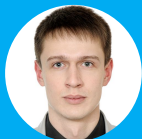


Сборка проектов. Maven и Gradle



Григорий
Вахмистров



Григорий Вахмистров

Backend Developer в Tennisi.bet

План занятия

1. [Вспоминаем прошлое занятие](#)
2. [Что такое сборка проекта?](#)
3. [Инструменты для сборки проектов](#)
4. [Maven](#)
5. [Описание процесса сборки](#)
6. [Gradle](#)
7. [Итоги](#)
8. [Домашнее задание](#)



Что такое сборка проекта?

Сборка в обычной жизни

Представим, что у нас есть цех по сборке автомобиля.

Можем ли мы в одном цеху **одновременно выполнять** множество операций:

1. плавить металл для подготовки,
2. иметь ленту для сбора из частей готового кузова,
3. одновременно красить кузова
4. и выполнять много других операций?

Естественно, нет.

Для каждой части автомобиля есть свой цех по ее изготовлению. При этом, каждая деталь проходит несколько стадий готовности.



Сборка при создании приложения

В создании приложения - точно также.

Приложение состоит из многих составных частей. От написания первой строчки кода до создания полноценного продукта нам необходимо **выполнить ряд последовательных действий**.

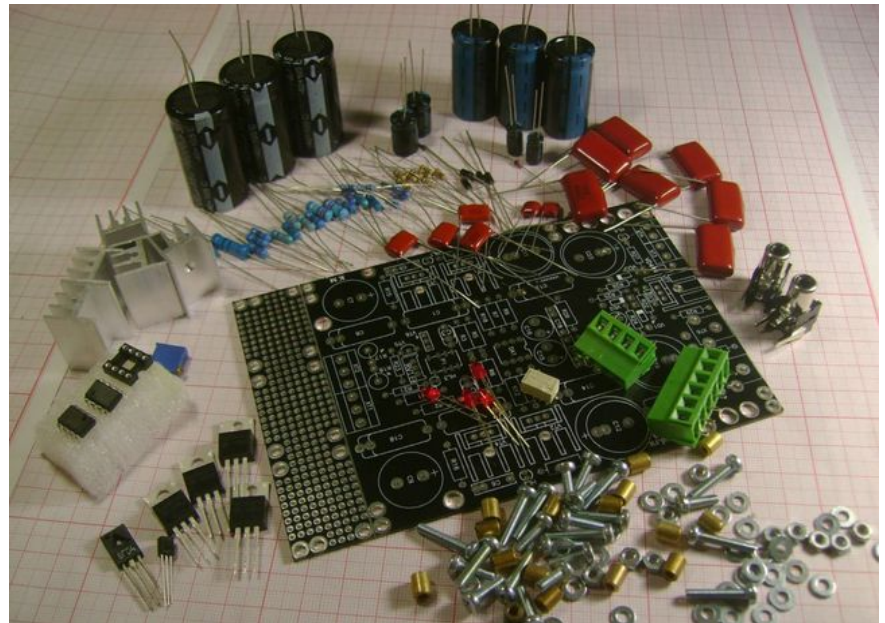
Например:

- скомпилировать код
- протестировать написанный код
- сгенерировать файл MANIFEST с нужными данными
- подготовить готовый файл проекта (собрать JAR и положить в него все библиотеки)
- запустить интеграционные тесты
- выполнить ряд других действий
- и наконец-то, опубликовать готовый файл в репозитории

Что такое сборка?

Под понятием “сборка” мы будем понимать две составляющие - процесс и результат (файл).

1. **Сборка (build)** — двоичный файл, содержащий исполняемый код программы или другой подготовленный для использования информационный продукт.
2. **Сборка** - процесс создания проекта из нескольких частей. Например, упомянутые нами: скомпилировать код, протестировать написанный код, сгенерировать файл MANIFEST и пр.





Какие проблемы решаются при сборке

Кроме правильности написания кода, существует ряд проблем, которые необходимо решать. Их много, но **к основным проблемам сборки относят:**

1. Подключение библиотек
2. Перенос кода и доставка
3. Модульность (функциональность)
4. Версионность продукта
5. Непрерывная интеграция (CI, Continuous Integration)

Давайте посмотрим, что это за проблемы.

Проблема 1. Подключение библиотек

Каждый раз к проекту необходимо **подключать различные библиотеки** - в том числе, и на определенной версии.

Например, для тестирования необходима библиотека `junit-jupiter-engine` версии 5.5.2. Чтобы запустить код, разработчику необходимо установить в **classpath** в IDEA библиотеки:

commons-lang3, slf4j-api, junit-jupiter-api, junit-jupiter-engine, junit-jupiter-params, junit-vintage-engine, hamcrest-all, mockito-core, mockito-junit-jupiter, orika-core, tomcat-jdbc, spring-web, spring-aspects, spring-tx, spring-jms, spring-test, javax.jms-api, guava, lombok, liquibase-core, postgresql, jackson-dataformat-xml.

Также надо знать, какую версию необходимо подключать, а таких библиотек у проекта могут быть десятки и сотни.

Основная проблема в том, что данный процесс отнимает значительное время и это рутинные операции, которых хочется избежать.

Проблема 2. Перенос и доставка кода

Тут у нас содержится две проблемы.

1. **Для конкретного разработчика**

Каждому разработчику при импорте или переносе проекта придется делать рутинный труд по подключению и поднятию проекта. Этот процесс можно автоматизировать с помощью сборщика.

2. **Для автоматизированных систем***

Есть рутинные процессы при создании проекта. Они выполняются автоматизированными системами. Эти системы приходится настраивать вручную - например, писать скрипты. Если мы используем сборщик проекта, это процесс удастся автоматизировать и упростить.

Отсюда необходимо решить проблему удобства переносимости и настройки проекта, как внутри команды между разработчиками, так и для автоматических систем сборки (CI).

* автоматизированные системы - такие системы, как Jenkins, Nexus, Git и др.

Проблема 3. Модульность (функциональность)

В начале лекции мы приводили аналогию с конвейером для автомобилей: кто-то собирает кузов, а кто-то собирает двигатель.

Также и в написании программы: **конкретный элемент может быть отдельным проектом или подпроектом прямо в текущем.**

Например, исходный код модуля может являться *библиотекой* или *jar*. Библиотека в нашем случае является модулем проекта (его подпроектом), но в других случаях - является отдельным самодостаточным проектом.




Проблема 4. Версионность проекта

Какая проблема может возникнуть с версиями проекта?

Мы можем создать текстовый файл и каждый раз вручную проставлять в нем версию проекта. Более того, нам часто требуется добавить не только версию, но и другую мета-информацию, относящуюся к версии проекта - автора, компанию и т.д.

Такая работа обычно не вызывает восторга у разработчиков. Для экономии своего времени, этот процесс мы также сможем автоматизировать.



Версия проекта - это состояние проекта в определенный интервал времени. Например, отличие между собой версий языка программирования: Java 7, Java 8, java 11.

Версионность проекта - процесс ведения и документирования версий проекта.



Версия проекта

Вопрос к аудитории:

Какие способы определить версию проекта вы знаете или можете придумать?



Версия проекта

Вопрос к аудитории:

Какие способы определить версию проекта вы знаете или можете придумать?

Возможные ответы:

Инкремент версии (1, 2, 3, ...N) - последовательность выхода версий, основанная на нумерации версий

Дата - последовательность выхода версий, основанная на точной дате

Тест / не тест - деление версий на две категории: тестируемая версия (альфа-тестирование, бета-тестирование), не тестируемая версия (продакшн-версия)

global.major.minor (2.2.1) - дифференцированная последовательность выхода версий, основанная на различных параметрах

snapshot (2.2.1-snapshot) - последовательность изменения версий, которые меняются почти каждый день (ежедневные сборки, build)




Проблема 5. Непрерывная интеграция (CI)

CI (Continuous Integration, непрерывная интеграция) - автоматизированный процесс сборки и тестирования кода. Вместе со сборщиком направлен на облегчение подготовки релиза.

При работе с непрерывной интеграцией без использования сборщика все процессы подготовки релиза необходимо прописывать с помощью скриптов.

При использовании сборщика мы можем делать это “из коробки” - автоматически. Таким образом, использование сборщика позволяет нам сократить время и расходы на разработку.



Инструменты для сборки проектов



Инструменты для сборки проекта

Немного истории

Make - один из первых инструментов автоматизации сборки, который позволяет писать скрипты сборки, определяя порядок их вызова, этапы компиляции и компоновки для сборки программы.

GNU Make предоставляет дополнительные возможности - зависимости (makedepend), которые позволяют указать условия подключения исходного кода на каждом этапе сборки. Это стало началом автоматизации сборки.



Основной целью была автоматизация вызовов компиляторов и компоновщиков.

По мере роста и усложнения процесса сборки разработчики начали добавлять действия до и после вызовов компиляторов, как например, проверку (англ. check-out) версий копируемых объектов на тестовую систему.




Инструменты для сборки проекта

Немного истории

В 2000-х годах появившиеся инструменты по управлению сборкой сделали более удобным и управляемым процесс автоматизированной сборки.

Ant - начал автоматизировать процесс сборки. Является платформонезависимым аналогом утилиты Make, где все команды записываются в XML-формате. По сравнению с Make, у нее более удобная и декларативная оболочка. Не поддерживает версиюность и доставку.

Maven, Gradle - выходят за рамки действий до и после обработки скриптов и полностью автоматизируют процесс компиляции и компоновки, избавляя от ручного написания сценариев.



Такие инструменты полезны для непрерывной интеграции (CI), когда требуются частые вызовы компиляции и обработка промежуточных сборок.



Инструменты для сборки проекта

Современные сборщики

В современной разработке используют два инструмента автоматизации процесса сборки - **Maven** и **Gradle**.

- **Maven** - популярный инструмент для управления и сборки проектов. Он позволяет разработчикам полностью управлять жизненным циклом проекта - помогает автоматизировать процессы, связанные со сборкой, тестированием и упаковкой проекта.
- **Gradle** - основанная на Groovy популярная система управления сборкой, подходящая для создания проектов на основе Java. Gradle известен своим удобством и простотой использования.

Далее мы подробно рассмотрим эти два инструмента.



Maven



Что такое Maven?

Maven – это программа для управления и сборки проекта, компиляции, создании jar, тестирования с возможностью установки на сервер. Он позволяет разработчикам полностью управлять жизненным циклом проекта.



Преимущества Maven

1. Независимость от OS

Сборка проекта происходит в любой операционной системе. Файл проекта один и тот же.

2. Управление зависимостями

Почти всегда проекты пишутся с использованием сторонних библиотек (зависимостей). Эти сторонние библиотеки используют библиотеки разных версий. **Maven** позволяет управлять такими сложными зависимостями. Это позволяет разрешать конфликты версий и легко переходить на новые версии библиотек.

3. Возможность сборки из командной строки

Это часто необходимо для автоматической сборки проекта на сервере (Continuous Integration).



Преимущества Maven

4. Интеграция со средами разработки

Основные среды разработки на Java легко открывают проекты, которые собираются при помощи Maven. Зачастую проект настраивать не нужно - он сразу готов к дальнейшей разработке.

При этом, с Maven удобно работать в разных IDE. Настраиваемый файл среды разработки и файл для сборки - один и тот же, что позволяет не дублировать данные и избежать связанных с этим ошибок.



Преимущества Maven

5. Декларативное описание проекта

Существуют 2 подхода: императивный и декларативный

Императивный — это описание того, как ты делаешь что-то (то есть должны перечислить все шаги этого процесса), а декларативный — того, что ты делаешь.

Декларативный подход в разработке — это парадигма, в которой задается спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат.

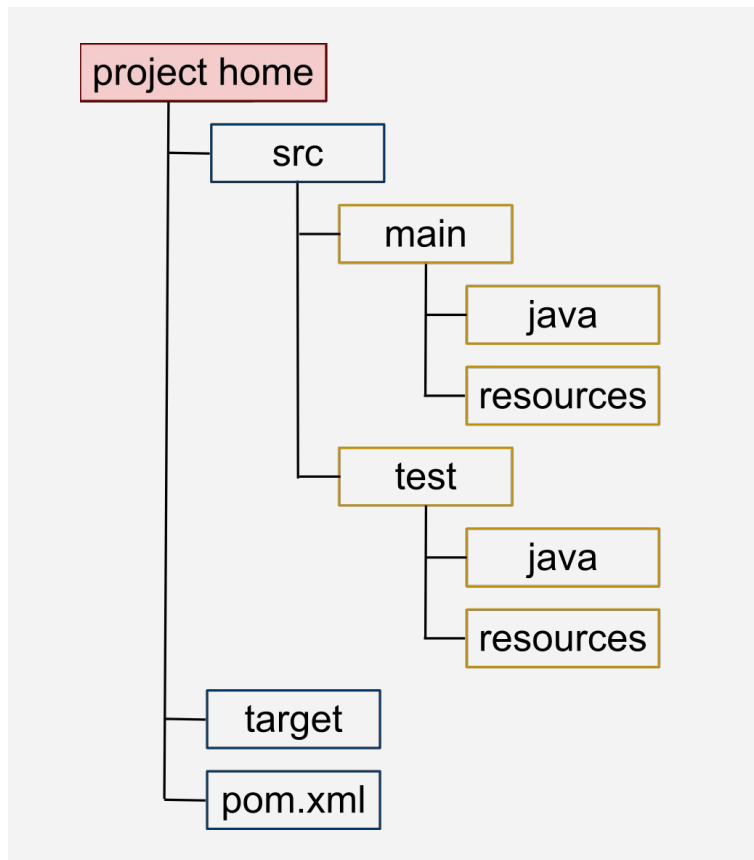
Примеры императивного подхода: C/C++, Java

Примеры декларативного подхода: HTML, SQL

Maven обеспечивает декларативное описание сборки проекта. То есть, в файлах проекта (pom.xml) содержится декларативное описание, а не отдельные команды. Все задачи по обработке файлов Maven выполняет через плагины.

Структура проекта Maven

Maven имеет фиксированную структуру папок в проекте (таблица слева).



<code>/src/main/java</code>	Исходный код
<code>/src/main/resources</code>	Ресурсы
<code>/src/test</code>	Тесты
<code>/target</code>	Дистрибутив
<code>/target/classes</code>	Скомпилированный байт-код



Описание процесса сборки



Из чего состоит описание процесса сборки?

Описание процесса сборки может содержать:

- создание файл сборки `pom.xml`
- описываем 3 составляющие *groupId, artifactId, version*
- указываем тип артефакта (*packing*)
- добавляем зависимости (*dependencies*)
- описываем служебную информацию о сборке (*build*)
- добавляем репозитории (*repository*)
- запускаем сборку

Далее давайте их рассмотрим подробнее.

Описание процесса сборки через pom.xml?

pom.xml - это основной файл, который описывает проект. Вообще могут быть дополнительные файлы, но они играют второстепенную роль.

Как его можно создать?

- Его можно создать и положить самому
- С помощью IDEA
- С помощью команды в консоли

С помощью команды в консоли это выглядит так:

```
mvn archetype:generate -DgroupId=ru.netology -DartifactId=App  
-DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4  
-DinteractiveMode=false
```

Давайте разберёмся, из чего состоит файл pom.xml

pom.xml

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

  <name>App service</name>
  <groupId>ru.netology</groupId>
  <artifactId>App</artifactId>
  <version>0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.3.1</version>
    </dependency>
  </dependencies>
</project>
```



pom.xml

- **project** - это элемент верхнего уровня (корневой) во всех файлах pom.xml Maven. У него есть множество дочерних тегов.
- **modelVersion** - этот элемент указывает, какую версию объектной модели этого POM используют.
- **name** - имя для проекта, которое часто используется в генерации документации Maven

pom.xml

Проектов много, и их объединяют в группы проектов.

Все POM файлы должны иметь три обязательных элемента:
groupId, artifactId, version. (groupId:artifactId:version)

- **groupId** - это идентификатор (ID) группы проектов. Зачастую, это уникальная организация или проект (ru.netology).
- **artifactId** - это идентификатор самого проекта. Чаще всего, это его имя (my-app1).
- **version** - версия проекта. Определяет конкретную версию продукта (0.1-SNAPSHOT).



packing

Элемент **packing** указывает на тип файла, в который будет упакован данный проект - например, в JAR, WAR, EAR и др. Это не только означает, что файл будет создан как JAR, WAR или EAR, но также указывает на конкретный жизненный цикл, который будет использован в процессе сборки.

JAR — обычное Java приложение, которое предназначено для запуска напрямую.

WAR — приложение, которое предназначено для запуска на веб-сервере.

В зависимости от того, какой тип вы выберете, такая будет и сборка (сборка для веб-приложений отличается от “настольной”).

Зависимости

В **Зависимостях** хранится список всех библиотек (зависимостей), которые используются в проекте. Каждая библиотека идентифицируется также, как и сам проект: тройкой *groupId*, *artifactId*, *version*.

Объявление зависимостей заключено в тэг **<dependencies>...</dependencies>**.

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.3.1</version>
  </dependency>
</dependencies>
```

Тэг <build>

Тэг <build> необязательный, для него существуют значения по умолчанию. Этот раздел содержит информацию о самой сборке: где находятся исходные файлы, ресурсы, какие плагины используются.

Давайте рассмотрим пример более подробно.

Тег <build>

<sourceDirectory> - определяет, откуда Maven будет брать файлы исходного кода. По умолчанию это *src/main/java*, но вы можете сами определить, где вам удобно. Директория может быть только одна.

<resources> и вложенные в него тэги определяют одну или несколько директорий, где хранятся файлы ресурсов. Ресурсы, в отличие от файлов исходного кода, при сборке просто копируются. Директория по умолчанию *src/main/resources*

<outputDirectory> - определяет, в какую директорию компилятор будет сохранять результаты компиляции - файлы *.class. Значение по умолчанию - *target/classes*

<finalName> - имя результирующего jar (war, ear..) файла с соответствующим типом расширения, который создаётся на фазе package. Значение по умолчанию — *artifactId-version*.

Ter <build>

```
<build>
<outputDirectory>another_target</outputDirectory>
<finalName>APP</finalName>
<sourceDirectory>src/java</sourceDirectory>
  <resources>
    <resource>
      <directory>${basedir}/src/java</directory>
      <includes>
        <include>**/*.properties</include>
      </includes>
    </resource>
  </resources>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>2.4</version>
    </plugin>
  </plugins>
</build>
```

Репозитории

Репозитории - места, где хранятся ваши артефакты (файлы):

- **Локальный** репозиторий - по умолчанию расположен в *<home_директория>/.m2/repository* и является персональным для каждого пользователя.
- **Центральный** репозиторий - расположен в *<http://repo1.maven.org/maven2/>* и доступен на чтение для всех пользователей в интернете.
- **Внутренний** (корпоративный) репозиторий - дополнительный репозиторий, один на несколько пользователей.

Репозитории

Добавить собственный репозиторий можно так:

```
...  
<repositories>  
  <repository>  
    <id>repo1</id>  
    <url>http://my-site.netology.ru</url>  
  </repository>  
  <repository>  
    <id>repo2</id>  
    <url>http://mirror.netology.ru</url>  
  </repository>  
</repositories>  
...
```



Модульность

Модуль — функционально законченный фрагмент программы.

Модульное программирование — это организация программы как совокупности небольших независимых блоков (модулей), структура и поведение которых подчиняются определенным правилам.

Например, часть текущего проекта будет потом задействована на другом проекте (предположим, методы работы с базой данных (БД)). Для этих целей мы можем вынести функционал работы с БД в отдельный модуль.

Как это описать в **pom.xml**?

Модульность

В **родительском проекте** для этого укажем:

```
...  
<packaging>pom</packaging>  
<modules>  
  <module>app-api</module>  
  <module>app-db</module>  
</modules>  
...
```

А в **дочернем**, который будет создан в текущей директории:

```
...  
<parent>  
  <artifactId>app-api</artifactId>  
  <groupId>ru.netology</groupId>  
  <version>0.1-SNAPSHOT</version>  
</parent>  
...
```


<property>

Properties - по сути, переменные, которые можно использовать в проекте.
Предположим, у нас должна использоваться согласованная версия в 2 зависимостях:

```
...
<properties>
  <junit-jupiter.version>5.3.1</junit-jupiter.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>${junit-jupiter.version}</version>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit-jupiter.version}</version>
  </dependency>
</dependencies>
```

...

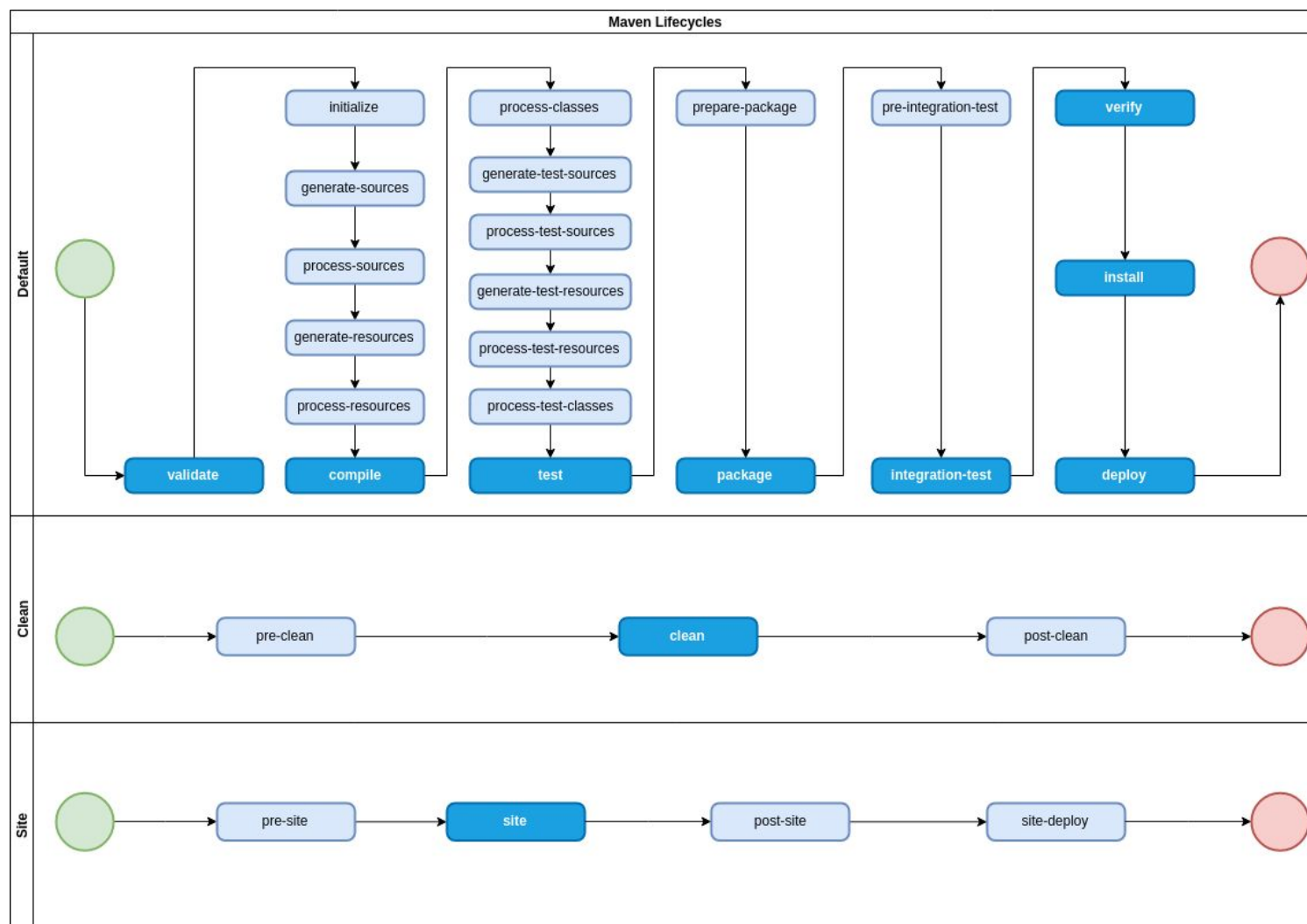


Жизненный цикл сборки

Жизненный цикл сборки - это чётко определённая последовательность фаз, во время выполнения которых должны быть достигнуты определённые цели.

Основной жизненный цикл Maven, который используется для сборки проектов, включает в себя 23 фазы. Далее мы рассмотрим основные из них.

Жизненный цикл сборки в Maven



Основные фазы жизненного цикла сборки (default)

- **validate** - подтверждает, является ли проект корректным и вся ли необходимая информация доступна для завершения процесса сборки.
- **compile** - компилирует исходный код проекта.
- **test** - тестирование, например, с помощью JUnit тестов.
- **package** - создание файла .jar или .war.
- **verify** - выполняет проверки для подтверждения того, что пакет пригоден и отвечает критериям качества.
- **install** - устанавливает пакет в локальный репозиторий, который может быть использован как зависимость в других локальных проектах.
- **deploy** - копирует финальный пакет (архив) в удалённый репозиторий, чтобы сделать его доступным другим разработчикам и проектам.

Фаза жизненного цикла Clean

Отдельно стоит **Clean** - удаление всех созданных в процессе сборки артефактов: .class, .jar и др. файлов.

В простейшем случае результат - просто удаление каталога *target*. Она не выполняется, если не указана в команде запуска сборки.

Более подробно посмотреть все стадии сборки можно на официальном сайте:

http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference

Как запустить конкретную фазу жизненного цикла?

Выполнить фазу сборки можно из IDEA или выполнив в консоли команду **mvn <стадия сборки>**

Тут надо понимать, если выполняем **mvn install**, это значит все предыдущие стадии будут выполнены, включая **install**

Часто для сборки проекта используют команду **mvn clean install**

Тег **<scope>**

Не все библиотеки необходимы на каждой фазе сборки проекта. Например, библиотеки для тестирования необходимы только на этапе тестирования продукта, и в них нет необходимости на этапе работы собранного приложения. Для этих целей используется тег **<scope>** — область видимости. Это позволяет указать сборщику, когда и зачем нужна данная зависимость.

Всего существует 6 областей видимости:

- compile
- provided
- runtime
- test
- system
- import

Тег <scope>

Пример добавления в зависимости:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```




Gradle

Что такое Gradle

Gradle - система управления сборкой, разработанная специально для создания проектов на основе Java. Является комбинацией процедурного и декларативного подхода. Скрипты для Gradle пишутся на языках Kotlin Groovy.

С одной стороны, Gradle позволяет собрать проект, используя конфигурацию как в **Maven**. Но можно, если необходимо, добавить дополнительные действия в процесс сборки.

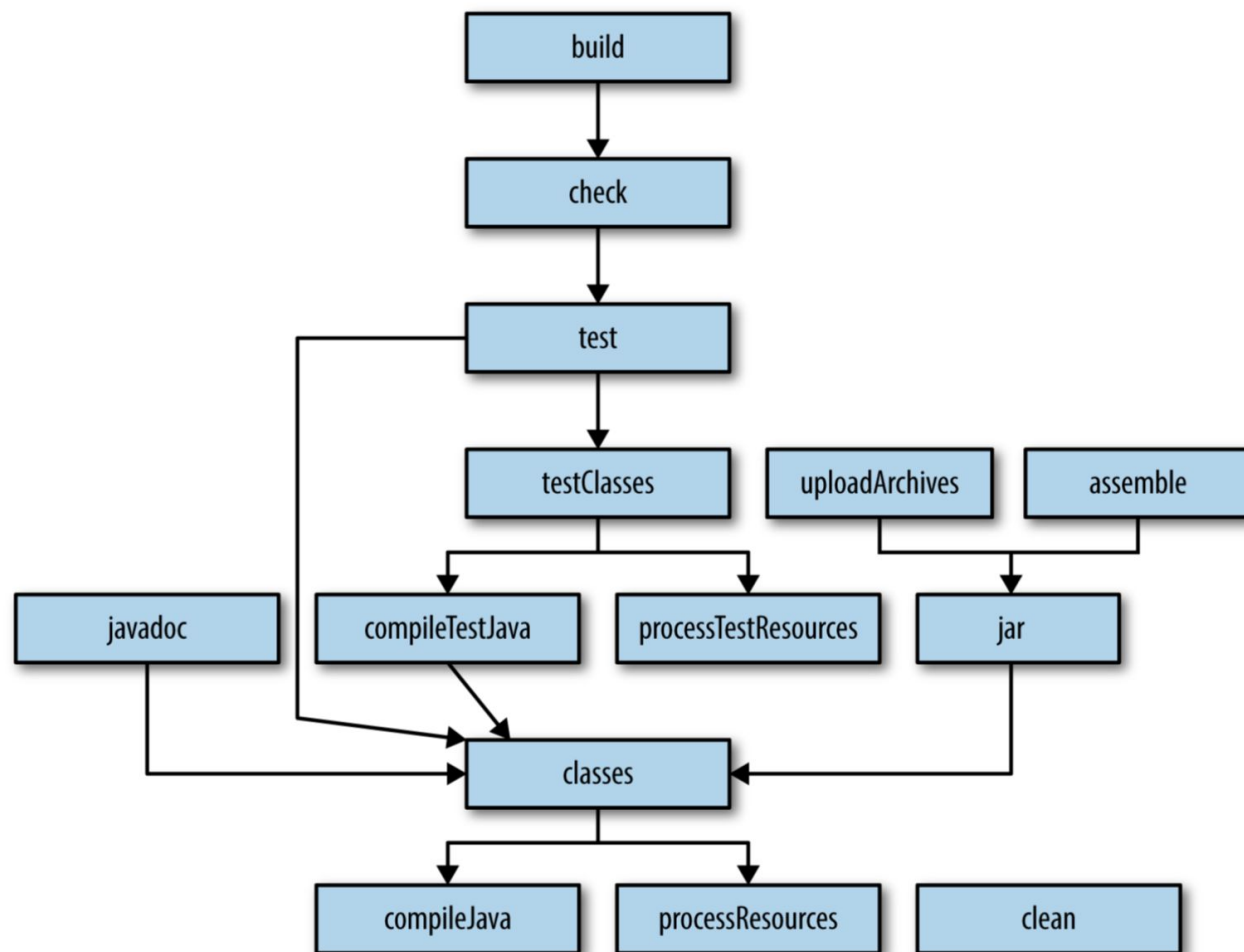


Жизненный цикл сборки

Жизненный цикл сборки проекта на **Gradle**, очень похож на ЖЦ **Maven**. Основные стадии у него такие же. Могут только называться по другому.

Стадии сборки можем увидеть в диаграмме на следующем слайде.

Жизненный цикл сборки



Как создать проект Gradle?

Как можно создать проект на **Gradle**?

- С помощью IDEA
- С помощью команды в консоли `gradle init`

В итоге мы получим несколько файлов для работы с проектом:

- `build.gradle` - основной файл описания сборки (как `pom.xml` в Maven). Он поддерживает описание на 2 языках: Groovy(`build.gradle`) или Kotlin(`build.gradle.kts`)
- `settings.gradle` - описание основных настроек проекта, например имя проекта
- `gradlew`, `gradlew.bat` - скрипты для работы в linux-системах и Windows соответственно
- папка `gradle/wrapper` с 2 файлами: `gradle-wrapper.jar`, `gradle-wrapper.properties` для разворачивания проекта



groupid, artifactid, version

В **Gradle** аналогично, как и в **Maven** указывается при создании проекта тройка: `groupid`, `artifactid`, `version`.

Для `groupid` и `version` используем переменные в `build.gradle`:

```
version = '1.0.0-SNAPSHOT'
```

```
group 'ru.netology'
```

Наименованием артефакта(по умолчанию) в данном случае будет имя проекта в `settings.gradle`:

```
rootProject.name = 'my-app'
```

packing

Для указания на тип файла, в который будет упакован данный проект в **Gradle**, существуют плагины.

Доступные плагины можно искать по следующей ссылке
<https://plugins.gradle.org/>

Для упаковки можно использовать плагин:

- в jar

```
plugins {  
    id 'java'  
}
```

или

```
apply plugin: 'java'
```
- в war

```
plugins {  
    id 'war'  
}
```

Зависимости

Аналогично и в **Gradle** существует возможность добавлять зависимости:

```
dependencies {  
    compile group: 'javax.validation', name: 'validation-api', version:  
'2.0.1.Final'  
}
```

```
// или так  
dependencies {  
    implementation 'javax.validation:validation-api:2.0.1.Final'  
}
```

В современных проектах лучше использовать уже implementation

Зависимости

Предположим, нам необходима зависимость только на этапе прохождения тестов, аналогично как из **Maven** (для этих целей используется `<scope>`) в **Gradle** можно добавить:

```
dependencies {  
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.4.1'  
}
```

Репозитории

Gradle использует **Maven** хранилища для подтягивания зависимостей.

Если в **Maven** центральное хранилище подключено автоматически, то здесь его необходимо указывать.

По аналогии с Maven в Gradle можно указать репозитории:

```
repositories {  
    mavenCentral()  
}
```

Модульность

Многомодульный проект в Gradle строится намного проще.
По аналогии с **Maven** дочерние проекты лежат в папке родительского.
В **родительском проекте** для этого укажем в файле settings.gradle:

```
include 'app-api'  
include 'app-db'
```

А в **дочернем**, в Gradle указывать ничего не надо.

Как запустить конкретную фазу жизненного цикла?

Выполнить фазу сборки можно из IDEA или выполнив в консоли команду из директории проекта **`./gradlew <стадия сборки>`**

Часто для сборки проекта используют команду **`./gradlew build`**

Данная команда включает в себя в том числе и выполнение тестов. Для запуска фазы тестирования отдельно использовать можно **`./gradlew test`**

Для получения только артефакта достаточно выполнить **`./gradlew assemble`**



Итоги



Итоги

- Узнали, что такое сборка приложения
- Рассмотрели, какие сборщики приложений бывают
- Подробно остановились на 2 инструментах:
 - **Maven**
 - **Gradle**
- Посмотрели жизненный цикл сборки проекта на данных инструментах



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера .
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и
пишите отзыв о лекции!**

Григорий Вахмистров