймворком.

Архитектура простого maven проекта

Следующая структура показывает директории простого *maven* проекта.



Проектный файл *pom.xml* располагается в корне каталога.

- src: исходные файлы;
- src/main: исходные коды проекта;
- src/main/java: исходные java-файлы;
- src/main/resources: ресурсные файлы, которые используются при компиляции или исполнении, например properties-файлы;
- src/test: исходные файлы для организации тестирования;
- src/test/java: JUnit-тест-задания для автоматического тестирования;
- target: создаваемые в процессе работы maven'a файлы для сборки проекта.

В зависимости от типа приложения (консольное, с интерфейсом, web, gwt и т.д.) структура может отличаться. В директории *target* **maven** собирает проект (jar/war).

На официальном сайте **Apache Maven Project** можно получить дополнительную информацию об архетипах (Introduction to Archetypes).

Жизненный цикл maven проекта

Жизненный цикл **maven** проекта – это чётко определённая

последовательность фаз. Когда *maven* начинает сборку проекта, он проходит через определённую последовательность фаз, выполняя задачи, указанные в каждой из фаз. *Maven* имеет 3 стандартных жизненных цикла :

- clean — жизненный цикл для очистки проекта;
- default — основной жизненный цикл;
- site — жизненный цикл генерации проектной документации.

Каждый из этих циклов имеет фазы pre и post. Они могут быть использованы для регистрации задач, которые должны быть запущены перед и после указанной фазы.

Фазы жизненного цикла clean

- pre-clean;
- clean;
- post-clean.

Фазы жизненного цикла default

- validate - выполнение проверки, является ли структура проекта полной и правильной;
- generate-sources - включение исходного кода в фазу;
- process-sources - подготовка/обработка исходного кода; например, фильтрация определенных значений;
- generate-resources - генерирование ресурсов, которые должны быть включены в пакет;
- process-resources - копирование ресурсов в указанную директорию (перед упаковкой);
- compile - компиляция исходных кодов проекта;
- process-test-sources - обработка исходных кодов тестов;
- process-test-resources - обработка ресурсов для тестов;
- test-compile - компиляция исходных кодов тестов;
- test - собранный код тестируется, используя приемлемый фреймворк типа JUnit;
- package - упаковка откомпилированных классов и прочих ресурсов в дистрибутивный формат;
- integration-test - программное обеспечение в целом или его крупные модули подвергаются интеграционному тестированию. Проверяется взаимодействие между составными частями программного продукта;
- install - установка программного обеспечения в maven-репозиторий, чтобы сделать его доступным для других проектов;
- deploy - стабильная версия программного обеспечения копируется в удаленный maven-репозиторий, чтобы сделать его доступным для других пользователей и проектов;

Фазы жизненного цикла site

- pre-site;
- site;
- post-site;
- site-deploy;

Стандартные жизненные циклы могут быть дополнены функционалом с

помощью *maven-плагинов*. Плагины позволяют вставлять в стандартный цикл новые шаги (например, распределение на сервер приложений) или расширять существующие шаги.

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

mvn clean dependency:copy-dependencies package

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

Зависимости, dependency

Зависимость, эта связь, которая говорит, что для некоторых фаз жизненного цикла *maven* проекта, требуются некоторые артефакты. Зависимости проекта описываются в секции <*dependencies*> файла *pom.xml*. Для каждого используемого в проекте артефакта необходимо указать GAV (**g**roupId, **a**rtifactId, **v**ersion), где

- **groupId** - идентификатор производителя объекта. Часто используется схема принятая в обозначении пакетов Java. Например, если производитель имеет домен domain.com, то в качестве значения *groupId* удобно использовать значение *com.domain*. То есть, *groupId* это по сути имя пакета.
- **artifactId** - идентификатор объекта. Обычно это имя создаваемого модуля или приложения.
- **version** - версия описываемого объекта. Для незавершенных проектов принято добавлять суффикс SNAPSHOT. Например 1.0-SNAPSHOT.

Как правило информации GAV достаточно *maven*'у, для поиска указанного артефакта в репозиториях. Пример описания зависимости библиотеки JDBC для работы с БД Oracle.

```
<dependency>
   <groupId>com.oracle</groupId>
   <artifactId>ojdbc6</artifactId>
   <version>11.2.0.4</version>
</dependency>
```

Но иногда при описании зависимости требуется использовать необязательный параметр <classifier>. Следующий пример демонстрирует описание зависимости библиотеки *json-lib-2.4-jdk15.jar* с параметром *classifier*.

```
<dependency>
   <groupId>net.sf.json-lib</groupId>
   <artifactId>json-lib</artifactId>
   <version>2.4</version>
```

```
    <classifier>jdk15</classifier>
</dependency>
```

Более подробная информация о зависимостях и областях их действия, а также о способе построения транзитивных зависимостей в виде дерева представлена на странице dependency в maven-проекте.

Плагины, plugins

Maven базируется на plugin-архитектуре, которая позволяет использовать плагины для различных задач (test, compile, build, deploy и т.п). Иными словами, *maven* запускает определенные плагины, которые выполняют всю работу. То есть, если мы хотим научить *maven* особенным сборкам проекта, то необходимо добавить в *pom.xml* указание на запуск нужного плагина в нужную фазу и с нужными параметрами. Это возможно за счет того, что информация поступает плагину через стандартный вход, а результаты пишутся в его стандартный выход.

Количество доступных плагинов очень велико и включает разнотипные плагины, позволяющие непосредственно из *maven* запускать web-приложение для тестирования его в браузере, генерировать Web Services. Главной задачей разработчика в этой ситуации является найти и применить наиболее подходящий набор плагинов.

В простейшем случае запустить плагин просто - для этого необходимо выполнить команду в определенном формате. Например, чтобы запустить плагин «maven-checkstyle-plugin» (artifactId) с groupId равным «org.apache.maven.plugins» необходимо выполнить следующую команду :

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

Целью (goal) выполнения данного плагина является проверка "check". Можно запустить в более краткой форме :

```
mvn maven-checkstyle-plugin:check
```

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

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>2.6</version>
  </plugin>
</plugins>
```

Объявление плагина в *pom.xml* позволяет зафиксировать версию плагина, задать ему необходимые параметры, определить конфигурационные

параметры, привязать к фазам.

Что касается списка конфигурационных переменных плагина, то его легко можно найти на сайте *maven*. К примеру, для *maven-compiler-plugin*, на странице Apache Maven Project можно увидеть перечень всех переменных, управляющих плагином.

Разные плагины вызываются *maven'ом* на разных стадиях жизненного цикла. Так проект, формирующий настольное java-приложение с использованием библиотек swing или swt, имеет стадии жизненного цикла отличные от тех, что характерны для разработке enterprise application (ear). Еак например, когда выполняется команда «mvn test», инициируется целый набор шагов в жизненном цикле проекта: «process-resources", «compile», «process-classes», «process-test-resources», «test-compile», «test». Упоминания этих фаз отражаются в выводимых maven-ом сообщениях :

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building carousel 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) \
        @ carousel ---
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) \
        @ carousel
[INFO] --- maven-resources-plugin:2.6:testResources \
        (default-testResources) @ carousel ---
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) \
        @ carousel ---
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) \
        @ carousel ---
[INFO] Surefire report directory: \
        E:\maven.projects\carousel\target\surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running ru.carousel.AppTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.04 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

```
[INFO] Total time: 5.042 s
[INFO] Finished at: 2016-09-24T12:33:45+04:00
[INFO] Final Memory: 7M/18M
[INFO] ------------------------------------------------------------------------
```

В каждой фазе жизненного цикла проекта вызывается определенный плагин (jar-библиотека), который включает некоторое количество целей (goal). Например, плагин «maven-compiler-plugin» содержит две цели: «compiler:compile» для компиляции основного исходного кода проекта и «compiler:testCompile» для компиляции тестов. Формально, список фаз можно изменять, хотя такая ситуация случается крайне редко.

В проектном файле *pom.xml* можно настроить для каждого из плагинов жизненного цикла набор конфигурационных переменных, например :

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <verbose>true</verbose>
        <executable>
              C:/Program_Files/Java/jdk1.7.0_67/bin/javac.exe
        </executable>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

В случае необходимости выполнения нестандартных действий в определенной фазе, например, на стадии генерации исходников «generate-sources», можно добавить вызов соответствующего плагина в файле *pom.xml* :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>имя-плагина</artifactId>
  <executions>
    <execution>
      <id>customTask</id>
      <phase>generate-sources</phase>
      <goals>
         <goal>pluginGoal</goal>
      </goals>
...
```

Самое важное в данном случае – это определить для плагина наименование фазы «execution/phase», в которую нужно встроить вызов цели плагина «goal». Фаза «generate-sources» располагается перед вызовом фазы compile и очень удобна для генерирования части исходных кодов проекта.

Описание различных плагинов представлено на странице Maven плагины для сборки проекта.

Основные исполняемые цели, goal

Использование **maven** часто сводится к выполнению одной из команды

следующего набора, которые можно назвать целями (по аналогии с другими системами сборки типа ant, make) :

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

В общем случае для выполнения команды *maven* необходимо выполнить следующий код : «mvn цель». В качестве параметров указываются не только имена фаз, но и имена и цели плагинов в формате «mvn плагин:цель». Например, вызов фазы цикла «mvn clean» эквивалентен вызову плагина «mvn clean:clean».

Секция свойств maven проекта, properties

Отдельные настройки проекта можно определить в переменных. Это может быть связанно, к примеру, с тем, что требуется использовать семейство библиотек определенной версии. Для этого в проектном файле используется секция <properties>, в которой объявляются переменные. Обращение к переменной выглядит следующим образом : ${имя переменной}. Пример описания свойств проекта и их использование :

```
<properties>
    <junit.version>4.11</junit.version>
    <maven.compiler.source>1.4</maven.compiler.source>
    <maven.compiler.target>1.6</maven.compiler.target>
</properties>

…
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
…
<build>
    <finalName>${project.artifactId}</finalName>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
```

```
            <version>2.3.2</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
            </configuration>
        </plugin>
    </build>
    …
```

Кодировка maven проекта

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

[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ ...
[WARNING] Using platform encoding (Cp1251 actually) to copy filtered resources, i.e. build is platform dependent!

[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ ...
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding Cp1251, i.e. build is platform dependent!

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

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

Для просмотра свойств проекта можно использовать плагин «maven-echo-plugin» :

```
<plugin>
    <groupId>org.codehaus.gmaven</groupId>
    <artifactId>groovy-maven-plugin</artifactId>
    <version>2.0</version>
    <executions>
        <execution>
            <phase>validate</phase>
            <goals>
                <goal>execute</goal>
            </goals>
            <configuration>
                <source>
                    log.info('JUnit версия : {0}', junit.version)
                </source>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Проектный файл, pom.xml

Структура проекта описывается в файле **pom.xml**, который должен находиться в корневой папке проекта. Содержимое проектного файла имеет следующий вид :

```
<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>

    <!-- GAV параметры описания проекта -->
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <packaging>...</packaging>
    <version>...</version>

    <!-- Секция свойств -->
    <properties>
        . . .
    </properties>

    <!-- Секция репозиториев -->
    <repositories>
        . . .
    </repositories>

    <!-- Секция зависимостей -->
    <dependencies>
        . . .
    </dependencies>

    <!-- Секция сборки -->
    <build>
        <finalName>projectName</finalName>
        <sourceDirectory>${basedir}/src/java</sourceDirectory>
        <outputDirectory>${basedir}/targetDir</outputDirectory>
        <resources>
            <resource>

<directory>${basedir}/src/java/resources</directory>
                <includes>
                    <include>**/*.properties</include>
                </includes>
            </resource>
        </resources>
        <plugins>
            . . .
        </plugins>
    </build>
</project>
```

Не все секции могут присутствовать в описании pom.xml. Так секции properties и repositories часто не используются. Параметры GAV проекта являются обязательными. Выше на странице было представлено описание использования различных секций. Здесь рассмотрим только секцию <build>.

Секция build

Секция <build> также не является обязательной, т. к. существует значение по умолчанию. Данная секция содержит информацию по самой сборке, т.е. где находятся исходные файлы, файлы ресурсов, какие плагины используются.

- <finalName> - наименование результирующего файла сборки (jar, war, ear..), который создаётся в фазе package. Значение по умолчанию — «artifactId-version»;
- <sourceDirectory> - определение месторасположения файлов с исходным кодом. По умолчанию файлы располагаются в директории «${basedir}/src/main/java», но можно определить и в другом месте;
- <outputDirectory> - определение месторасположения директории, куда компилятор будет сохранять результаты компиляции - *.class файлы. По умолчанию определено значение «target/classes»;
- <resources> и вложенные в неё тэги <resource> определяют местоположение файлов ресурсов. Ресурсы, в отличие от файлов исходного кода, при сборке просто копируются в директорию, значение по умолчанию которой равно «src/main/resources».

Тестирование проекта

Maven позволяет запускать JUnit case приложения на тестирование. Для этого следут выполнить команду "mvn test". Отдельные команды maven, например "mvn verify", автоматически запускают тесты приложения. Тестирование можно запретить на уровне выполнения команды или в секции "properties" файла pom.xml. Подробнее информация о тестировании с использованием maven представлена здесь.

Предопределёные переменные maven

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

- Встроенные свойства проекта :
  - ${basedir} - корневой каталог проекта, где располагается pom.xml;
  - ${version} - версия артефакта; можно использовать ${project.version} или ${pom.version};
- Свойства проекта. На свойства можно ссылаться с помощью префиксов «project» или «pom» :
  - ${project.build.directory} - «target» директория (можно ${pom.build.directory});
  - ${project.build.outputDirectory} - путь к директории, куда компилятор складывает файлы (по умолчанию «target/classes»);
  - ${project.name} - наименование проекта (можно ${pom.name});
  - ${project.version} - версия проекта (можно ${pom.version}).
- Настройки. Доступ к свойствам settings.xml можно получить с помощью префикса settings

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

Maven зависимости, dependency

Редко когда какой-либо проект обходится без дополнительных библиотек. Как правило, используемые в проекте библиотеки необходимо включить в сборку, если это не проект OSGi или WEB (хотя и для них зачастую приходится включать в проект отдельные библиотеки). Для решения данной задачи в maven-проекте необходимо использовать зависимость **dependency**, устанавливаемые в файле pom.xml, где для каждого используемого в проекте артефакта необходимо указать :

- параметры GAV (**g**roupId, **a**rtifactId, **v**ersion) и, в отдельных случаях, «необязательный» классификатор classifier;
- области действия зависимостей scope (compile, provided, runtime, test, system, import);
- месторасположение зависимости (для области действия зависимости system).

Параметры GAV

- **groupId** - идентификатор производителя объекта. Часто используется схема принятая в обозначении пакетов Java. Например, если производитель имеет домен domain.com, то в качестве значения *groupId* удобно использовать значение *com.domain*. То есть, *groupId* это по сути имя пакета.
- **artifactId** - идентификатор объекта. Обычно это имя создаваемого модуля или приложения.
- **version** - версия описываемого объекта. Для незавершенных проектов принято добавлять суффикс SNAPSHOT. Например 1.0.0-SNAPSHOT.

Значения идентификаторов *groupId* и *artifactId* подключаемых библиотек практически всегда можно найти на сайте www.mvnrepository.com. Если найти требуемую библиотеку в этом репозитории не удается, то можно использовать дополнительный репозиторий http://repo1.maven.org/maven2.

Структура файла pom.xml и описание секции подключения к проекту репозитория представлены на главной странице фреймворка maven.

Объявление зависимостей заключено в секции <dependencies>...</dependencies>. Количество зависимостей не ограничено. В следующем примере представлено объявление зависимости библиотеки JSON, в которой используется классификатор *classifier* (в противном случае библиотека не будет найдена в центральном репозитории) :

```
<dependencies>
  <dependency>
    <groupId>net.sf.json-lib</groupId>
    <artifactId>json-lib</artifactId>
    <version>2.4</version>
    <classifier>jdk15</classifier>
  </dependency>
<dependencies>
```

Классификатор classifier

Классификатор **classifier** используется в тех случаях, когда деление артефакта по версиям является недостаточным. К примеру, определенная библиотека (артефакт) может быть использована только с определенной JDK (VM), либо разработана под windows или linux. Определять этим библиотекам различные версии – идеологически не верно. Но вот использованием разных классификаторов можно решить данную проблему.

Значение *classifier* добавляется в конец наименования файла артефакта после его версии перед расширением. Для представленного выше примера полное наименование файла имеет следующий вид : json-lib-2.4-jdk15.jar.

Расположение артефакта в репозитории

В maven-мире «оперируют», как правило, артефактами. Это относится и к создаваемому разработчиком проекту. Когда выполняется сборка проекта, то формируется наименование файла, в котором присутствуют основные параметры GAV. После сборки этот артефакт готов к установке как в локальный репозиторий для использования в других проектах, так и для распространения в public-репозитории. Помните, что в начале файла pom.xml указываются параметры GAV артефакта :

```
<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>com.examples</groupId>
  <artifactId>example1</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  ...
</project>
```

Формально координата артефакта представляет четыре слова, разделенные знаком двоеточия в следующем порядке groupId:artifactId:packaging:version.

Полный путь, по которому находится файл артефакта в локальном репозитории, использует указанные выше четыре характеристики. В нашем примере для зависимости JSON это будет "HOME_PATH/.m2/repository/net/sf/json-lib/json-lib/2.4/json-lib-2.4-jdk15.jar". Параметру groupId соответствует директория (net/sf/json-lib) внутри репозитория (/.m2/repository). Затем идет поддиректория с artifactId (json-lib), внутри которой располагается поддиректория с версией (2.4). В последней располагается сам файл, в названии которого присутствуют все параметры GAV, а расширение файла согласуется с параметром *packaging*.

Здесь следует заметить, что правило, при котором «расширение файла с артефактом соответствует его packaging» не всегда верно. К примеру, те, кто знаком с разработкой enterprise приложений, включающих бизнес-логику в виде ejb-модулей и интерфейса в виде war-модулей, знают, что модули ejb- внешне представляют собой обычный архивный файл с расширением jar, хотя

в теге *packaging* определено значение ejb.

В каталоге артефакта, помимо самого файла, хранятся связанные с ним файлы с расширениями *.pom, *.sha1 и *.md5. Файл *.pom содержит полное описание сборки артефакта, а в файлах с расширениями sha1, md5 хранятся соответствующие значения MessageDidgest, полученные при загрузке артефакта в локальный репозиторий. Если исходный файл в ходе загрузки по открытым каналам Internet получил повреждения, то вычисленное значения sha1 и md5 будут отличаться от загруженного значения. А, следовательно, *maven* должен отвергнуть такой артефакт и попытаться загрузить его из другого репозитория.

Область действия зависимости, scope

Область действия scope определяет этап жизненного цикла проекта, в котором эта зависимость будет использоваться.

test

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

compile

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

provided

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

runtime

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

system

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

```
<dependencies>
  <dependency>
```

```
        <groupId>ru.carousel</groupId>
        <artifactId>carousel-lib</artifactId>
        <version>1.0</version>
        <scope>system</scope>
        <systemPath>
            /projects/libs/carousel-lib.jar
        </systemPath>
    </dependency>
</dependencies>
```

Версия SNAPSHOT

При определении версии релиза можно использовать **SNAPSHOT**, который будет свидетельствовать о том, что данный артефакт находится в процессе разработки и в него вносятся постоянные изменения, например делается bugfixing или дорабатывается функционал. В этом случае код и функциональность артефакта последней сборки в репозитории могут не соответствовать реальному положению дел. Таким образом нужно четко отделять стабильные версии артефактов от не стабильных. Связываясь с нестабильными артефактами нужно быть готовыми к тому, что их поведение может измениться и наш проект, использующий такой артефакт, может вызывать исключения. Следовательно, нужно определиться с вопросом: нужно ли обновлять из репозитория артефакт, ведь его номер формально остался неизменным.

Если версия модуля определяется как SNAPSHOT (версия 2.0.0-SNAPSHOT), то maven будет либо пересобирать его каждый раз заново вместо того, чтобы подгружать из локального репозитория, либо каждый раз загружать из public-репозитория. Указывать версию как SNAPSHOT нужно в том случае, если проект в работе и всегда нужна самая последняя версия.

Транзитивные зависимости

Начиная со второй версии фреймворка *maven* были введены транзитивные зависимости, которые позволяет избегать необходимости изучения и определения библиотек, которые требуются для самой зависимости. Maven включает их автоматически. В общем случае, все зависимости, используемые в проекте, наследуются от родителей. Ограничений по уровню наследований не существует, что, в свою очередь, может вызвать их сильный рост. В качестве примера можно рассмотреть создание проекта «A», который зависит от проекта «B». Но проект «B», в свою очередь, зависит от проекта «C». Подобная цепочка зависимостей может быть сколь угодно длинной. Как в этом случае поступает *maven* и как связан проект «A» и с проектом «C».

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

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

Плагин dependency

Имея приведенную выше таблицу правил переноса scope и набор соответствующих артефактам файлов pom можно построить дерево зависимостей для каждой из фаз жизненного цикла проекта. Строить вручную долго и сложно. Можно использовать maven-плагин *dependency* и выполнить команду «mvn dependency:list», в результате выполнения которой получим итоговый список артефактов и их scope :

```
F:\Projects\example>mvn dependency:list
Scanning for projects...

------------------------------------------------------------------
Building example 2.1
------------------------------------------------------------------

--- maven-dependency-plugin:2.8:list (default-cli) @ example ---

The following files have been resolved:
   net.sf.ezmorph:ezmorph:jar:1.0.6:compile
   ru.test:dao:jar:2.11:compile
   net.sf.json-lib:json-lib:jar:jdk15:2.4:compile
   ru.test:iplugin:jar:1.1:compile
   commons-collections:commons-collections:jar:3.2.1:compile
   commons-beanutils:commons-beanutils:jar:1.8.0:compile
   commons-lang:commons-lang:jar:2.5:compile
   org.eclipse.swt.win32.win32.x86:org.eclipse.swt.win32.win32.\
                              x86:jar:3.7.1.v3738:compile
   commons-logging:commons-logging:jar:1.1.1:compile
```

Однако к такому списку могут возникнуть вопросы : откуда взялся тот или иной артефакт? Т.е. желательно показать транзитные зависимости. И вот, команда «mvn dependency:tree» позволяет сформировать такое дерево зависимостей :

```
F:\Projects\example>mvn dependency:tree
Scanning for projects...
```

```
------------------------------------------------------------------------
Building example 2.1
------------------------------------------------------------------------

--- maven-dependency-plugin:2.8:tree (default-cli) @ example ---
 ru.test:example:jar:2.1
  +- net.sf.json-lib:json-lib:jar:jdk15:2.4:compile
  |  +- commons-beanutils:commons-beanutils:jar:1.8.0:compile
  |  +- commons-collections:commons-collections:jar:3.2.1:compile
  |  +- commons-lang:commons-lang:jar:2.5:compile
  |  +- commons-logging:commons-logging:jar:1.1.1:compile
  |  \- net.sf.ezmorph:ezmorph:jar:1.0.6:compile
  +- ru.test:iplugin:jar:1.1:compile
  |  \- ru.test:dao:jar:2.11:compile
  \- org.eclipse.swt.win32.win32.x86:org.eclipse.swt.win32.win32.\
                                    x86:jar:3.7.1.v3738:compile
```

Плагин *dependency* содержит большое количество целей *goal*, к наиболее полезным из которых относятся :

- **dependency:list** – выводит список зависимостей и области их действия scope;
- **dependency:tree** – выводит иерархический список зависимостей и области их действия scope;
- **dependency:purge-local-repository** – служит для удаления из локального репозитория всех артефактов, от которых прямо или косвенно зависит проект. После этого удаленные артефакты загружаются из Internet заново. Это может быть полезно в том случае, когда какой-либо артефакт был загружен со сбоями. В этом случае проще очистить локальный репозиторий и попробовать загрузить библиотеки заново;
- **dependency:sources** - служит для загрузки из центральных репозиториев исходников для всех артефактов, используемых в проекте. Порой отлаживая код, часто возникает необходимость подсмотреть исходный код какой-либо библиотеки;
- **dependency:copy-dependencies** - копирует зависимости/артефакты в поддиректорию target/dependency;
- **dependency:get** - копирует зависимость в локальный репозиторий.

Копирование зависимости в локальный репозиторий

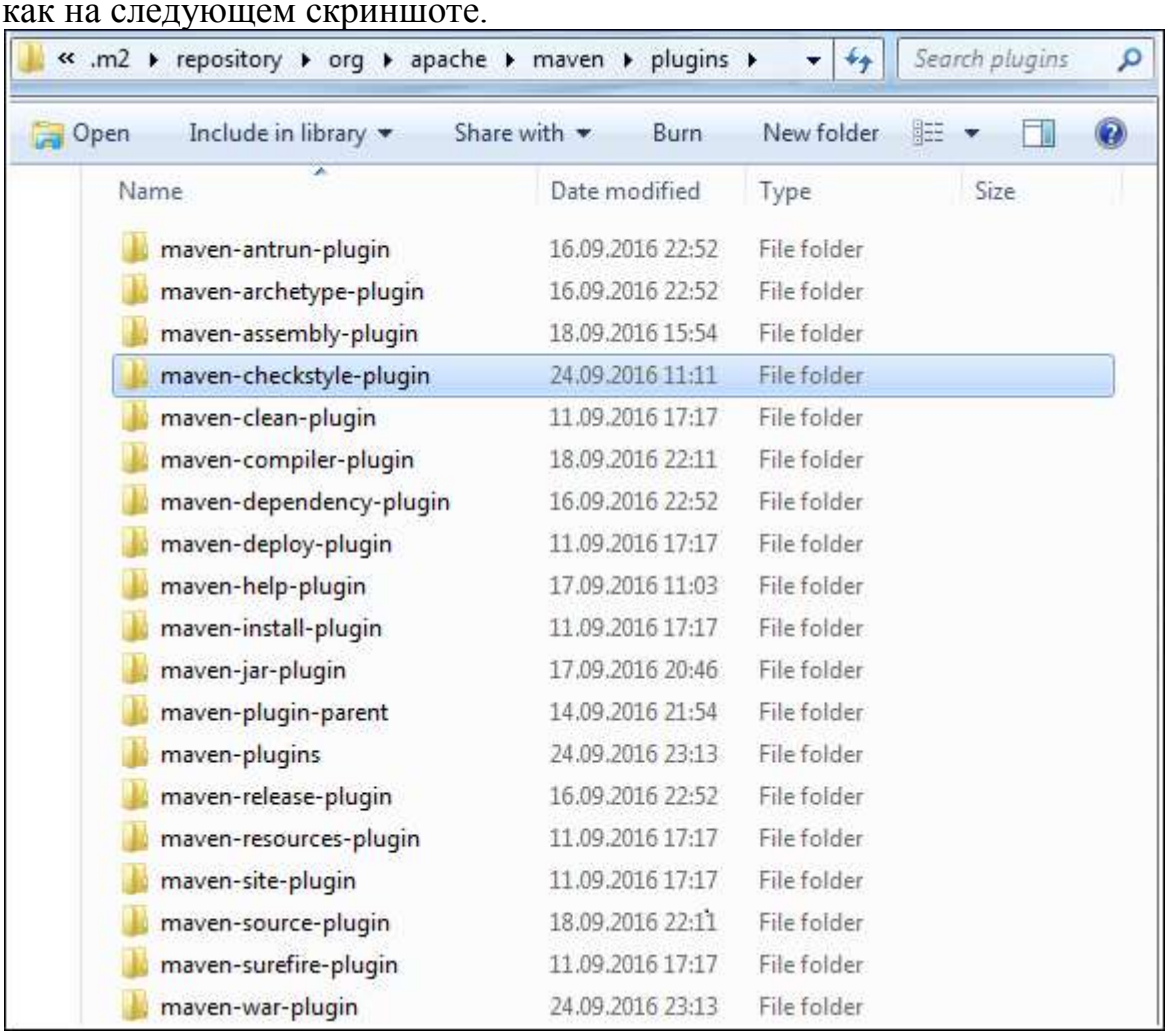
Следующий команда загрузит библиотеку JFreeChart (версия 1.0.19) в локальный репозиторий.

```
mvn dependency:get -Dartifact=org.jfree:jfreechart:1.0.19:jar
```

Maven плагины для сборки проекта

С описанием фреймворка **maven** можно познакомиться здесь. На этой

странице рассматриваются наиболее распространенные плагины, используемые при сборке проекта. Список установленных на компьютере плагинов *maven* можно увидеть в директории ${M2_HOME}/repository/org/apache/maven/plugins приблизительно в таком виде, как на следующем скриншоте.



На странице рассмотрены следующие плагины с примерами :

- maven-compiler-plugin - плагин компиляции;
- maven-resources-plugin - плагин включения ресурсов;
- maven-source-plugin - плагин включения исходных кодов;
- maven-dependency-plugin - плагин копирования зависимостей;
- maven-jar-plugin - плагин создания jar-файла;
- maven-surefire-plugin - плагин запуска тестов;
- buildnumber-maven-plugin - плагин генерации номера сборки;

Плагин создания проекта maven-archetype-plugin

Одним из самых первых плагинов, с которым приходится знакомиться или начинать новый проект, это *maven-archetype-plugin*. Данный плагин позволяет по определенному шаблону (archetype) сформировать структуру проекта. Примеры **maven** проектов для разнотипных приложений можно увидеть здесь.

Плагин компиляции maven-compiler-plugin

Самый популярный плагин, позволяющий управлять версией

компилятора и используемый практически во всех проектах, это компилятор *maven-compiler-plugin*. Он доступен по умолчанию, но практически в каждом проекте его приходится переобъявлять. В простейшем случае плагин позволяет определить версию java машины (JVM), для которой написан код приложения, и версию java для компиляции кода. Пример использования :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <source>1.7</source>
    <target>1.7</target>
    <encoding>UTF-8</encoding>
  </configuration>
</plugin>
```

В данном примере определена версия java-кода 1.7, на котором написана программа (source). Версия java машины, на которой будет работать программа, определена тегом <target>. В теге <encoding> указана кодировка исходного кода (UTF-8). По умолчанию используется версия java 1.3, а кодировка выбирается из операционной системы. Плагин позволяет указать путь к компилятору javac тегом <executable>.

Плагин *maven-compiler-plugin* имеет две цели :

- compiler:compile - компиляция исходников, по умолчанию связана с фазой compile;
- compiler:testCompile - компиляция тестов, по умолчанию связана с фазой test-compile.

Кроме приведёных настроек компилятор позволяет определить аргументы компиляции :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <compilerArgs>
      <arg>-verbose</arg>
      <arg>-Xlint:all,-options,-path<arg>
    </compilerArgs>
  </configuration>
</plugin>
```

Плагин позволяет даже выполнить компилирование не-java компилятором :

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
```

```
    <configuration>
      <compilerId>csharp</compilerId>
    </configuration>
    <dependencies>
      <dependency>
        <groupId>org.codehaus.plexus</groupId>
        <artifactId>plexus-compiler-csharp</artifactId>
        <version>1.6</version>
      </dependency>
    </dependencies>
  </plugin>
```

Плагин копирования ресурсов maven-resources-plugin

Перед сборкой проекта необходимо все ресурсы (файлы изображений, файлы .properties) скопировать в директорию *target*. Для этого используется плагин **maven-resources-plugin**. Пример использования плагина :

```
  <plugin>
    <artifactId>maven-resources-plugin</artifactId>
    <version>2.6</version>
    <executions>
      <execution>
        <id>copy-resources</id>
        <phase>validate</phase>
        <goals>
          <goal>copy-resources</goal>
        </goals>
        <configuration>
          <outputDirectory>
              ${basedir}/target/resources
          </outputDirectory>
          <resources>
            <resource>
              <directory>src/main/resources/props</directory>
              <filtering>true</filtering>
              <includes>
                <include>**/*.properties</include>
              </includes>
            </resource>
            <resource>
              <directory>src/main/resources/images</directory>
              <includes>
                <include>**/*.png</include>
              </includes>
            </resource>
          </resources>
        </configuration>
```

```
        </execution>
      </executions>
    </plugin>
```

Тег <outputDirectory> определяет целевую директорию, в которую будет происходить копирование.

В данном примере **maven** должен положить в директорию *target* всё точно также, как и было в проекте. При копировании ресурсов можно использовать дополнительные возможности *maven-resources-plugin*, позволяющие вносить изменения в файлы свойств. Для этого используется тег <filtering>, который при значении *true* предлагает плагину заглянуть во внутрь файла и при наличии определенных значений заменить их переменными *maven'a*. Файлы изображений не фильтруются. Поэтому ресурсы можно разнести по разным тэгам <resource />.

Дополнительно об использовании фильтрации для корректировки значений в файле свойств .properties можно почитать здесь.

Плагин включения исходных кодов maven-source-plugin

Плагин *maven-source-plugin* позволяет включать в сборку проекта исходный код. Данная возможность особенно полезна, если создается многомодульная архитектура проекта, включающая различные файлы .jar, и требуется отладка отдельных частей. Пример использования *maven-source-plugin* :

```
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-source-plugin</artifactId>
      <version>2.2.1</version>
      <executions>
        <execution>
          <id>attach-sources</id>
          <phase>verify</phase>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
```

В данном примере в релиз проекта будут включены исходные коды программы.

Плагин копирования зависимостей maven-dependency-plugin

Для копирования зависимостей в директорию сборки используют плагин *maven-dependency-plugin*. Пример копирования библиотек в директорию ${project.build.directory}/lib :

```
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.5.1</version>
```

```
<configuration>
  <outputDirectory>
    ${project.build.directory}/lib/
  </outputDirectory>
  <overWriteReleases>false</overWriteReleases>
  <overWriteSnapshots>false</overWriteSnapshots>
  <overWriteIfNewer>true</overWriteIfNewer>
</configuration>
<executions>
  <execution>
    <id>copy-dependencies</id>
    <phase>package</phase>
    <goals>
      <goal>copy-dependencies</goal>
    </goals>
  </execution>
</executions>
</plugin>
```

Назначение опций :

- outputDirectory - определение директории, в которую будут копироваться зависимости;
- overWriteReleases - флаг необходимости перезаписывания зависимостей при создании релиза;
- overWriteSnapshots - флаг необходимости перезаписывания неокончательных зависимостей, в которых присутствует SNAPSHOT;
- overWriteIfNewer - флаг необходимости перезаписывания библиотек с наличием более новых версий.

По умолчанию <overWriteReleases> и <overWriteSnapshots> - false, для <overWriteIfNewer> - true.

В примере определен раздел <execution> с идентификатором *copy-dependencies* - копирование зависимостей. Плагин используется в фазе сборки <package>, цель *copy-dependencies*. В разделе конфигурации *configuration* определен каталог, в который будут копироваться зависимости. Дополнительные параметры говорят о том, что перезаписываем библиотеки с наличием более новых версий, не перезаписываем текущие версии и не перезаписываем зависимости без окончательной версии (SNAPSHOT).

Плагин *maven-dependency-plugin* включает несколько целей, некоторые приведены ниже :

- mvn dependency:analyze - анализ зависимостей (используемые, неиспользуемые, указанные, неуказанные);
- mvn dependency:analyze-duplicate - определение дублирующиеся зависимостей;
- mvn dependency:resolve - разрешение (определение) всех зависимостей;
- mvn dependency:resolve-plugin - разрешение (определение) всех

плагинов;

- mvn dependency:tree - вывод на экран дерева зависимостей.

Плагин создания jar-файла maven-jar-plugin

Плагин *maven-jar-plugin* позволяет сформировать манифест, описать дополнительные ресурсы, необходимые для включения в jar-файл, и упаковать проект в jar-архив. Пример проектного файла pom.xml, описывающий настройку данного плагина :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.4</version>
  <configuration>
    <includes>
      <include>**/properties/*</include>
    </includes>
    <excludes>
      <exclude>**/*.png</exclude>
    </excludes>
    <archive>
      <manifestFile>src/main/resources/META-
INF/MANIFEST.MF</manifestFile>
    </archive>
  </configuration>
</plugin>
```

В примере определена директория и манифест, включаемые в сборку. Тегом <excludes> блокируется включение в сборку определенных файлов изображений.

Плагин *maven-jar-plugin* может создать и включить в сборку MANIFEST.MF самостоятельно. Для этого следует в секцию <archive> включить тег <manifest> с опциями :

```
<configuration>
  <archive>
    <manifest>
      <addClasspath>true</addClasspath>
      <classpathPrefix>lib/</classpathPrefix>
      <mainClass>ru.company.AppMain</mainClass>
    </manifest>
  </archive>
</configuration>
```

Опции тега <manifest> :

- <addClasspath> определяет необходимость добавления в манифест CLASSPATH;
- <classpathPrefix> позволяет дописывать префикс (в примере lib) перед каждым ресурсом;
- <mainClass> указывает на главный исполняемый класс.

Определение префикса в <classpathPrefix> позволяет размещать зависимости в отдельной папке.

Пример создания сборки (исполняемый jar-файл) с зависимостями библиотеки SWT можно посмотреть здесь.

Плагин тестирования maven-surefire-plugin

Плагин *maven-surefire-plugin* предназначен для запуска тестов и генерации отчетов по результатам их выполнения. По умолчанию на тестирование запускаются все java-файлы, наименование которых начинается с «Test» и заканчивается «Test» или «TestCase» :

- **/Test*.java
- **/*Test.java
- **/*TestCase.java

Если необходимо запустить java-файл с отличным от соглашения наименованием, например Sample.java, то необходимо в проектный файл pom.xml включить соответствующую секцию с плагином *maven-surefire-plugin*.

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.12.4</version>
    <configuration>
      <includes>
        <include>Sample.java</include>
      </includes>
    </configuration>
  </plugin>
</plugins>
```

Плагин *maven-surefire-plugin* содержит единственную цель *surefire:test*. Для разработки кодов тестирования можно использовать как JUnit, так и TestNG. Результаты тестирования в виде отчетов в форматах .txt и .xml сохраняются в директории ${basedir}/target/surefire-reports.

Иногда приходится отдельные тесты исключать. Это можно сделать включением в секцию <configuration> тега <excludes>.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.12.4</version>
  <configuration>
    <excludes>
      <exclude>**/TestCircle.java</exclude>
      <exclude>**/TestSquare.java</exclude>
    </excludes>
  </configuration>
</plugin>
```

Чтобы запустить проект на тестирование необходимо выполнить одну из следующих команд :

```
mvn test
```

```
mvn -Dmaven.surefire.debug test
```

Чтобы пропустить выполнение тестов на этапе сборки проекта, можно выполнить команду.

```
mvn clean package -Dmaven.test.skip=true
```

Также можно проигнорировать выполнение тестирования проекта включением в секцию <configuration> тега <skipTests>

```
<configuration>
   <skipTests>true</skipTests>
</configuration>
```

Плагин генерации номера сборки buildnumber-maven-plugin

Предположим, что нам нужно в манифест MANIFEST.MF нашего WEB-приложения и в файл свойств src/main/resources/app.properties положить номер сборки, который определяется переменной *${buildNumber}*. Файл манифеста будем генерить автоматически. А в файл свойств проекта app.properties включим параметры, значения которых будут определяться на этапе сборки проекта :

```
# application.properties
app.name=${pom.name}
app.version=${pom.version}
app.build=${buildNumber}
```

В режиме выполнения программы (runtime) можно обращаться к данному файлу свойств для получения наименования приложения и номера версии сборки.

Для генерирования уникального номера сборки проекта используется плагин *buildnumber-maven-plugin*. Найти плагин можно в репозитории в директории ${M2_HOME}/repository/org/codehaus/mojo. Пример настройки плагина в проектном файле pom.xml :

```
<modelVersion>4.0.0</modelVersion>
<groupId>ru.hellogwt.sample</groupId>
<artifactId>hellogwt</artifactId>
<packaging>war</packaging>
<version>1.0</version>
<name>GWT Maven Archetype</name>
. . . .
<plugins>
   <plugin>
```

```
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>buildnumber-maven-plugin</artifactId>
            <version>1.2</version>
            <executions>
              <execution>
                <phase>validate</phase>
                <goals>
                  <goal>create</goal>
                </goals>
              </execution>
            </executions>
            <configuration>
              <revisionOnScmFailure>true</revisionOnScmFailure>
              <format>{0}-{1,date,yyyyMMdd}</format>
                <items>
                 <item>${project.version}</item>
                 <item>timestamp</item>
                </items>
            </configuration>
          </plugin>
        </plugins>
```

В приведенном примере плагин запускается во время фазы жизненного цикла *validate* и генерирует номер версии ${buildNumber}, который собирается из нескольких частей и определен тэгом <format />. Каждая часть номера версии заключается в фигурные скобки и формируется согласно описанию *MessageFormat* языка Java. Каждой части соответствует тэг <item />, указывающий, какое значение должно быть подставлено.

Если в pom.xml не настроена работа с SCM (Source Code Management) типа Subversion, Git и т.п., то при попытке генерации номера сборки будет получено сообщение об ошибке "The scm url cannot be null". В этом случае можно указать в pom.xml заглушку SCM.

```
        <scm>
          <connection>scm:svn:http://127.0.0.1/dummy</connection>

<developerConnection>scm:svn:https://127.0.0.1/dummy</developerConnection>
          <tag>HEAD</tag>
          <url>http://127.0.0.1/dummy</url>
        </scm>
```

Чтобы номер сборки генерировался независимо от подключения к SCM в настройках конфигурации плагина следует указать свойство *revisionOnScmFailure* равным true.

Теперь настроим плагин *maven-war-plugin*, чтобы номер версии поместить в MANIFEST.MF, который будет создаваться автоматически :

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

```
<artifactId>maven-war-plugin</artifactId>
<version>2.6</version>
<configuration>
  <archive>
    <manifest>
      <addDefaultImplementationEntries>
        true
      </addDefaultImplementationEntries>
    </manifest>
    <manifestEntries>
      <Implementation-Build>
        ${buildNumber}
      </Implementation-Build>
    </manifestEntries>
  </archive>
</configuration>
</plugin>
```

Генерируемое значение, по-умолчанию, сохраняется в файле ${basedir}/buildNumber.properties и имеет имя buildNumber. При необходимости данные параметры могут быть переопределены через свойства buildNumberPropertiesFileLocation и buildNumberPropertyName соответственно.

Чтобы определить значения в файле свойств src/main/resources/app.properties включим фильтрацию ресурсов в разделе <build> :

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
   ....
</build>
```

Дополнительную информацию о фильтрации для корректировки значений в файле свойств *.properties можно почитать здесь.

После выполнения сборки проекта манифест MANIFEST.MF в файле .war будет иметь приблизительно следующий вид :

```
Manifest-Version: 1.0
Implementation-Title: GWT Maven Archetype
Implementation-Version: 1.0
Implementation-Vendor-Id: ru.hellogwt.sample
Built-By: Father
Build-Jdk: 1.7.0_67
Created-By: Apache Maven 3.3.9
```

Implementation-Build: 1.0-20161001
Archiver-Version: Plexus Archiver

Конечно же изменится и файл свойств app.properties в сборке :

```
# application.properties
app.name=GWT Maven Archetype
app.version=1.0
app.build=1.0-20161001
```

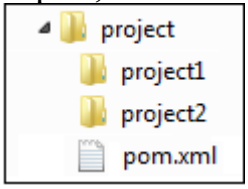
Чтобы в сервлете WEB-приложения прочитать версию сборки в файле MANIFEST.MF можно использовать следующий код :

```
import java.io.IOException;
import java.util.Properties;
...
String version = "UNDEFINED";
Properties prop = new Properties();
try {
   prop.load(getServletContext().getResourceAsStream("/META-INF/MANIFEST.MF"));
   version = prop.getProperty("Implementation-Build");
} catch (IOException e) {}
...
```

Наследование проектов в maven

Одним из важных аспектов многомодульного приложения является возможность независимой разработки отдельных модулей, обеспечивая, таким образом, расширение и изменение функциональности системы в целом. И здесь существенную помощь разработчикам оказывает фреймворк maven, который позволяет связать все проекты системы в единое целое. Чтобы объединить несколько maven-проектов в один связанный проект необходимо использовать **наследование**, которое определяет включение дополнительных секций в pom.xml (POM - Project Object Model).

Допустим необходимо разработать два взаимосвязанных проекта (project1, project2), которые должны быть объединены в едином родительском проекте *project*. Физически проекты необходимо расположить в одной родительской директории, в которой дочерние maven-проекты являются поддиректориями. Родительский файл pom.xml располагается в корневой директории, как это представлено на следующем скриншоте :



Настройка родительского pom.xml

В pom.xml родительского проекта необходимо определить параметры GAV (groupId, artifactId, version) и в теге <packaging> указать значение «pom». Дополнительно вводится секция <modules>, в которой перечисляются все дочерние проекты.

```
<groupId>com.example</groupId>
<artifactId>project</artifactId>
<version>0.0.1</version>
<packaging>pom</packaging>

. . .

<modules>
  <module>project1</module>
  <module>project2</module>
</modules>
```

Настройка дочерних pom.xml

В pom.xml дочерних проектов необходимо ввести секцию <parent> и определить GAV-параметры родительского проекта.

```
<parent>
  <groupId>com.example</groupId>
  <artifactId>project</artifactId>
  <version>0.0.1</version>
</parent>
```

На этом можно сказать, что все дочерние проекты привязаны к родительскому.

Применение наследования maven

Наследование в maven-проектах широко используется при разработке плагинов/бандлов для контейнеров OSGi (Open Services Gateway Initiative) и компонентов EJB (Enterprise JavaBeans). Также можно использовать «преимущества» наследования и в простых проектах.

Зачем объединять проекты?

В связанных многомодульных проектах можно исключить дублирование свойств и зависимостей, а также использовать централизованное управление и контролировать зависимости.

1. Общие свойства проектов

Связанные проекты позволяют определить общие свойства проектов, зависимости и разместить их в родительском pom.xml. Дочерние проекты будут автоматически наследовать свойства родителя. В следующем примере создаются общие секции <properties> и <dependencies>. В секцию зависимостей включены junit и log4j.

```
<properties>
  <junit.version>4.11</junit.version>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
  <dependency>
```

```
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
</dependencies>
```

Дочерние объекты наследуют значения следующих GAV-параметров родителя : <groupId>, <version>, но их можно при необходимости переопределить.

2. Централизованное управление проектами

Выполняя maven-команды в родительском проекте, они автоматически будут выполнены для каждого из подпроектов. В следующем примере выполняется команда install, согласно которой после сборки всех проектов они будут размещены в локальном репозитории.

```
$ cd project

$ mvn install
```

Таким образом, можно выполнять *maven* команды как для отдельного подпроекта, перемещаясь в его поддиректорию, так и для всех проектов вместе, располагаясь в родительской директории.

Чтобы посмотреть список зависимостей проекта/ов, необходимо выполнить команду «mvn dependency:tree» :

```
$ cd project

$ mvn dependency:tree

[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] Main project
[INFO] Chaild project1
[INFO] Chaild project2
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Main project 0.0.1
```

```
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ project ---
[INFO] com.example:project:pom:0.0.1
[INFO] +- junit:junit:jar:4.11:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] \- log4j:log4j:jar:1.2.17:compile
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Chaild project1 0.0.1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ project1 ---
[INFO] com.example:project1:jar:0.0.1
[INFO] +- junit:junit:jar:4.11:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] \- log4j:log4j:jar:1.2.17:compile
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Chaild project2 0.0.1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ project2 ---
[INFO] com.example:project2:jar:0.0.1
[INFO] +- junit:junit:jar:4.11:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] \- log4j:log4j:jar:1.2.17:compile
[INFO] ------------------------------------------------------------------------
```

Maven сначала выводит в консоль зависимости главного проекта, а потом зависимости для каждого из дочерних проектов. Как видно в примере значения groupId (com.example) и version (0.0.1) дочерних проектов совпадают с родительским. Они теперь необязательны и берутся по умолчанию у parent проекта, хотя можно определить собственные значения для каждого подпроекта.

Кроме свойств <properties> и зависимостей <dependencies> в родительском проекте часто объявляют необходимые для сборки плагины и репозитории.

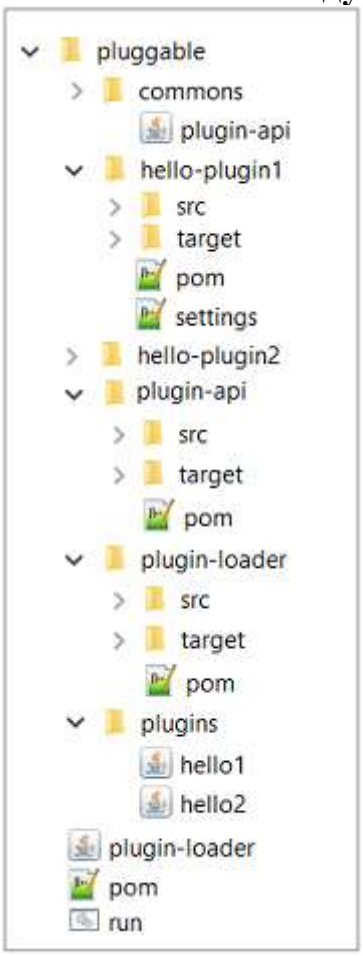
Многомодульный maven проект

Maven позволяет собирать проект из нескольких модулей. Каждый программный модуль включает свой проектный файл pom.xml. Один из проектных pom.xml файлов является корневым. Корневой pom.xml позволяет объединить все модули в единый проект. При этом в корневой проектный файл можно вынести общие для всех модулей свойства. А каждый модульный pom.xml должен включать параметры GAV (groupId, artifactId, version)

корневого pom.xml.

Общие положения разработки многомодульного maven-приложения рассмотрены на странице Наследование проектов в maven. В данной статье рассмотрим пример сборки многомодульного приложения. В качестве «подопытного» приложения используем пример, представленный на странице Pluggable решение. На выходе данного примера мы должны получить 3 архивных и один исполняемый jar-файлов. Главный исполняемый jar-модуль динамически «при необходимости» загружает остальные архивные jar'ники. Данный «подопытный» пример был использован для «оборачивания» jar'ника в exe-файл с использованием maven-плагина launch4j.

Описание многомодульного проекта



На скриншоте представлена структура проекта pluggable, включающая следующие проектные модули :

- hello-plugin1 – динамически загружаемый плагин №1 (hello1.jar);
- hello-plugin2 – динамически загружаемый плагин №2 (hello2.jar);
- plugin-api – интерфейсы описания плагинов (plugin-api.jar);
- plugin-loader – главный исполняемый jar модуль.

Дополнительные поддиректории проекта, используемые для размещения jar-модулей :

- commons – поддиректория размещения архивного jar-модуля описания интерфейса плагинов;
- plugins – поддиректория размещения jar-модулей (плагинов);

Главный исполняемый модуль plugin-loader.jar размещается в корневой директории проекта, где размещается и проектный/корневой pom.xml. Файл run.bat можно использовать для старта plugin-loader.jar из консоли в Windows.

Примечание : в исходные коды классов внесены изменения, связанные с из размещением в пакетах. В исходном примере все классы располагаются в «корне».

Начнем рассмотрение примера с корневого многомодульного pom.xml.

Листинг многомодульного корневого pom.xml

Корневой pom.xml включает параметры GAV (groupId, artifactId, Version), общую для всех модулей проекта секцию <properties> и секцию описания модулей <modules>. Обратите внимание на атрибут <**packaging**>, значение которого должно быть «**pom**».

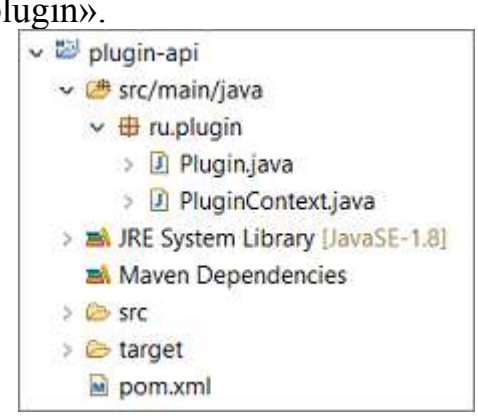
Следует отметить, что порядок включения программных модулей проекта составлен таким образом, что сначала представлены исполняемый модуль plugin-loader.jar и плагины (hello-plugin1.jar, hello-plugin2.jar), после чего следует интерфейсный модуль plugin-api.jar. Если собирать проект по-отдельности, то модуль plugin-api.jar должен быть собран в первую очередь и размещен в репозитории командой «mvn install». В этом случае зависимые модули plugin-loader.jar и плагины (hello-plugin1, hello-plugin2) собрались бы нормально. Ну, а мы в этом примере посмотрим, как поступит Maven в случае, если порядок описания модулей для сборки «нарушен».

```
<project
   xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>ru.pluggable.main</groupId>
   <artifactId>pluggable</artifactId>
   <packaging>pom</packaging>
   <version>1.0.0</version>
   <name>MultiModule application</name>

   <properties>
      <maven.test.skip>true</maven.test.skip>
      <java.version>1.8</java.version>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>
```

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

  <modules>
    <module>plugin-loader</module>
    <module>hello-plugin1</module>
    <module>hello-plugin2</module>
    <module>plugin</module>
  </modules>
</project>
```

Модуль описания интерфейсов плагинов plugin-api.jar

На следующем скриншоте представлена структура проекта plugin-api. Интерфейсные классы Plugin, PluginContext располагаются в пакете «ru.plugin».



Листинг pom.xml

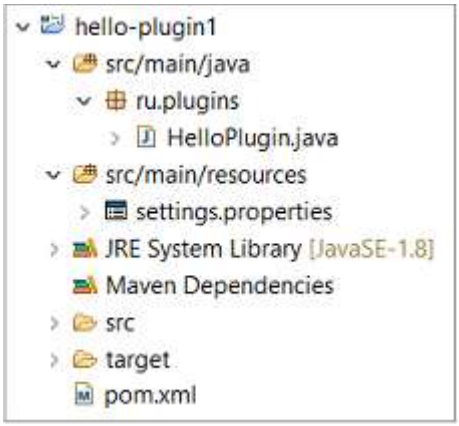
Проектный pom.xml модуля plugin-api.jar включает GAV-параметры, секцию описания родительского GAV (<parent>) и секцию сборки <build>, где в качестве выходной директории размещения (<outputDirectory>) указана поддиректория «${basedir}/../commons».

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ru.pluggable</groupId>
  <artifactId>plugin-api</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>Plugin API</name>

  <parent>
    <groupId>ru.pluggable.main</groupId>
    <artifactId>pluggable</artifactId>
```

```
    <version>1.0.0</version>
  </parent>

  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>
          org.apache.maven.plugins
        </groupId>
        <artifactId>
          maven-jar-plugin
        </artifactId>
        <version>2.3.1</version>
        <configuration>
          <outputDirectory>
            ${basedir}/../commons
          </outputDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Модуль описания плагина hello-plugin1.jar

Структура проекта hello-plugin1 представлена на следующем скриншоте.



Класс HelloPlugin, расположенный в пакете «ru.plugins», реализует свойства интерфейса Plugin. При инициализации класса в методе init определяется значение контекста PluginContext родительского/вызвавшего объекта. Метод invoke выводит в консоль сообщение и изменяет надпись на кнопке родительского объекта.

```
package ru.plugins;

import ru.plugin.Plugin;
import ru.plugin.PluginContext;
```

```
public class HelloPlugin implements Plugin
{
   private PluginContext pc;

   @Override
   public void invoke() {
      System.out.println("Hello world. I am a plugin 1");
      pc.getButton().setText("Other text 1");
   }
   @Override
   public void init(PluginContext context) {
      this.pc = context;
   }
}
```

Листинг pom.xml

Проектный pom.xml модуля hello-plugin1.jar включает GAV-параметры, секцию описания родительского GAV (<parent>), секцию зависимостей (<dependencies>) и секцию сборки <build>. В секции зависимостей указываются параметры модуля plugin-api. В описании секции сборки используется 2 плагина (maven-resources-plugin, maven-jar-plugin). Первый плагин включает в сборку ресурсы (resources/settings.properties), второй плагин создает jar и размещяет его в выходной директории (<outputDirectory>) «${basedir}/../plugins».

```
<project
   xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>ru.plugins</groupId>
   <artifactId>hello1</artifactId>
   <packaging>jar</packaging>
   <version>1.0.0</version>
   <name>Plugin Hello1</name>

   <parent>
      <groupId>ru.pluggable.main</groupId>
      <artifactId>pluggable</artifactId>
      <version>1.0.0</version>
   </parent>

   <dependencies>
     <dependency>
        <groupId>ru.pluggable</groupId>
```

```
      <artifactId>plugin-api</artifactId>
      <version>1.0.0</version>
    </dependency>
</dependencies>

<build>
<finalName>${project.artifactId}</finalName>
<plugins>
  <plugin>
    <artifactId>
      maven-resources-plugin
    </artifactId>
    <version>2.6</version>
    <executions>
      <execution>
        <id>copy-resources</id>
        <configuration>
          <outputDirectory>
            ${basedir}/target/resources
          </outputDirectory>
          <resources>
            <directory>
              src/main/resources
            </directory>
            <resource>
              <includes>
                <include>
                  **/*.properties
                </include>
              </includes>
            </resource>
          </resources>
        </configuration>
      </execution>
    </executions>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.3.1</version>
    <configuration>
      <outputDirectory>
        ${basedir}/../plugins
      </outputDirectory>
    </configuration>
```

```
        </plugin>
      </plugins>
    </build>
  </project>
```

Примечание : второй плагин hello-plugin2 структурно ничем не отличается от hello-plugin1. Отличия касаются текста сообщения в консоли, надписи на кнопке и параметров GAV в pom.xml.

Проектный pom.xml модуля plugin-loader

Проектный pom.xml включает GAV-параметры jar-модуля, секцию описания родительского GAV (<parent>), секцию зависимостей (<dependencies>) и секцию сборки <build>. В секции зависимостей указываются параметры модуля plugin-api. В описании секции сборки используется плагин maven-jar-plugin, который создает jar и размещяет его в корневой директории проекта (<outputDirectory>).

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ru.pluggable.loader</groupId>
  <artifactId>plugin-loader</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>Plugin Loader</name>

  <parent>
    <groupId>ru.pluggable.main</groupId>
    <artifactId>pluggable</artifactId>
    <version>1.0.0</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>ru.pluggable</groupId>
      <artifactId>plugin-api</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>

  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

```
                <artifactId>maven-jar-plugin</artifactId>
                <version>2.3.1</version>
                <configuration>
                  <outputDirectory>
                    ${basedir}/..
                  </outputDirectory>
                  <archive>
                    <manifest>
                      <mainClass>
                        ru.pluggable.loader.Boostrap
                      </mainClass>
                    </manifest>
                  </archive>
                 </configuration>
              </plugin>
            </plugins>
          </build>
        </project>
```

Сборка проекта

Сборка всех проектов выполняется одной командой «mvn package» для корневого pom.xml. Maven сначала просматривает проектные файлы pom.xml всех модулей, определенных в корневом pom.xml, и после этого определяет порядок сборки модулей. Как следует из представленных ниже сообщений, выводимых Maven в консоль, порядок сборки был изменен и первым собирается модуль Plugin API, после чего формируются зависимые от него Plugin Loader, Plugin Hello1, Plugin Hello2.

```
D:\pluggable>mvn package
[INFO] Scanning for projects...
[INFO] ------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] MultiModule application
[INFO] Plugin API
[INFO] Plugin Loader
[INFO] Plugin Hello1
[INFO] Plugin Hello2
[INFO]
[INFO] ------------------------------------------------
[INFO] Building MultiModule application 1.0.0
[INFO] ------------------------------------------------
[INFO]
[INFO] ------------------------------------------------
[INFO] Building Plugin API 1.0.0
[INFO]
```

```
[INFO] --- maven-resources-plugin:2.6:resources \
          (default-resources) @ plugin-api ---
[INFO] Using 'UTF-8' encoding to copy filtered \
          resources.
[INFO] skip non existing resourceDirectory \
          D:\pluggable\plugin\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile \
         (default-compile) @ plugin-api ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to \
          D:\pluggable\plugin\target\classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources \
          (default-testResources) @ plugin-api ---
[INFO] Not copying test resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile \
          (default-testCompile) @ plugin-api ---
[INFO] Not compiling test sources
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test \
          (default-test) @ plugin-api ---
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) \
          @ plugin-api ---
[INFO] Building jar: \
          D:\pluggable\plugin\..\commons\plugin-api.jar
[INFO]
[INFO] ------------------------------------------------
[INFO] Building Plugin Loader 1.0.0
[INFO] ------------------------------------------------
[INFO]
. . .
[INFO]
[INFO] ------------------------------------------------
[INFO] Building Plugin Hello1 1.0.0
[INFO] ------------------------------------------------
[INFO]
. . .
[INFO]
[INFO] ------------------------------------------------
[INFO] Building Plugin Hello2 1.0.0
[INFO] ------------------------------------------------
```

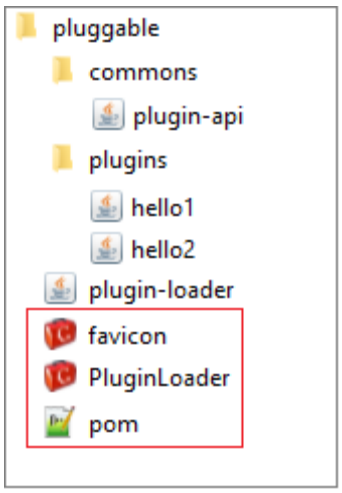
```
. . .
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) \
        @ hello2 ---
[INFO] Building jar: \
        D:\pluggable\hello-plugin2\..\plugins\hello2.jar
[INFO] ------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] MultiModule application ..... SUCCESS [  0.006 s]
[INFO] Plugin API ..................... SUCCESS [  1.957 s]
[INFO] Plugin Loader ................. SUCCESS [  0.391 s]
[INFO] Plugin Hello1 ................. SUCCESS [  0.183 s]
[INFO] Plugin Hello2 ................. SUCCESS [  0.115 s]
[INFO] ------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------
[INFO] Total time: 2.802 s
[INFO] Finished at: 2019-06-05T11:07:45+03:00
[INFO] Final Memory: 18M/199M
[INFO] ------------------------------------------------
```

Создание exe-файла из jar

Пользователям Windows привычнее использовать исполняемое приложение в виде exe-файла, нежели архивного jar-файла. Разработчики настольных java-приложений могут плагином **launch4j** не только обернуть исполняемый архивный jar-файл в оболочку exe-файла, но и включить в него иконку, автора, версию. Также данный плагин позволяет определить минимальную версию используемой JRE. В данной статье рассмотрим использование maven-плагина **launch4j** для получения exe-файла.

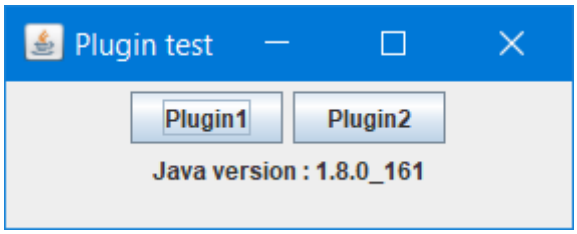
Описание java-примера

В качестве java-примера используем pluggable решение, включающее несколько jar-файлов. На следующем скриншоте представлена структура нашего экспериментального примера. Три файла, выделенные красным прямоугольником и относящиеся к задаче создания исполняемого exe-файла, рассматриваются ниже.



Несколько слов о структуре примера. Описание с исходными кодами данного java-примера представлено на странице Pluggable решение. Желающие могут поближе познакомиться с технологией динамической загрузки jar-файлов (классов), открыв страницу с подробным описанием исходников. На «выходе» данного примера получаем главный исполняемый модуль *plugin-loader.jar*, который использует *common/plugin-api.jar* для загрузки при необходимости (вызове) плагинов *plugins/hello1.jar* и *plugins/hello2.jar*.

Графический интерфейс примера, представленный на следующем скриншоте, включает 2 кнопки с надписями 'Plugin1' и 'Plugin2'. При нажатии на одну из кнопок приложение подгружает необходимый плагин, который меняет надпись на кнопке.



Сообщения в консоли

Динамически загружаемые плагины выводят в консоль дополнительно сообщения. Ниже представлены сообщения от двух плагинов.

```
Hello world. I am a plugin 1
I am a plugin 2
```

Изменения в исходных кодах

Необходимо отметить, что в модули PluginLoader.java и Boostrap.java были внесены изменения. Так в PluginLoader.java добавлена метка JLabel с отображением в интерфейсе версии Java :

```
    . . .
    JLabel label = new JLabel("Java version : " +
System.getProperty("java.version"));
    label.setSize(200, 24);
    frame.getContentPane().add(label);
    . . .
```

В класс Boostrap.java внесены изменения, связанные с чтением классов (*.class) из jar'ника, а не из директории bin, как это представлено в исходных кодах. Если этого не сделать, то придётся с собой ещё «таскать» и директорию bin с class'ами.

Листинг класса Boostrap.java

В главный класс Boostrap внесены изменения определения url : ниже исходной закомментированной строки размещается код определения url в jar-файле.

```
import java.io.File;
import java.lang.reflect.Method;

import java.net.URL;
import java.net.URLClassLoader;

public class Boostrap {

    public static void main(String[] args) throws Exception
    {
        File commonsDir = new File("commons");

        File[] entries = commonsDir.listFiles();
        URL[] urls = new URL[entries.length];

        for (int i = 0; i < entries.length; i++)
            urls[i] = entries[i].toURI().toURL();

        URLClassLoader loader;
        loader = new URLClassLoader(urls, null);

//      URL url = new File("bin").toURI().toURL();

        File file = new File(".");
        String path = "jar:file:/" + file.getCanonicalPath();
        URL url = new URL(path+"/plugin-loader.jar!/");

        URLClassLoader appLoader;
        appLoader = new URLClassLoader(new URL[]{url},loader);

        Class<?> appClass = loader.loadClass("PluginLoader");
        Object appInstance = appClass.newInstance();
        Method m = appClass.getMethod("start");
        m.invoke(appInstance);
    }
}
```

Оборачивание исполняемого jar в exe-файл

Обычно плагин **maven.plugins.launch4j** включают в проектный pom.xml файл, в котором формируется и исполняемый jar-файл. Поскольку основная цель данной статьи наглядно продемонстрировать возможность оборачивания jar в exe, то уберем из проектного pom.xml все лишнее, что связано с формированием jar-файла. Правильнее сказать создадим такой pom.xml, который и будет решать основную задачу оборачивания jar в exe.

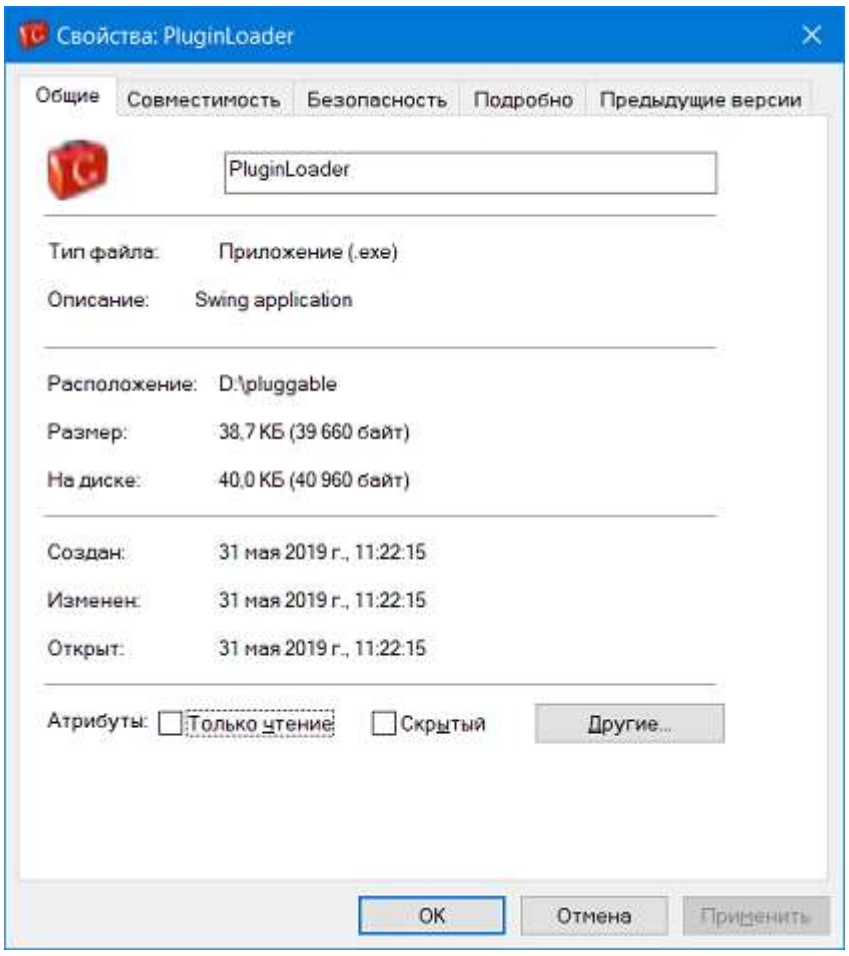
Следующий листинг проектного файла pom.xml решает данную задачу. Сам pom.xml существенно упростился и стал более наглядным. В разделе <properties> определяются наименование компании (product.company) и наименование исполняемого файла (exeFileName), а также минимальная версия jdkVersion. Основные настройки плагина

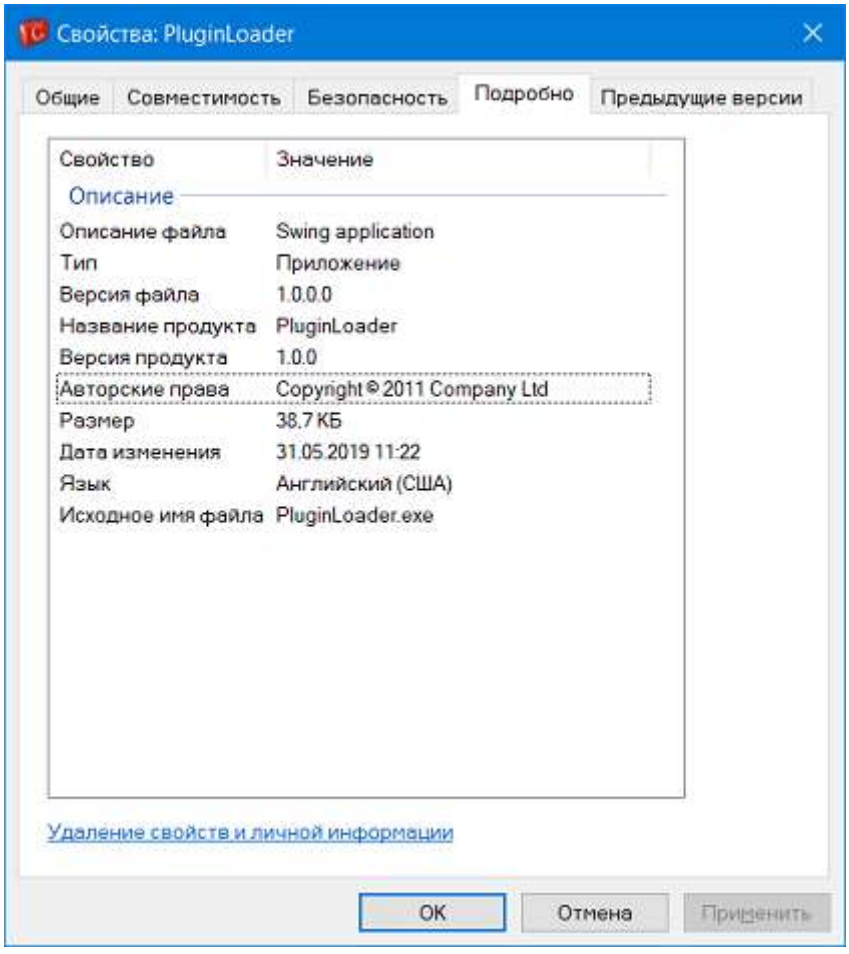
определяются в разделе <executions>. В секции <configuration> указываются jar-файл, exe-файл (outfile) и иконка испольняемого файла (icon). Плагин будет ругаться, если не укажете наименование иконки. Следует отметить, что в секции <classPath> необходимо указать главный стартуемый java-класс (mainClas).

```
<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>com.demo</groupId>
  <artifactId>plugin-loader</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>plugin-loader</name>

  <properties>
      <jdkVersion>1.8</jdkVersion>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>

      <project.build.sourceEncoding>
          UTF-8
      </project.build.sourceEncoding>
      <product.company>MultiModule</product.company>
      <product.title>PluginLoader</product.title>
      <exeFileName>PluginLoader</exeFileName>
  </properties>

  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
    <plugin>
        <groupId>
            com.akathist.maven.plugins.launch4j
        </groupId>
        <artifactId>launch4j-maven-plugin</artifactId>
        <executions>
            <execution>
                <id>plugin-loader</id>
                <phase>package</phase>
                <goals>
                    <goal>launch4j</goal>
                </goals>
                <configuration>
                    <headerType>gui</headerType>
                    <outfile>${exeFileName}.exe</outfile>
                    <jar>${project.artifactId}.jar</jar>
                    <errTitle>${product.title}</errTitle>
                    <icon>favicon.ico</icon>
                    <classPath>
                        <mainClass>Boostrap</mainClass>
                        <addDependencies>
                            true
                        </addDependencies>
                        <preCp>anything</preCp>
                    </classPath>
                    <jre>
                        <minVersion>
                           ${jdkVersion}
                        </minVersion>
                    </jre>
                    <versionInfo>
```

```
                        <fileVersion>
                            ${project.version}
                        </fileVersion>
                        <txtFileVersion>
                            ${project.version}
                        </txtFileVersion>
                        <fileDescription>
                            Swing application
                        </fileDescription>
                        <copyright>
                          Copyright © 2011 ${product.company}
                        </copyright>
                        <productVersion>
                            ${project.version}
                        </productVersion>
                        <txtProductVersion>
                            ${project.version}
                        </txtProductVersion>
                        <companyName>
                            ${product.company}
                        </companyName>
                        <productName>
                            ${product.title}
                        </productName>
                        <internalName>
                            ${exeFileName}
                        </internalName>
                        <originalFilename>
                            ${exeFileName}.exe
                        </originalFilename>
                    </versionInfo>
                </configuration>
            </execution>
        </executions>
    </plugin>
    </plugins>
  </build>
</project>
```

На следующих скриншотах представлены вкладки свойств созданного PluginLoader.exe.


Свойства: PluginLoader
Общие
Совместимость
Безопасность
Подробно
Предыдущие версии
PluginLoader
Тип файла: Приложение (.exe)
Описание: Swing application
Расположение: D:\pluggable
Размер: 38,7 КБ (39 660 байт)
На диске: 40,0 КБ (40 960 байт)
Создан: 31 мая 2019 г., 11:22:15
Изменен: 31 мая 2019 г., 11:22:15
Открыт: 31 мая 2019 г., 11:22:15
Атрибуты: Только чтение Скрытый Другие...
OK
Отмена
Применить


Свойства: PluginLoader
Общие
Совместимость
Безопасность
Подробно
Предыдущие версии
Свойство
Значение
Описание
Описание файла Swing application
Тип Приложение
Версия файла 1.0.0.0
Название продукта PluginLoader
Версия продукта 1.0.0
Авторские права Copyright © 2011 Company Ltd
Размер 38,7 КБ
Дата изменения 31.05.2019 11:22
Язык Английский (США)
Исходное имя файла PluginLoader.exe
Удаление свойств и личной информации
OK
Отмена
Применить