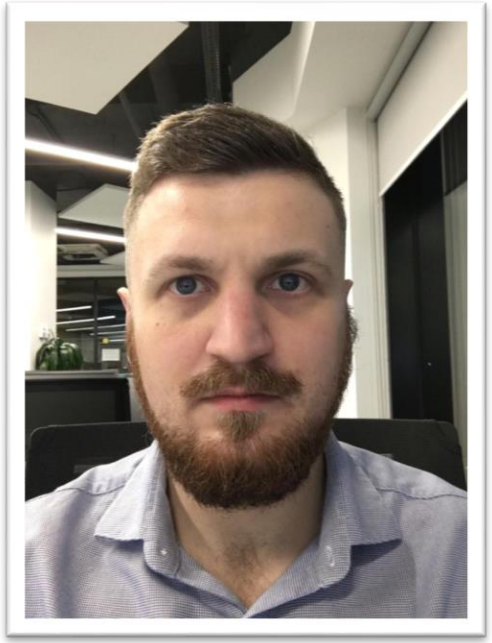
Maven и Gradle: сходства и различия

INTRODUCTION

Yuri Miliutin

Contacts:
+7 9037080267

http://luxoft-training.ru
http://luxoft-training.com



TRAINING ROADMAP: OVERVIEW

This training covers major aspects of Java build tools.

The goal of this training is to became proficient with both Maven and Gradle.

This training is targeted to both junior and senior developers.

Pre-requisites:

- **Basic Java development skills**

TRAINING ROADMAP: STRUCTURE

- 2 Hour session
- Live-coding
- Slides are available for downloading
- Video recording would be available in the next 2 days

SECTION 1:
Build tool conventions

Конвенции: Версия

Концепции Сборки и Версии появилась задолго до IT.
Производству требовалась возможность идентифицировать объекты реального мира.
При помощи версии можно описать явно объект в каталоге и заказать его.
Пример: серийные номера моделей автомобилей.

Автомобиль состоит из компонентов — двигатель, колеса, кузов.
Это **зависимости** автомобиля. И они тоже имеют **версию**.
У кузова есть свои компоненты — двери, определенной модели и версии

Сборка — процесс объединения множества компонентов в готовый продукт.

Конвенции: описание артефакта


```
<dependency>
    <groupId>com.audemarspiguet</groupId>
    <artifactId>royal-oak</artifactId>
    <version>1.5.4.00</version>
</dependency>
```

Group: сайт производителя артефакта

ArtifactId: имя модели

Version: номер версии

Конвенции: Семантическое версионирование

Версия состоит из последовательности чисел.

1.5.4

МАЖОРНАЯ версия, когда сделаны обратно несовместимые изменения API

МИНОРНАЯ версия, когда вы добавляете новую функциональность, не нарушая обратной совместимости

ПАТЧ-версия, когда вы делаете обратно совместимые доработки

При изменении артефакта его версия обязательно увеличивается.

Нельзя выпустить два разных артефакта с одной версией.

Выпуск новой версии зовется Release

Конвенции: выпуск новой версии

1.5.4 => 1.5.**5**

Конвенции: Релизная и -SNAPSHOT версия

Когда этап работы над продуктом завершается — публикуется его релизная версия.
Вида: 1.2.3 или 1.2.3.RELEASE

В бинарный репозиторий нельзя загрузить две поставки с одной релизной версией.

Но как быть в процессе разработки, когда каждый день выходит новая поставка?
Такая версия называется release-кандидатом и имеет вид 1.2.4-SNAPSHOT.

Snapshot-ы одной версии можно перезаписывать новой поставкой.
Когда SNAPSHOT прошел тестирование — публикуется релизная версия:
1.2.4-SNAPSHOT => 1.2.**4**

Конвенции: Релизная и -SNAPSHOT версия

1.2.4-SNAPSHOT **1.2.4-SNAPSHOT** **1.2.4**

Разработка закончена.

Версия **1.2.4** отправляется на выставку

Транзитивные зависимости и их конфликты

- **Транзитивные** зависимости — зависимости зависимостей.

  cellphone => sim-card => mobile-operator

  => pricing-plan

  => account

  => services

Фундаментальная проблема: Dependency hell

- Фундаментальная проблема систем сборки:

  Зависимость A имеет зависимость B, которая зависит от D версии **1.0**

  Зависимость A имеет зависимость C, которая зависит от D версии **2.5**

- Какую версию D в итоге использовать при сборке поставки? 1.0? 2.5?
  А может положить сразу обе, пусть Java сама разберется?

  Java не разберется. Конвенция загрузки классов Java:
  Если в поставке (war, jar) есть два класса с одинаковым именем — будет загружен только один из них. Java не гарантирует какой именно. Это непредсказуемо.
  В промышленном проекте сотня зависимостей. И еще сотня транзитивных.
  Конфликт версий — это вопрос времени.

Анатомия катастрофы

Дорогая!
Я на работу!
Тороплюсь!

Принеси часы!

Runtime: Ожидания


```
watch.getCurrentTimeWithSeconds()
```

Пора свистеть!

Runtime: Реальность


```
watch.getCurrentTimeWithSeconds()
```

java.lang.NoSuchMethodError:

Процесс сборки 15 лет назад

HelloWorld-приложение можно написать и без системы контроля версий.

Чтобы загрузить зависимость (например драйвер доступа к Базе Данных) нужно:

1. Зайти на сайт Базы данных
2. Зарегистрироваться
3. Скачать jar-файл
4. Добавить его в проект
5. Сохранить вместе с кодом или в специальную папочку.

Но что если у вас сотня зависимостей?

Maven добавил ряд конвенций, которые сильно упростили жизнь.

SECTION 2:
Live-coding Maven

Конвенции Maven: meta-информация

Поставка артефакта состоит из двух частей:

1. Сама библиотека -- jar или war файл с набором классов

my-library.**jar**

2. Pom-файла с описанием артефакта и списком версий его зависимостей

my-library.**pom**

Артефакты загружаются не в “папочку”, а в бинарный репозиторий -- сетевое хранилище, каталог артефактов.

Конвенции Maven: .m2 и settings.xml

Это медленный процесс -- при каждой сборке заново загружать по сети зависимости из бинарного репозитория.

Поэтому Maven сохраняет загруженные артефакты на диск локального компьютера в папку с именем **.m2**

Для доступа в бинарный репозиторий может потребоваться пароль.

Хранить пароль в исходном коде проекта нельзя -- тогда его сможет увидеть каждый.

Security Violation!

Поэтому адреса репозиториев и пароли к ним хранятся на локальной машине в файле maven/**settings.xml**

Конвенции Maven: команды

clean -- очищает папку target с результатами предыдущих сборок

compile -- компилирует классы. Превращает файлы .java в файлы .class в папке target

package -- упаковывает все .class-файлы в jar/war артефакт.

install -- сохраняет этот артефакт в локальную папку .m2

deploy -- публикует артефакт в удаленный бинарный репозиторий команды/компании

SECTION 3:
Publishing

Конвенции Maven: Nearest First conflict resolution

Проблема: в графе зависимостей есть две версии одного артефакта.
Какую выбрать для поставки?

Решение:
Nearest: Выбираем версию наиболее близкую к текущему проекту на уровнях графа зависимостей.

First: если на одном уровне транзитивности все равно несколько версий?
Берем ту, чья строка декларации находится выше в pom-файле.

Кто первый -- тот и папа.
Конвенция получилась очень спорной -- мы можем предпочесть новой версии старую.
Гораздо чаще используется механизм dependencyManagement.

Конвенции Maven: Dependency Management

Maven предоставляет возможность в теге <dependencyManagement> явно указать и зафиксировать версию, которая будет использована при сборке.

Это делает сборку предсказуемой и повторяемой, но не защищает от всех проблем.

Что если, в графе зависимостей есть более новая версия, а в dependencyManagement мы указали старую?

Кто знает... возможно ничего.
А может и **NoSuchMethodError** / **NoClassDefFound** в Runtime.
Пока вы летите в отпуск на высоте 10k метров.

Dependency Hell

Как избежать конфликта зависимостей, перестать беспокоиться и начать жить?

1. Разделить большой проект на группу проектов поменьше.

Меньше логики и кода => меньше зависимостей => ниже риск их конфликта

2. Писать тесты и запускать их.

Тесты проверяют runtime-поведение проекта. А значит могут выявить проблемы, вызванные конфликтами

3. Осуждать разработчиков библиотек, нарушающих обратную совместимость

4. Использовать Bill of Materials (BOM)

Конвенции Maven: Bill of Materials

Крупные библиотеки могут состоять из десятка модулей и сотни транзитивных зависимостей. Пример: Spring Framework.

Вместе с такой библиотекой идет pom-файл (i.e. spring-boot-dependencies) в котором перечислены версии зависимостей, на которых поставка была протестирована.

На этих версиях точно не будет конфликтов.

Этот pom-файл можно добавить как parent или в секцию dependencyManagement (scope: import).
Таким образом версии в пакете зависимостей будут консистентны.

Проблемы Maven: Why do we mvn:clean?

Сборка большого проекта с запуском всех тестов может занимать час.
И этот час вы проведете вглядываясь (философски) в логи сборки Maven.
Хорошо если только вы -- поставку на стенд может ждать вся команда.

И вот вы задаетесь вопросом:
А почему именно: `mvn clean install`?
Я же всего пару классов поменял — зачем собирать модули, которые не менялись год?
Зачем запускать на них длительные тесты?
Можно просто сделать `install`?

Тимлид, убеленный благородной сединой, скажет:
Нет. `mvn clean install`. Просто поверь мне.
И будет прав.

Проблемы Maven: Why do we mvn:clean?

Вам хочется инкрементальную компиляцию — собирать только те классы, что изменились.
Запускать только тесты, зависящие от этих классов.
Вы понимаете, что это возможно. Но Maven так не умеет.
Он не отслеживает результаты предыдущих сборок.

Пока вы тратите время на `clean install` в башнях из серого стекла ваш продукт ждут люди в дорогих костюмах.
Они не принимают оправданий и не знают о системах сборки.
Им нужен результат и он нужен сегодня.

Gradle может вам помочь.

Конвенции Gradle: UP-TO-DATE

Каждая из задач Gradle имеет свои `@Inputs` и `@Outputs`.

Если исходный код модуля не изменился — Gradle не станет заново компилировать его классы. Запускать тесты тоже не станет — ведь классы не изменились.

Сборка станет в десятки раз быстрее.

И чем крупнее проект — тем быстрее станет сборка.

Важно: UP-TO-DATE проверка происходит на уровне модуля.

Не храните весь код в одном модуле.

Разделяйте проект на небольшие, изолированные модули.

Конвенции Gradle: Build cache

- Вы сделали gradle clean и очистили сборочную папку. Так вышло.
- build
- Gradle сделает up-to-date проверку
- Если исходные данные не изменились — он загрузит @Outputs из локального кеша. Без затрат времени на сборку и запуска тестов.

Конвенции Gradle: Incremental compilation

- Gradle может проводить up-to-date проверки не только на уровне модуля, но и на уровне конкретного java-файла.
- И собирать только нужные
- Это может ускорить сборку. А может и замедлить — придется проверить

Конвенции Gradle

Конвенции Maven распространены и Gradle их переиспользует:

package => build

.m2 => build cache

settings.xml => init.gradle

pom.xml => build.gradle

<modules> => settings.gradle

Параллельная сборка:

-T 1.5C => --parallel

<dependencyManagement> => dependencies.constraints (*)

Gradle: highest version in graph conflict resolution

Проблема: в графе зависимостей есть две версии одного артефакта.

Какую выбрать для поставки?

Решение:

Выбираем наиболее семантически высокую версию из представленных в графе.

1.3.5 или **1.2.8** ?

Gradle выберет **1.3.5**.

Верим, что разработчики библиотек не нарушают обратную совместимость

Gradle: dependency management plugins

До версии 5 (2019 год) в Gradle не было аналога `dependencyManagement` из Maven.

Так появились `spring-dependency-management-plugin` и `nebula-dm-plugin`.

В версии Gradle 5 появились механизмы для управления конфликтами зависимостей и BOM.

Рекомендуется использовать механизмы Gradle —`platform()` и `constraints`.

Плагины для управления зависимостями конфликтуют с нативным механизмом и не рекомендуются к использованию.

Конвенции Gradle: constraints

`Constraints` это аналог `<dependencyManagement>`.

Он позволяет явно указать версию зависимости и не дублировать ее в дочерних модулях.

Отличие от Maven:

Версия будет добавлена в граф зависимостей.

Но использовать при сборке Gradle будет самую высокую версию в графе.

Она может оказаться транзитивной.

Чтобы строго зафиксировать версию в constraints нужно передать флаг force=true

Конвенции Gradle: platform

- Gradle может импортировать список версий из pom-файла (bill of materials).

  `platform (my.company:my-bom:1.2.3)`

- Отличие от Maven:

  Версии будут добавлены в граф зависимостей.

  Но использовать при сборке Gradle будет самую высокую версию в графе.

  Она может оказаться транзитивной.

- Чтобы строго зафиксировать версию вместо `platform()` нужно использовать `enforcedPlatform()`.

  Тогда при разрешении конфликтов версии из BOM будут иметь приоритет.

Конвенции Gradle: Group substitution

Иногда бывает, что библиотека меняет не версию, а **название**.

У вас две зависимости с одинаковыми именами классов.
Gradle не видит конфликта — имена библиотек отличаются.

Нужно сказать *“Артефакт с именем A и артефакт с именем B это одна библиотека”*
В поставке будет использована только один из артефактов — обладающий наивысшей версией.

```
modules {
    module("com.google.collections:google-collections") {
        replacedBy("com.google.guava:guava", "collections is now part of Guava")
    }
}
```

Конвенции Gradle: Будущее

В версии 6 в Gradle появился свой формат описания meta-информации (аналог .pom-файла).

Сейчас Gradle публикует в бинарный репозиторий два meta-файла: pom и .gradle.

Возможности нового формата:

1. Загрузка различных наборов зависимостей в зависимости от условий (например версии Java)

2. Возможность создания опциональных условий (capabilities).

Например: “Загрузи библиотеку spring-data-jpa:1.2, настроенную для работы с базой данных Oracle”.