

Инструмент сборки программных проектов Apache Maven

Сидоров Кирилл

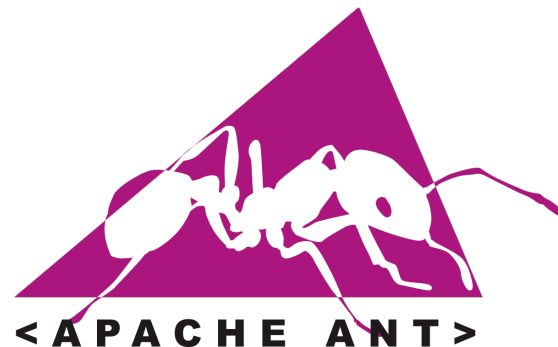
Что такое инструмент сборки программного проекта?

Инструмент сборки — это инструмент, который автоматизирует все, что связано с созданием итогового программного продукта.

Инструмент сборки позволяет автоматизировать следующие основные действия.

1. Загрузка из сетевого хранилища сторонних зависимостей (для java архивы .jar (библиотеки java)).
2. Компиляция исходного кода.
3. Упаковка скомпилированного кода в архив (для java архивы .jar, .war или .ear).
4. Генерация документации по коду проекта.
5. Установка архива с программой на сервер или сохранение в хранилище зависимостей.

Существующие инструменты для сборки программного проекта для Java



Описание вашего java
проекта в файле **pom.xml**

Консольная программа **mvn**
на вашем компьютере

Maven

- Читает pom.xml
- **Загружает** зависимости в локальный репозиторий
- Выполняет **Жизненный цикл, Фазы сборки и Цели**
- Выполняет **Плагины**

Все действия выполняются в соответствии с заданным **Профилем**

POM (Project Object Model)

Project coordinates
(координаты проекта)

Properties
(свойства)

Dependencies
(зависимости)

Plugins (плагины)

Profiles (профили)

Удаленное хранилище
зависимостей
(в интернете)

Например, Maven Central.
Веб интерфейс:
<https://central.sonatype.com/>
Само хранилище файлов:
<https://repo.maven.apache.org/maven2/>

Загрузка

Сохранение

Локальное хранилище зависимостей



Папка **.m2** на вашем компьютере

Жизненный цикл сборки, фазы сборки и цели сборки

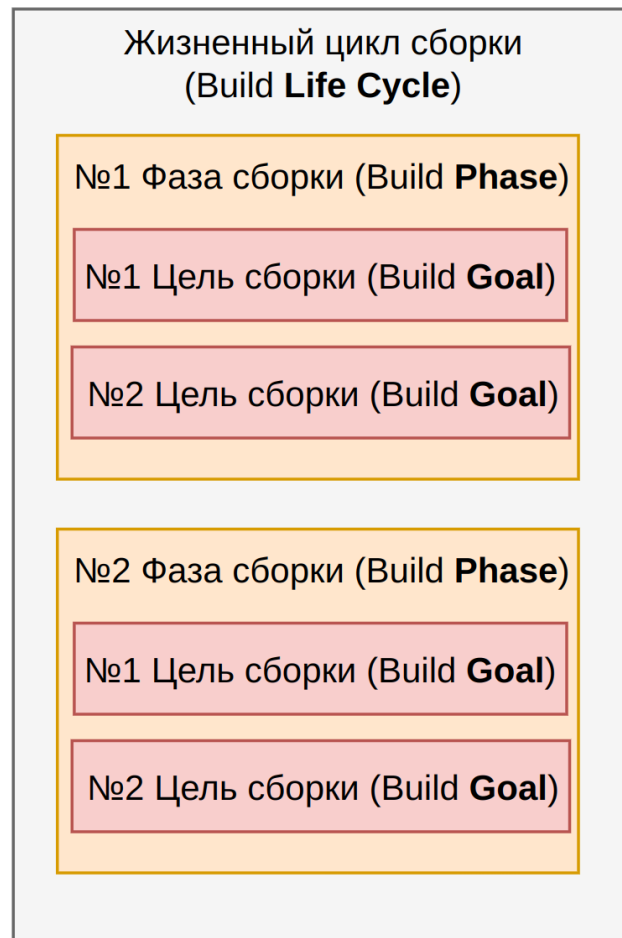
Когда Maven создает программный проект, он следует определенному **Жизненному циклу сборки**.

Каждый **Жизненный цикл** сборки разделен на **Фазы сборки**, а фазы сборки — на **Цели сборки**.

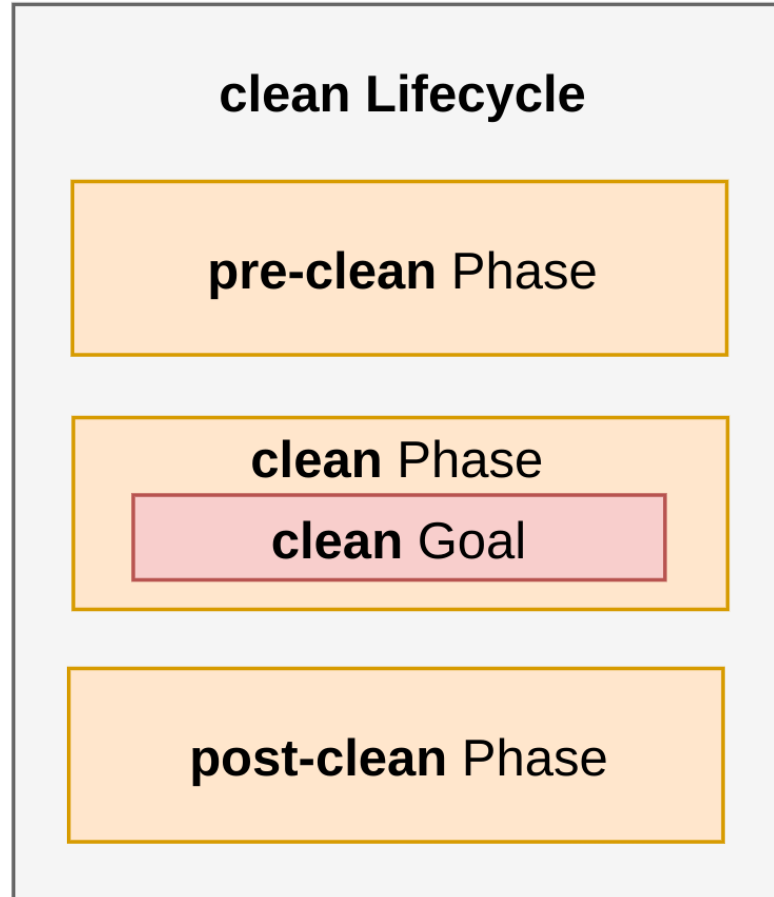
Maven имеет **три** встроенных **Жизненных цикла** сборки.

1. **default** (основной)
2. **clean** (очистка ресурсов)
3. **site** (создание документации)

Структура жизненных циклов в Maven



Пример устройства Жизненного цикла **clean**



Выполнение Жизненных циклов, Фаз и Целей

Maven может выполнить либо весь жизненный цикл целиком (**кроме жизненного цикла default!**), либо определенную **фазу** жизненного цикла, либо определенную **цель** фазы.

Пример запуска выполнения **Жизненного цикла clean**:

```
mvn clean
```

Пример запуска выполнения **Фазы install** Жизненного цикла default:

```
mvn install
```

Пример запуска выполнения **Цели compile** Фазы compile Жизненного цикла default:

```
mvn compile:compile
```

Выполнение Жизненных циклов, Фаз и Целей

- При запуске **жизненного цикла** будут выполнены все фазы и цели этого жизненного цикла.
- При запуске **фазы** будут выполнены все цели этой фазы (и предыдущие фазы).
- При запуске **цели** будет выполнена эта цель (и предыдущие фазы).

Основные фазы Жизненного цикла **default**

№	Фаза	Описание
1	validate	Проверяет проект на наличие всей необходимой информации, зависимостей и т.д.
2	compile	Компилирует исходного кода проекта.
3	test	Запускает тесты на основе скомпилированного кода, используя среду модульного тестирования.
4	package	Упаковывает скомпилированный код в архив .jar, .war или .ear.
5	install	Сохраняет проект в локальный репозиторий (папка .m2) для использования в качестве зависимостей в других локальных проектах.
6	deploy	Сохраняет проект в удаленный репозиторий.

При запуске определенной фазы выполняются все предшествующие этой фазе фазы. Например, при запуске фазы `test (mvn test)` будут также выполнены фазы: `validate` и `compile`.

Выполнение Жизненных циклов, Фаз и Целей




Полный список **Фаз** и **Целей** трех **Жизненных циклов** представлен в документации Maven:

<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

Plugins (Плагины)

1. **Плагин** — это программа на языке Java (**зависимость .jar**), которая может выполняться Maven-ом.
2. Можно сказать, что Maven — это только фреймворк (скелет или каркас), а вся полезная **работа выполняется конкретными Плагинами**.
3. Плагины подключаются в pom.xml в блоке
`<build> <plugins> </plugins> </build>`
и **привязываются к определенной Фазе** жизненного цикла.
4. У Maven-на есть **встроенные плагины** для выполнения основных задач.

Встроенные плагины в Maven

- >  **clean** (org.apache.maven.plugins:maven-clean-plugin:3.2.0)
- >  **compiler** (org.apache.maven.plugins:maven-compiler-plugin:3.11.0)
- >  **deploy** (org.apache.maven.plugins:maven-deploy-plugin:3.1.1)
- >  **install** (org.apache.maven.plugins:maven-install-plugin:3.1.1)
- >  **jar** (org.apache.maven.plugins:maven-jar-plugin:3.3.0)
- >  **resources** (org.apache.maven.plugins:maven-resources-plugin:3.3.1)
- >  **site** (org.apache.maven.plugins:maven-site-plugin:3.12.1)
- >  **surefire** (org.apache.maven.plugins:maven-surefire-plugin:3.1.2)

Настройка плагинов в Maven

1. **Плагин** может быть (или должен быть) привязан к определенной **Фазе** (или нескольким фазам) жизненного цикла.
2. Привязка плагина к определенной фазе описывается в **блоке execution**.
3. У **execution**-а плагина нужно **ВСЕГДА ОБЯЗАТЕЛЬНО** указывать его **id**!
4. Более тонкие настройки (варианты конфигурации) плагина нужно смотреть в документации к этому плагину.

Настройка плагинов в Maven

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>3.6.1</version>
  <executions>
    <execution>
      <id>analyze-when-verify</id>
      <phase>verify</phase>
      <goals>
        <goal>analyze</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Нужно обязательно указывать id у execution плагина!

Плагин для сборки исполняемого приложения

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>3.1.1</version>
  <configuration>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
    <archive>
      <manifest>
        <mainClass>org.example.Main</mainClass> <!-- Класс с методом main(String[] args)-->
      </manifest>
    </archive>
  </configuration>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase> <!-- Плагин должен быть прикреплен к Фазе package -->
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Предустановленный плагин компиляции maven-compiler-plugin

Предустановленный плагина компиляции maven-compiler-plugin (который выполняется в Фазе compile) по умолчанию использует компилятор версии Java 5.

Установить нужную версию компилятора можно 2 способами:

1. Установить в properties pom.xml нужную версию Java для компиляции, пример для Java 11:

```
<maven.compiler.target>11</maven.compiler.target>
```

```
<maven.compiler.source>11</maven.compiler.source>
```

Эти значения будут использованы плагином компиляции при запуске Фазы compile.

2. Установить значения версии java непосредственно в конфигурации плагина maven-compiler-plugin.

Предустановленный плагин компиляции maven-compiler-plugin

Первый способ установки версии для плагина компиляции

```
<properties>  
  <maven.compiler.source>11</maven.compiler.source>  
  <maven.compiler.target>11</maven.compiler.target>  
</properties>
```

Либо

Второй способ установки версии для плагина компиляции

```
<plugin>  
  <groupId>org.apache.maven.plugins</groupId>  
  <artifactId>maven-compiler-plugin</artifactId>  
  <version>3.11.0</version>  
  <configuration>  
    <source>11</source>  
    <target>11</target>  
  </configuration>  
</plugin>
```

Версионирование зависимостей

Стандартным способом является «Семантическое Версионирование 2.0.0», которое заключается в следующем (<https://semver.org/lang/ru/>).

Формат версии зависимости:

МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ.

Значение следует увеличивать, если:

1. МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения API.
2. МИНОРНУЮ версию, когда вы добавляете новую функциональность, не нарушая обратной совместимости.
3. ПАТЧ-версию (или Баг-Фикс), когда вы делаете обратно совместимые исправления.

```
<dependency>
```

```
  <groupId>junit</groupId>
```

```
  <artifactId>junit</artifactId>
```

```
  <version>4.13.1</version>
```

```
</dependency>
```



Версионирование зависимостей

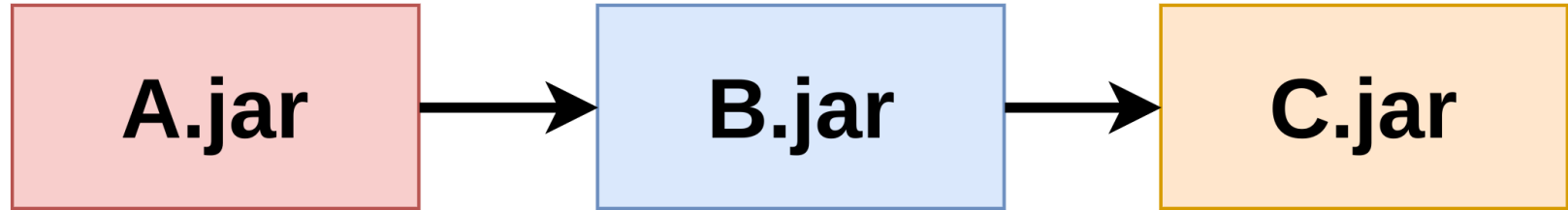
SNAPSHOT зависимость (например, 1.0.0-SNAPSHOT) — это зависимость, которая находится в процессе разработки и постоянно обновляется в удаленном репозитории.

Релизная версия зависимости (например, 1.0.0) может быть сохранена в удаленный репозиторий только один раз, после этого она уже не может изменяться. Повторное сохранение **релизной зависимости** в удаленный репозиторий невозможно.

Транзитивные зависимости

Общий пример **транзитивной зависимости**.

На рисунке зависимость A транзитивно зависит от C.



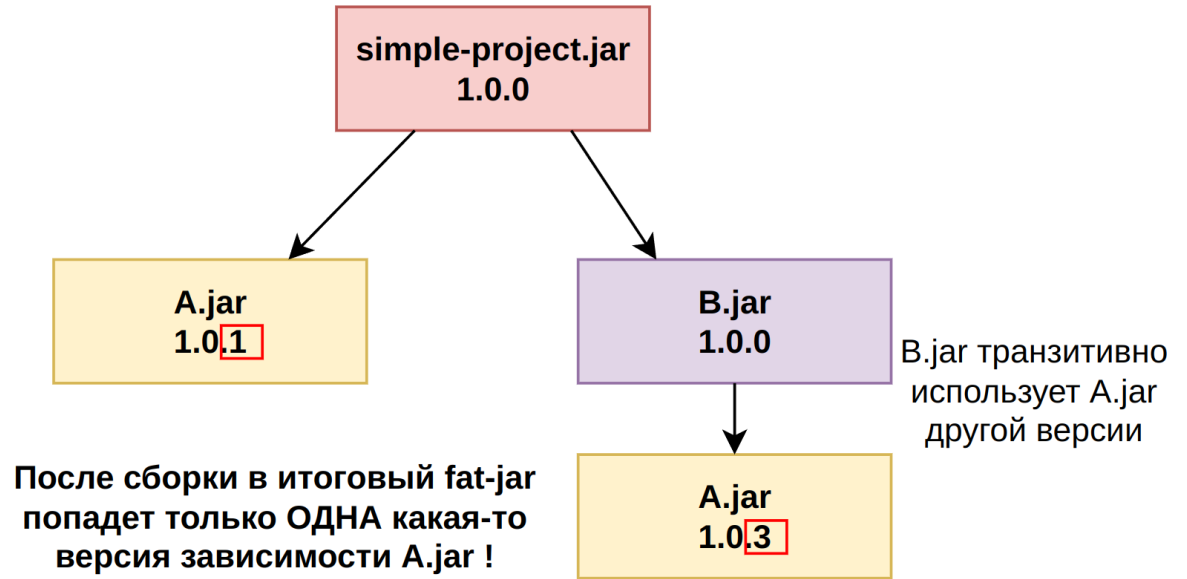
Конкретный пример **транзитивной зависимости**.



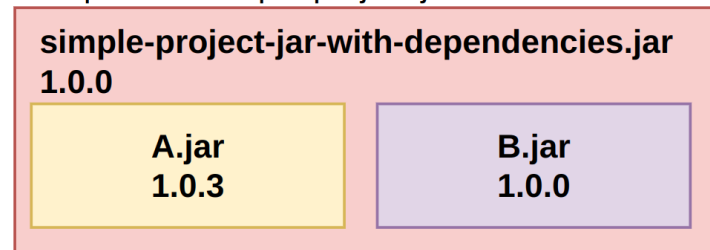
Как Maven решает конфликты зависимостей

В конечный проект, который собирается Maven-ом, попадает только одна версия конкретной зависимости!

Maven даже не сообщит об этом, какую конкретную версию он взял в итоговый .jar.



Например, что может быть в собранном simple-project.jar:



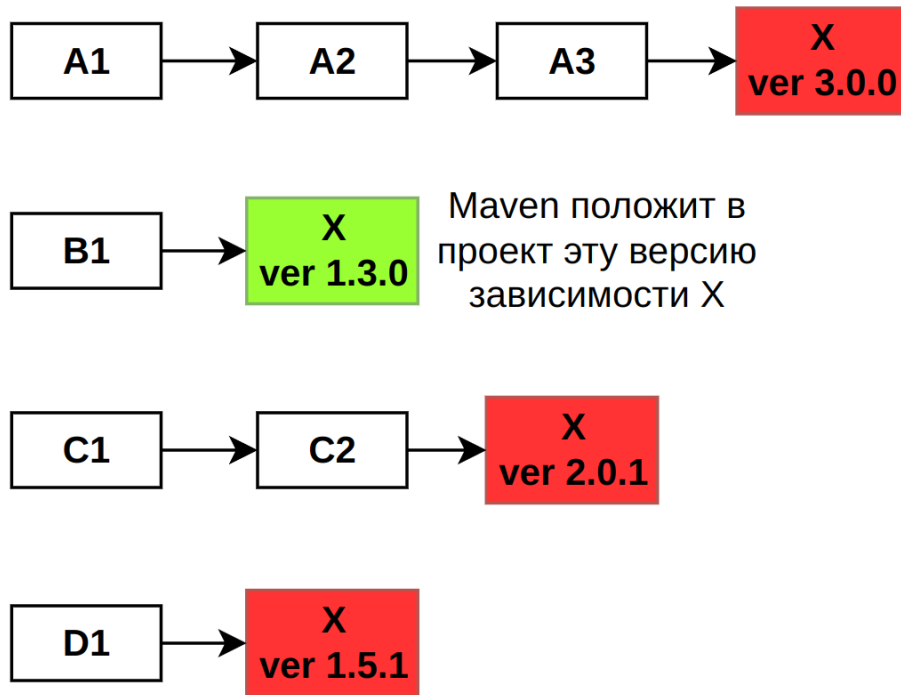
Как Maven решает конфликты зависимостей

pom.xml

```
<dependencies>
  <dependency>
    <groupId>A1</groupId>
    <artifactId>A1</artifactId>
    <version>1</version>
  </dependency>
  <dependency>
    <groupId>B1</groupId>
    <artifactId>B1</artifactId>
    <version>1</version>
  </dependency>
  <dependency>
    <groupId>C1</groupId>
    <artifactId>C1</artifactId>
    <version>1</version>
  </dependency>
  <dependency>
    <groupId>D1</groupId>
    <artifactId>D1</artifactId>
    <version>1</version>
  </dependency>
</dependencies>
```

Глубина дерева транзитивных зависимостей

Порядок объявления



Maven будет использовать в проекте транзитивную зависимость **X ver 1.3.0**

Как Maven решает конфликты зависимостей

Maven берет версию зависимости **ближайшую по порядку объявления и ближайшую по глубине дерева транзитивных зависимостей !**

У плагина **maven-dependency-plugin** есть команда (цель), которая распечатывает дерево всех зависимостей проекта.

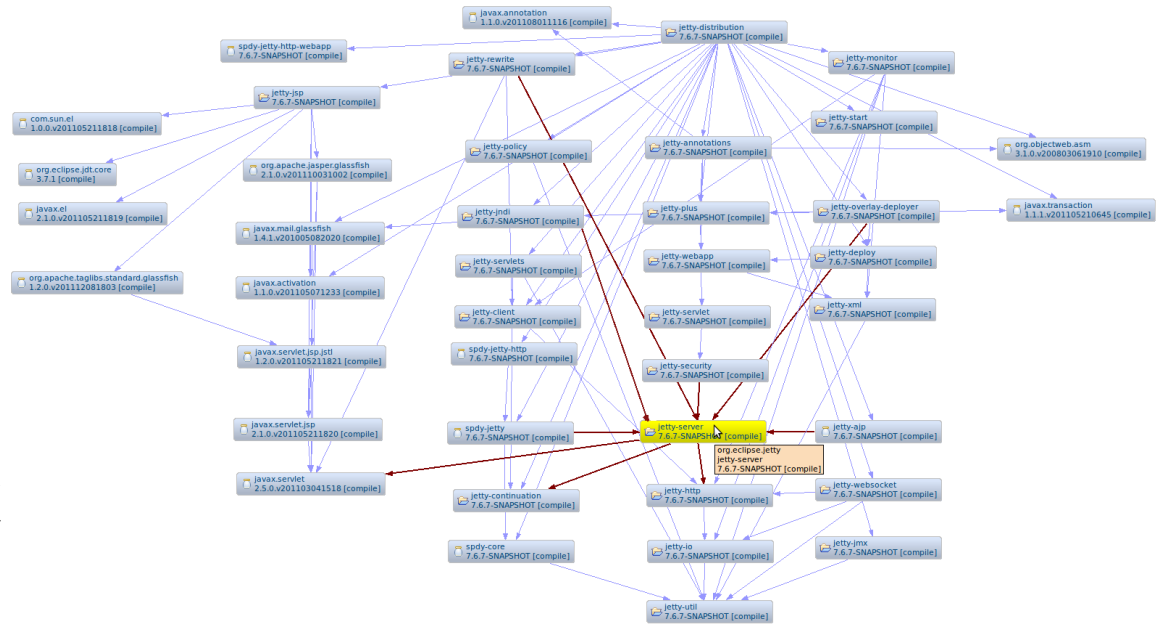
`mvn dependency:tree` (печатает дерево зависимостей, после разрешения конфликтов).

`mvn dependency:tree -Dverbose` (печатает дерево зависимостей до разрешения конфликтов (показывает повторяющиеся)).

Dependency hell (Ад зависимостей)

Dependency hell — это

разрастание графа
взаимных зависимостей
программных продуктов
и библиотек, приводящее
к сложности установки
новых и удаления старых
зависимостей.



Исключение (exclusion) одной из версий транзитивной зависимости

Избежать возникновения «Dependency hell» поможет только **внимательность и анализ каждой подключаемой зависимости**, особенно в проекте с большим количеством зависимостей.

В случае возникновения конфликта между версиями зависимостей необходимо определить, **какая версия останется в вашем проекте, а какая будет исключена** с помощью блока:

`<exclusion> </exclusion>`.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.1</version>
  <exclusions>
    <exclusion>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-core</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Области действия зависимостей (**Scope**)

Еще одним методом уменьшения количества транзитивных зависимостей (уменьшение риска возникновения «Dependency hell») является использование конкретных областей действия зависимостей (**Scope**). Scope указывается в блоке `<scope> </scope>` у зависимости.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.1</version>
  <scope>test</scope>
</dependency>
```

Области действия зависимостей (**Scope**)

В Maven существует 6 Scope-ов.

1) **compile**. Область действия по умолчанию (можно явно не указывать). Зависимость будет **доступна при компиляции программы и при запуске программы** (в runtime). При сборке проекта со всеми зависимостями (jar-with-dependencies) зависимость с областью compile **попадет** в итоговый архив (.jar, .war или .ear).

2) **test**. Зависимость будет **доступна только при компиляции и запуске тестов**. При сборке проекта со всеми зависимостями (jar-with-dependencies) зависимость с областью test **НЕ попадет** в итоговый архив.

Области действия зависимостей (**Scope**)

3) **provided**. Зависимость **будет доступна только при компиляции проекта!** При сборке проекта со всеми зависимостями (jar-with-dependencies) зависимость с областью provided **НЕ попадет** в итоговый архив.

При указании provided мы рассчитываем, что эта зависимость будет «предоставлена» при запуске (в runtime) какой-то другой библиотекой.

Цель использования provided зависимости — уменьшение объема итогового архива проекта (или нашей библиотеки).

Пример, provided может быть указана в проекте web-приложения у зависимости javax.servlet-api, так как данная зависимость будет «предоставлена» контейнером сервлетов, который будет выполнять web-приложение.

Области действия зависимостей (**Scope**)

4) **runtime**. Зависимость будет **доступна только при запуске (в runtime) проекта!** При сборке проекта со всеми зависимостями (jar-with-dependencies) зависимость с областью runtime **попадет** в итоговый архив.

Этот scope используется для пар зависимостей «Интерфейс (api) — Реализация (impl)». Зависимость «Реализация (impl)» подключается с областью runtime.

Цель использования runtime зависимости — разделение в проекте интерфейса и конкретной реализации какой-то библиотеки, возможность «переключения» между реализациями библиотеки.

Пример, библиотека-интерфейс для логирования slf4j-api и ее конкретные реализации: slf4j-simple или slf4j-log4j12 (реализации будут подключаться с областью runtime).

Области действия зависимостей (**Scope**)

5) **import**. Зависимость с областью `import` — это «**pom файл**» (BOM - Bill Of Materials (спецификация зависимостей)), который подключается к нашему `pom.xml`. Область `import` может указываться у зависимости только в блоке `<dependencyManagement>` `</dependencyManagement>` ! Пример будет рассмотрен ниже.

6) **system**. Устаревшая область действия. Зависимость с областью `system` — это зависимость, которая есть только на вашем компьютере (в удаленном репозитории ее нет). Путь до зависимости указывается в блоке `<systemPath>` `</systemPath>`.

Profiles (Профили)

Profiles — это блок в pom.xml, в котором может быть описана одна или несколько конфигураций вашего проекта (Profile).

В блоке Profile могут использоваться почти все сущности pom файла: Properties, Dependencies, Plugins и т. д. Настройки в блоке Profile переопределяют или дополняют Properties, Dependencies и Plugins в pom.xml.

Profiles используются, например, для разделения конфигураций запуска проекта при разработке на локальном компьютере (dev) и при запуске на промышленном сервере (prom).

```
<profiles>
  <profile>
    <id>dev</id> <!-- Id вашего профиля -->
    <activation>
      <!-- Способ активации профиля dev -->
    </activation>
    <properties>
      <!-- Свойства профиля dev -->
    </properties>
    <dependencies>
      <!-- Зависимости профиля dev -->
    </dependencies>
    <build>
      <plugins>
        <!-- Плагины профиля dev -->
      </plugins>
    </build>
  </profile>
</profiles>
```

Profiles (Профили)

Способы активации (применения) профилей.

1. **Явная активация** профиля с помощью флага командной строки:

`-P <id профиля>` (например, `mvn package -P dev`)

2. **«Автоматическая» (неявная) активация** профиля определяется в блоке `<activation> </activation>`. «Автоматическая» активация может быть настроена на следующие признаки.

2.1. По **версии JDK** на машине, запускающей проект.

2.2. По **типу Операционной Системы** на машине, запускающей проект.

2.3. По **наличию системного свойства** (указывается после флага `-D`).

2.4. По **наличию или отсутствию файла или папки** в каталоге проекта.

Более подробная информация в документации Maven:

https://maven.apache.org/guides/introduction/introduction-to-profiles.html#Details_on_profile_activation

Основные варианты упаковки проекта

Тип упаковки указывается в корне `pom.xml` в блоке `<packaging> </packaging>`.

1. **jar** архив (сокращение Java Archive). Этот вариант используется по умолчанию (если ничего не указано в `pom.xml`).

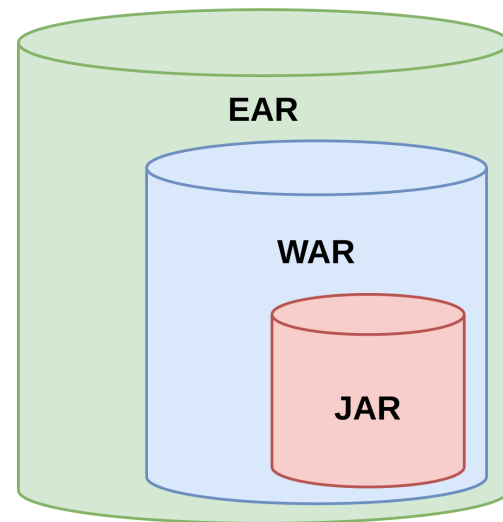
2. **war** архив (сокращение Web application ARchive) используется для упаковки web-приложений.

3. **ear** архив (сокращение Enterprise Application aRchive) используется для упаковки web-приложений Java EE (Enterprise Edition), включает в себя один или несколько war архивов.

4. **maven-plugin** тип упаковки для Maven плагина (если вы пишете свой плагин).

5. **pom** тип упаковки `pom.xml` (BOM — Bill Of Materials (спецификация зависимостей)) для использования в многомодульном проекте (пример рассмотрим ниже).

Также существуют другие варианты упаковки проекта: `ejb`, `rar`, `zip`, `tar` и т.д.

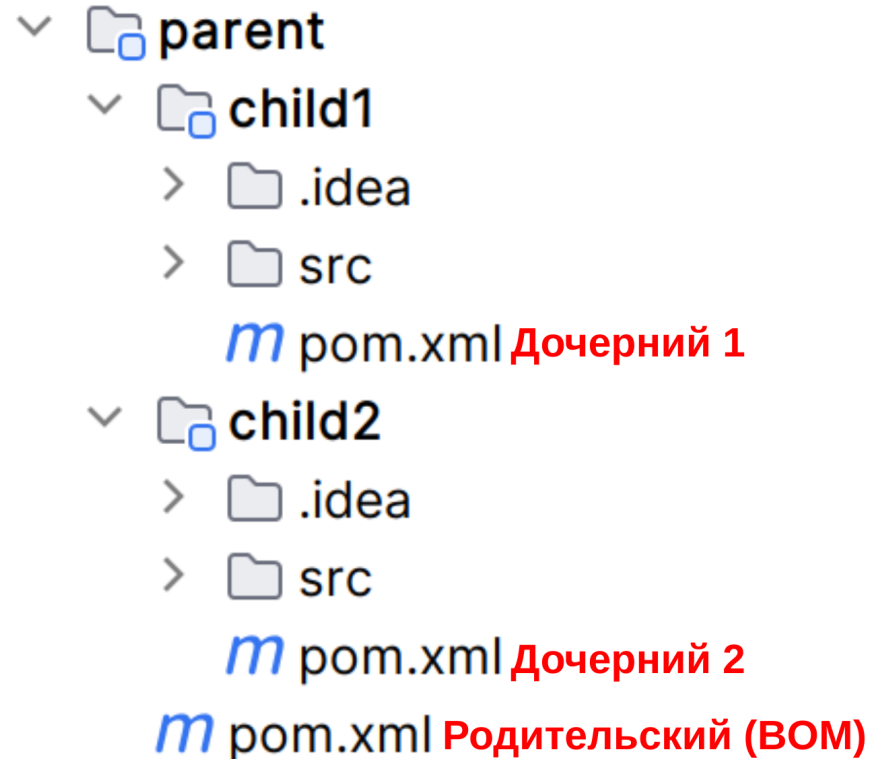


Многомодульные проекты Maven

В Maven существует возможность создания проекта, который будет состоять из нескольких «pom модулей»: родительского и дочерних.

Родительский pom называют BOM файлом (Bill Of Materials (спецификация зависимостей)).

Родительский pom должен иметь **тип упаковки pom**, а также блоки **dependencyManagement** и при необходимости **pluginManagement**.



Пример части родительского pom.xml файла (BOM)

```
<packaging>pom</packaging> <!-- Тип упаковки pom -->
<modules>
  <module>child1</module> <!-- Дочерние модули указываются в блоке modules -->
  <module>child2</module>
</modules>
<dependencyManagement>
  <dependencies>
    <!-- Конкретные версии зависимостей будут доступны дочерним модулям -->
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-math</artifactId>
      <version>2.2</version>
    </dependency>
  </dependencies>
</dependencyManagement>
<build>
  <pluginManagement>
    <plugins>
      <!-- Конкретные версии плагинов и их настройки будут доступны дочерним модулям -->
    </plugins>
  </pluginManagement>
</build>
```

Подключение родительского pom.xml к дочерним модулям

Подключить родительский pom можно 2 способами.

1. Указать родительский pom в блоке parent дочернего pom.xml. Недостаток данного способа — подключить можно только один родительский pom.

2. Указать родительский pom в блоке dependencyManagement со <scope> import </scope>.

Подключение родительского pom к дочерним модулям

Первый способ подключения родительского pom в дочерний pom.xml

```
<!-- Дочерний pom.xml -->

<!-- Подключение родительского pom -->
<parent>
    <!-- Координаты родительского pom -->
    <groupId>org.example</groupId>
    <artifactId>parent</artifactId>
    <version>1.0-SNAPSHOT</version>
</parent>
```

Второй способ подключения родительского pom в дочерний pom.xml

```
<!-- Дочерний pom.xml -->

<!-- Подключение родительского pom -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <!-- Координаты родительского pom -->
            <groupId>org.example</groupId>
            <artifactId>parent</artifactId>
            <version>1.0-SNAPSHOT</version>
            <scope>import</scope> scope import !
        </dependency>
        <!-- Могут быть подключены и другие "родители" -->
    </dependencies>
</dependencyManagement>
```

Пример дочернего pom.xml файла

```
<!-- Подключение родительского pom -->
<parent>
  <groupId>org.example</groupId>
  <artifactId>parent</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>
<dependencies>
  <dependency>
    <!-- Версию зависимости не нужно указывать,
           будет использована версия из родительского pom -->
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-math</artifactId>
  </dependency>
</dependencies>
<build>
  <plugins>
    <!-- При подключении плагина, который определен в родительском pom,
           также не нужно указывать версию плагина и его настройки
           (если нет необходимости их переопределить) -->
  </plugins>
</build>
```

Maven settings.xml

settings.xml — это глобальный файл конфигурации Maven. С помощью настроек в этом файле обычно **переопределяют адреса до удаленных репозиториев** зависимостей (большие компании по разработки ПО имеют свои внутренние хранилища зависимостей).

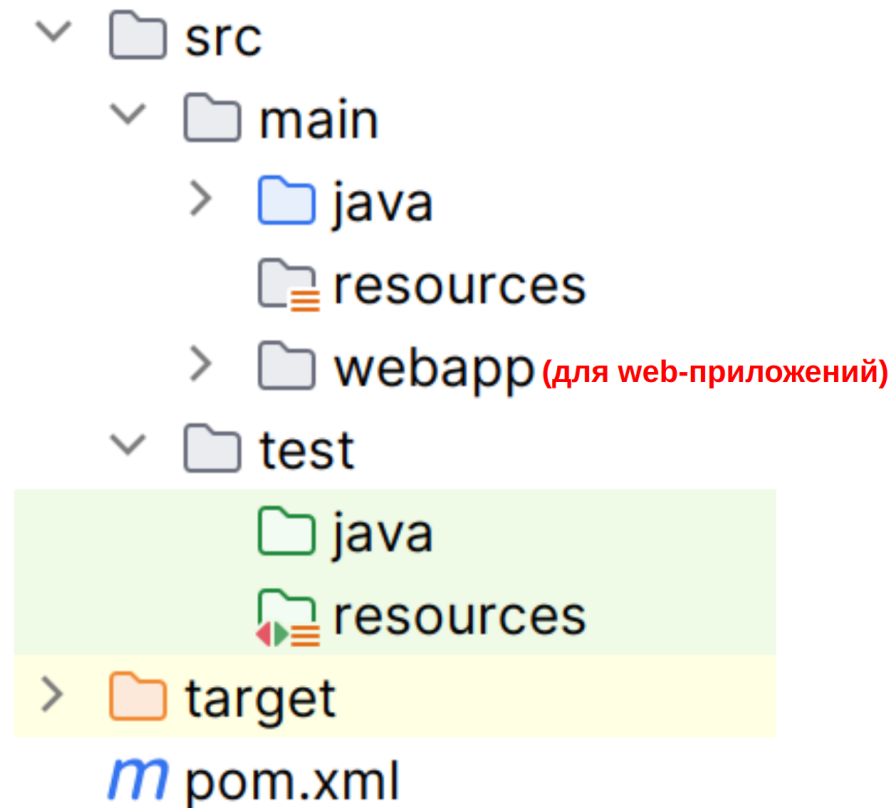
По умолчанию файл settings.xml должен располагаться в папке **/.m2/settings.xml**.

Архетипы Maven

Архетипы Maven — это заготовки структуры проектов, которые позволяют пользователям быстрее создавать новые проекты. Существует множество архетипов для различных вариантов проектов (например для web-приложений).

Пример создания проекта с помощью архетипа maven-archetype-quickstart:

```
mvn archetype:generate -  
DgroupId=com.example -DartifactId=app -  
DarchetypeArtifactId=maven-archetype-  
quickstart -DarchetypeVersion=1.4 -  
DinteractiveMode=false
```



Также Maven стандартизирует структуру проекта