

ВЫСТРАИВАНИЕ ПРОЦЕССА НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ (CI)

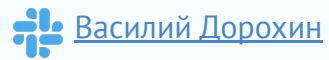


ВАСИЛИЙ ДОРОХИН



ВАСИЛИЙ ДОРОХИН

QuadCode, QA Engineer





ПЛАН ЗАНЯТИЯ

1. [Continuous Integration](#)
2. [Процесс](#)
3. [Maven LifeCycle & Plugins](#)
4. [Задача](#)
5. [Автотесты \(Surefire\)](#)
6. [Покрытие кода \(JaCoCo\)](#)
7. [Другие плагины](#)
8. [Итоги](#)



CONTINUOUS INTEGRATION



CONTINUOUS INTEGRATION

Оригинал: *Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily — leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.*

Martin Fowler

Вольный перевод: *Continuous Integration (далее — CI) — это практика разработки ПО, при которой участники команды интегрируют изменения настолько часто, насколько это возможно (как минимум, ежедневно или несколько раз в день). Каждая интеграция верифицируется автоматической сборкой (включая тесты) для определения ошибок интеграции настолько быстро, насколько это возможно. Мартин Фаулер*

CONTINUOUS INTEGRATION

Для нас — это будет практика (подход к разработке), при которой для каждого изменения кода (`git push`) будет автоматически запускаться конвейер сборки и тестирования.

По результатам сборки мы будем видеть: либо сборка «**Success**» и код можно интегрировать, либо «**Failed**» и код нуждается в доработке:

Update StatisticsServiceTest.java

 **coursar** committed 1 minute ago **✗** 

Update maven.yml

 **coursar** committed 3 minutes ago **✓** 

Create maven.yml

 **coursar** committed 6 minutes ago **✓** 



SUCCESS VS FAILED

Ключевое — это возможность быстро принимать решения.

Вопрос к аудитории: давайте подумаем, если у нас будет информация о том, что при внедрении новой функции 5% от общих тестов не прошло, можем ли мы быстро принять какое-либо решение?

SUCCESS VS FAILED

Информация о том, что не проходит 5% тестов без дополнительного разбирательства сообщает только о том, что не все тесты проходят.

Поэтому там, где важна скорость принятия решений, всё делится только на два возможных варианта:

- **Success** — всё ок;
- **Failed** — не ок (даже если хотя бы один тест упал).

Исходя из этого принимаются решения о возможности интеграции изменений в основную ветку.



CONTINUOUS DELIVERY

Continuous Delivery (непрерывная поставка) подразумевает ещё один шаг вперёд в вопросе автоматизации.

Мы **доверяем автотестам настолько**, что на базе их результатов автоматически **формируем и выкладываем релиз** (например, публикуем библиотеку в Maven Central) или внутреннем репозитории.

Затем его (релиз) можно подвергать исследовательскому тестированию, тестированию безопасности и т.д., но ключевое — мы **формируем релиз на базе решения автотестов**.



CONTINUOUS DEPLOYMENT

Continuous Deployment (непрерывное развёртывание) подразумевает ещё один шаг вперёд в вопросе автоматизации (ещё дальше).

Мы уже **доверяем автотестам настолько**, что на базе их результатов автоматически **выкатываем релиз на Production!**

Т.е. мы, фактически, исключаем ручное тестирование (его можно проводить уже на бою).

Благодаря этому, мы можем внедрять хоть по 10-20 новых функций (feature — фич) в день.

Здесь достаточно много психологических моментов, но требования развития (и конкуренция) требуют максимальной оптимизации.



CI/CD

В этой лекции мы с рассмотрим практику CI (Continuous Integration) на базе сервиса [GitHub Actions](#).

GitHub Actions — это бесплатный сервис, позволяющий запускать определённый набор операций на основе наших действий в GitHub.

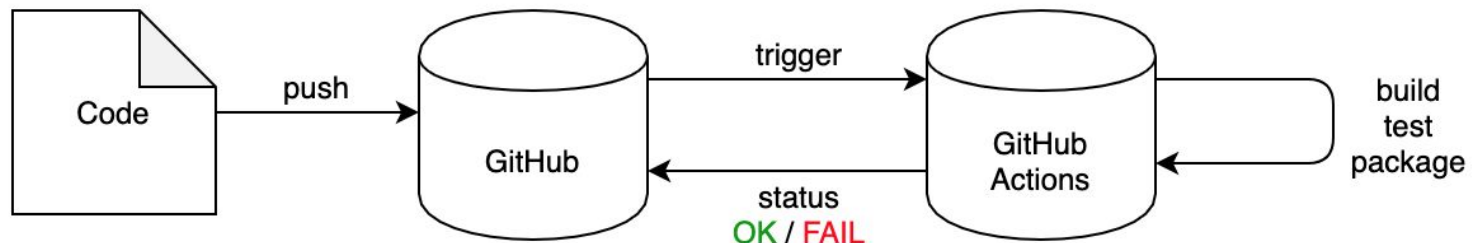
На других лекциях этого и последующих курсов мы рассмотрим другие популярные облачные и in-house решения.



ПРОЦЕСС

ПРОЦЕСС

Вот так будет выглядеть общая схема:



По шагам:

1. Код пушится (`git push`) в репозиторий.
2. На каждый пуш срабатывает запускаяется автоматическая сборка в CI.
3. Сервер CI клонирует целиком репозиторий и запускает сборку, тесты и любые другие настроенные проверки.
4. Сервер CI уведомляет GitHub о результатах сборки.



ПРОЦЕСС

Большинство команд идёт дальше и переносит этот процесс на ветки: разрабатывает новые функции и внедряет изменения в отдельных ветках (`git branches`) и «мёржат» (`git merge`) код **только при наличии положительной обратной связи от CI**.

ПРОЦЕСС

Рассмотрим всё на примере построения CI вокруг сервиса статистики для анализа доходов организации (код на след.слайде).

Для реализации нашего процесса мы:

1. Создадим пустой проект на базе Maven.
2. Подключим туда нужные зависимости.
3. Запустим (`git push`) всё в GitHub.
4. Настроим CI.
5. Будем пушить (`git push`) каждое изменение и смотреть на результаты сборки.

MAVEN PROJECT

Как создавать проекты на базе Maven вы знаете:

- GroupId: ru.netology;
- ArtifactId: statistic.

Ключевые настройки:

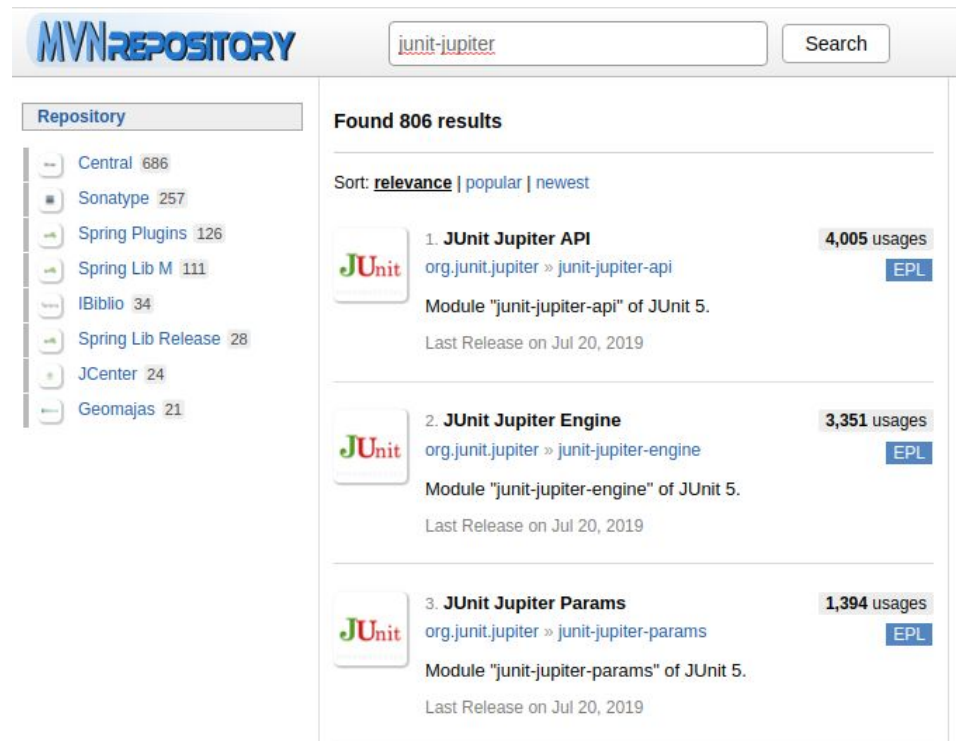
```
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```


DEPENDENCIES

Мы, конечно, можем скопировать зависимости из предыдущего проекта. Но пришло время научиться их искать самим.

Естественно, искать их нужно на Maven Central.

Для поиска в Maven Central и других репозиториях можно воспользоваться сайтом <https://mvnrepository.com/>.



The screenshot shows the Maven Repository website. The search bar at the top contains the text 'junit-jupiter'. Below the search bar, the results are sorted by 'relevance'. The first three results are listed:

- 1. JUnit Jupiter API** (4,005 usages)
org.junit.jupiter » junit-jupiter-api
Module "junit-jupiter-api" of JUnit 5.
Last Release on Jul 20, 2019
- 2. JUnit Jupiter Engine** (3,351 usages)
org.junit.jupiter » junit-jupiter-engine
Module "junit-jupiter-engine" of JUnit 5.
Last Release on Jul 20, 2019
- 3. JUnit Jupiter Params** (1,394 usages)
org.junit.jupiter » junit-jupiter-params
Module "junit-jupiter-params" of JUnit 5.
Last Release on Jul 20, 2019

On the left side of the page, there is a sidebar with a list of repositories and their artifact counts:

- Central 686
- Sonatype 257
- Spring Plugins 126
- Spring Lib M 111
- IBiblio 34
- Spring Lib Release 28
- JCenter 24
- Geomajas 21



JUNIT JUPITER

Как вы видите, самые популярные это:

- `junit-jupiter-engine` — ядро JUnit Jupiter;
- `junit-jupiter-api` — API для написания автотестов (готовый набор классов, аннотаций и т.д.);
- `junit-jupiter-params` — API для написания параметризованных автотестов.

JUNIT JUPITER

Можно зайти в каждую зависимость:



1. JUnit Jupiter API

org.junit.jupiter » junit-jupiter-api

Module "junit-jupiter-api" of JUnit 5.

5,216 usages

EPL

Выбрать версию (последнюю стабильную):

	Version	Repository	Usages	Date
5.7.x	5.7.0-M1	Central	27	Apr, 2020
5.6.x	5.6.2	Central	206	Apr, 2020
	5.6.1	Central	252	Mar, 2020
	5.6.0	Central	453	Jan, 2020
	5.6.0-RC1	Central	22	Jan, 2020
	5.6.0-M1	Central	58	Oct, 2019

JUNIT JUPITER

И на вкладке Maven получить заветные строки:

Maven Gradle SBT Ivy Grape Leiningen Buildr

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.6.2</version>
  <scope>test</scope>
</dependency>
```

SEMANTIC VERSIONING

В мире разработки ПО многие команды используют специальный подход к версионированию — [Semantic Versioning](#).

Версия продукта `major.minor.patch` разбивается на три ключевых составляющих:

- `major` — мажорная версия, в которой есть нарушающие совместимость изменения API;
- `minor` — минорная версия, в которой нет нарушающих совместимость изменений API;
- `patch` — багфиксы, не нарушающие совместимость API.

СОВМЕСТИМОСТЬ API

API (Application Programming Interface) — набор классов и методов*, предоставляемых нам библиотекой для использования из нашего кода.

Нарушение совместимости означает, что предоставляемый нам интерфейс меняется внешне (т.е. мы не можем использовать его как раньше):

// Например, было:

```
public long findMax(long[] purchases) { ... }
```

// Стало:

```
public long max(long[] purchases) { ... }
```

// Теперь и нам в своём коде придётся везде findMax на max поменять

Примечание:* не обязательно только классов и методов, это может быть HTTP API, определяющее набор допустимых параметров и их типов.

ВЕРСИИ

Важно: всегда выбирайте версии только с цифрами, без всяких суффиксов вроде:

- RC1, RC2 и т.д. — это Release Candidate (кандидат в релизы): почти готовый к применению выпуск
- M1, M2 и т.д. — это Milestone (веха): более-менее стабильный выпуск
- beta, alpha — бета и альфа-выпуски, соответственно
- SNAPSHOT — версия в разработке (вообще без каких либо гарантий)

Общую идеологию можно почитать <https://semver.org>, но не все ей следуют буква в букву.

JUNIT JUPITER

Q: но мы же раньше подключали только одну зависимость и всё работало?

Почему сейчас нужно три?

A: хороший вопрос. Есть так называемые метапакеты, задача которых предоставить одну зависимость, которая подключает другие (самые часто используемые) зависимости.

Именно такой зависимостью и является `junit-jupiter`:



4. **JUnit Jupiter (Aggregator)**

`org.junit.jupiter` » [junit-jupiter](#)

823 usages

EPL

Module "junit-jupiter" of JUnit 5.

Если вам нужны все три артефакта (`engine`, `api` и `params`), то можно использовать её.



JUNIT JUPITER

Q: где мне подробнее узнать о зависимостях и всём остальном?

A: из наших лекций, официальной документации и из полезных статей, например на dzone.com

SUREFIRE PLUGIN

Добавим Surefire Plugin:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
    </plugin>
  </plugins>
</build>
```

Вопросы к аудитории:

1. Зачем мы добавляем Surefire Plugin?
2. Что такое *Effective POM* и как он будет работать после добавления Surefire Plugin?
3. Как удостовериться, что сейчас (пока без кода) у нас всё работает?

GITHUB ACTIONS

Инициализируем пустой репозиторий, добавим `.gitignore` и запустим всё на GitHub.

Перейдём на вкладку GitHub Actions нашего репозитория:

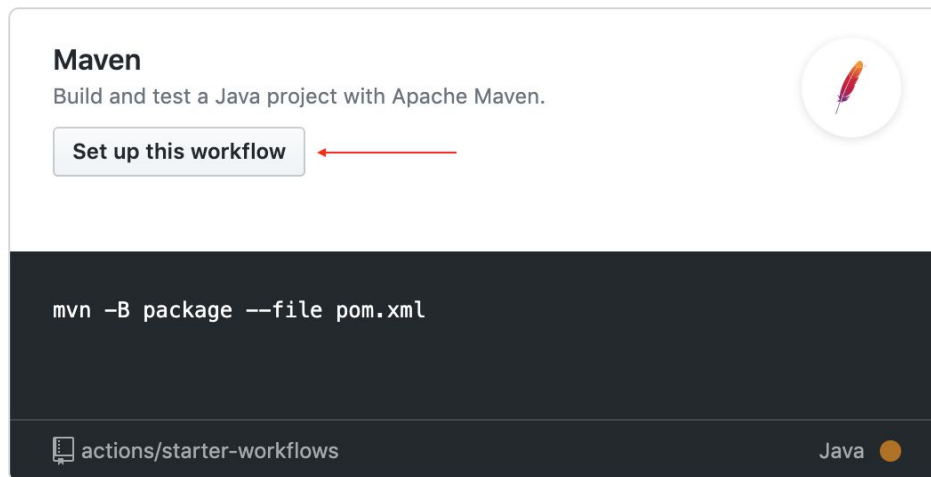


Get started with GitHub Actions

Choose a workflow to build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way *you* want.

GITHUB ACTIONS & MAVEN

В GitHub Actions уже предусмотрен шаблон для Maven проектов:



Для начала мы всё установим и настроим, а потом будем разбираться, как всё работает.

statistic / .github / workflows / maven.yml

<> Edit new file

Preview

```
1  name: Java CI # как называется Workflow
2
3  on: [push] # когда срабатывает (на push)
4
5  jobs: # какие задачи делаем
6    build: # сборка
7      runs-on: ubuntu-latest # на какой ОС запускаем
8
9      steps: # какие шаги выполняем
10     - uses: actions/checkout@v2 # выкачиваем репо
11     - name: Set up JDK 1.8 ← Меняем на 11
12       uses: actions/setup-java@v1 # устанавливаем JDK
13       with:
14         java-version: 1.8 # версия для установки
15     - name: Build with Maven
16       run: mvn -B package --file pom.xml # запускаем Maven
17
```

Use **Control** + **Space** to trigger autocomplete in most situations.

Start commit

maven.yml

Workflow — набор задач, в случае GitHub Actions — файл формата YML*, находящийся в каталоге `.github/workflows`.

В этом файле описана вся необходимая конфигурация.

При нажатии на кнопке `Start Commit` этот файл будет добавлен в ваш репозиторий и GitHub включит интеграцию с GitHub Actions (не забудьте сделать `git pull`):

Create new file

Upload files

Find file

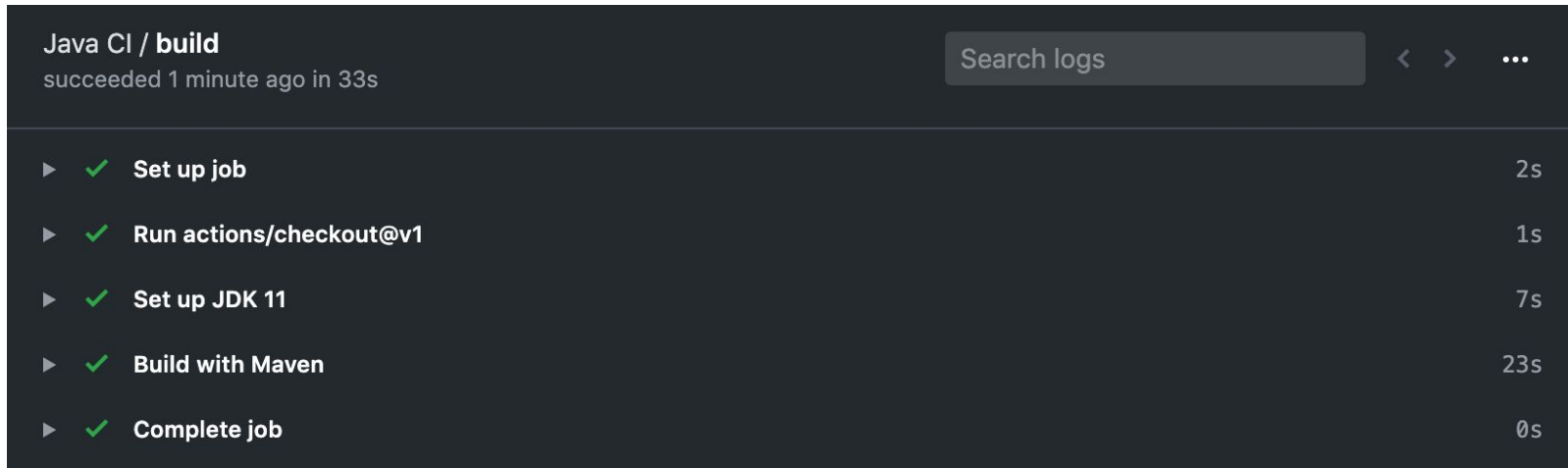
Clone or download ▼



Latest commit 4dce5ac 1 minute ago

Примечание*: самостоятельно ознакомьтесь с этим форматом (материала [Википедии](#) будет достаточно).

GITHUB ACTIONS

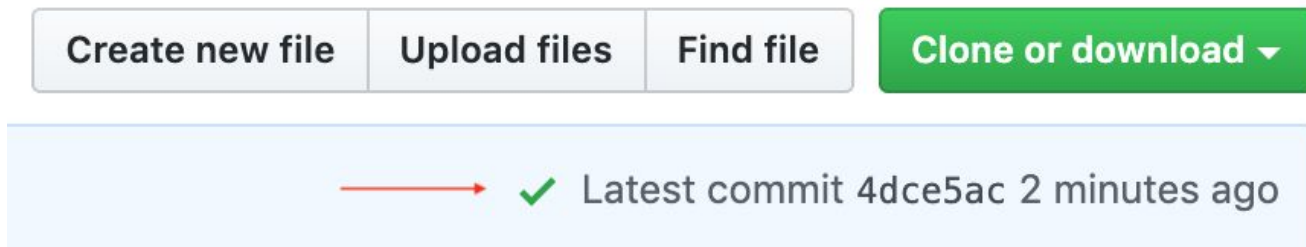


Java CI / **build**
succeeded 1 minute ago in 33s

Search logs

- ▶ ✓ Set up job 2s
- ▶ ✓ Run actions/checkout@v1 1s
- ▶ ✓ Set up JDK 11 7s
- ▶ ✓ Build with Maven 23s
- ▶ ✓ Complete job 0s

А на главной странице мы увидим статус последнего коммита:



Create new file Upload files Find file Clone or download ▾

→ ✓ Latest commit 4dce5ac 2 minutes ago



GITHUB ACTIONS

Демонстрация подключения GitHub Actions и просмотра результатов сборки.



Q & A

Q: зачем это всё нужно? Разве недостаточно того, что мы локально прогоняем тесты?

A: помимо вас в команде будут и другие участники, которые не всегда будут эти тесты прогонять. А CI будет это делать всегда.

Кроме того, мы стремимся избежать синдрома «А на моём компьютере работает» (works on my machine) — CI выкачивает из репозитория код и собирает (билдит) его на чистой машине.

Это обеспечивает нас уверенностью в том, что наш код будет собираться не только на машинах разработчиков/тестирующих.



EVERYTHING AS CODE

Обратите внимание: конфигурацию CI (как и конфигурацию Maven) мы храним в виде текстовых файлов в том же репозитории, что и сам проект.

Это позволяет нам вести их историю, а также **«держат всё в одном месте»**.

Сейчас это самый современный подход — мы храним в виде текстовых файлов в репозитории всё:

- сам код
- автотесты
- ресурсы (включая тестовые)
- настройки CI
- документацию
- и даже настройки публикации артефактов

ПРОМЕЖУТОЧНЫЕ ИТОГИ

Итак, мы настроили с вами простейшую (но уже очень полезную) конфигурацию CI, которая работает на пустом проекте.

На это мы потратили всего несколько кликов мыши.

Поскольку конфигурация хранится в обычном файле, мы можем не создавать её каждый раз через интерфейс GitHub, достаточно будет скопировать её в новый проект (как мы делаем с `.gitignore`).

Настало время разобраться с Maven и понять, что значит вот эта строка:

```
mvn -B package --file pom.xml.
```



MAVEN



MAVEN LIFECYCLES

Maven предлагает нам концепцию **LifeCycles** — жизненных циклов (набора последовательных активностей), которые нужны для управления проектом:

- `default` — сборка, тестирование и развёртывание;
- `clean` — очистка проекта и удаление всех сгенерированных артефактов;
- `site` — создание документации на проект.

Т.е. разработчики Maven выделили ключевые (с их точки зрения) типовые задачи при управлении проектом и "вшили" их в Maven.

Default Lifecycle ←

Phase	Description
<code>validate</code>	validate the project is correct and all necessary information is available.
<code>initialize</code>	initialize build state, e.g. set properties or create directories.
<code>generate-sources</code>	generate any source code for inclusion in compilation.
<code>process-sources</code>	process the source code, for example to filter any values.
<code>generate-resources</code>	generate resources for inclusion in the package.
<code>process-resources</code>	copy and process the resources into the destination directory, ready for packaging.
<code>compile</code>	compile the source code of the project.
<code>...</code>	...

Жизненные циклы разделены на фазы (Phases), а внутри фазы можно привязать цели плагинов (Goals):

Default Lifecycle Bindings - Packaging `ejb` / `ejb3` / `jar` / `par` / `rar` / `war`



Phase	plugin:goal
<code>process-resources</code>	<code>resources:resources</code>
<code>compile</code>	<code>compiler:compile</code>
<code>process-test-resources</code>	<code>resources:testResources</code>
<code>test-compile</code>	<code>compiler:testCompile</code>
<code>test</code>	<code>surefire:test</code>
<code>package</code>	<code>ejb:ejb</code> or <code>ejb3:ejb3</code> or <code>jar:jar</code> or <code>par:par</code> or <code>rar:rar</code> or <code>war:war</code>
<code>install</code>	<code>install:install</code>
<code>deploy</code>	<code>deploy:deploy</code>

Цель из себя представляет конкретную задачу, которую может выполнить конкретный плагин (представляйте это как метод в классе).



MAVEN LIFECYCLES

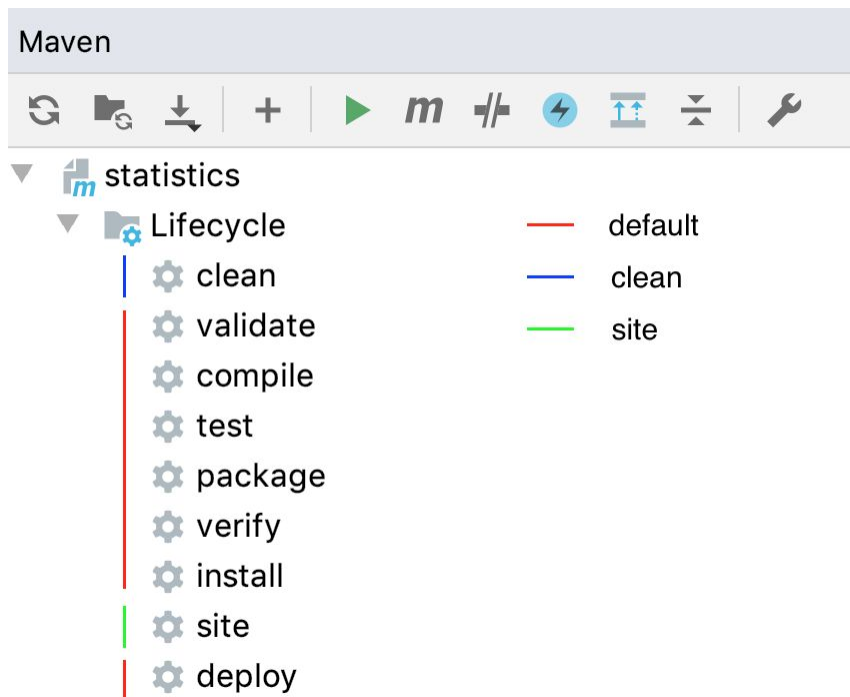
Важно: когда мы пишем `mvn test`, то запускается фаза `test`, а именно запускаются все цели плагинов, привязанные к этой фазе.

Но **фазы выполняются последовательно**: если мы запускаем фазу `test`, то срабатывают все фазы до неё (в рамках конкретного LifeCycle).

Именно поэтому когда мы запускали `mvn test` будет запускаться и обработка ресурсов и компиляция.

IDEA MAVEN LIFECYCLES

В панельке Maven IDEA ключевые фазы жизненных циклов выглядят вот так (вперемешку, но в рамках одного жизненного цикла упорядочены, т.е. test идёт после compile):














Q & A

Q: хорошо, но в рамках ДЗ мы же запускали вот так `spotbugs:check`, что это значит?

A: это возможность запускать отдельные цели плагинов (без привязки к Lifecycle).

Например, у плагина `Surefire` две цели (панель `Maven Plugins`):

- ▼  Plugins
 - ▶  clean (org.apache.maven.plugins:maven-clean-plugin:2.5)
 - ▶  compiler (org.apache.maven.plugins:maven-compiler-plugin:3.1)
 - ▶  deploy (org.apache.maven.plugins:maven-deploy-plugin:2.7)
 - ▶  install (org.apache.maven.plugins:maven-install-plugin:2.4)
 - ▶  jar (org.apache.maven.plugins:maven-jar-plugin:2.4)
 - ▶  resources (org.apache.maven.plugins:maven-resources-plugin:2.6)
 - ▶  site (org.apache.maven.plugins:maven-site-plugin:3.3)
 - ▼  **surefire (org.apache.maven.plugins:maven-surefire-plugin:2.22.2)**
 -  surefire:help
 -  surefire:test

Из одного из предыдущих слайдов видно, что цель `surefire:test` привязана к фазе `test`, а вот цель `surefire:help` — ни к какой фазе не привязана.

Q & A

Q: т.е. некоторые цели не привязаны к фазам, а внутри некоторых фаз может не быть целей?

A: совершенно верно.

Q: а как узнать, к какой фазе какие цели привязаны?

A: можно посмотреть Effective POM либо воспользоваться специальным плагином: `mvn help:describe -Dcmd=clean` (через `-D` передаются аргументы в формате `ключ=значение`).

```
[INFO] --- maven-help-plugin:3.2.0:describe (default-cli) @ statistics ---
[INFO] 'clean' is a phase within the 'clean' lifecycle, which has the following phases:
* pre-clean: Not defined
* clean: org.apache.maven.plugins:maven-clean-plugin:2.5:clean
* post-clean: Not defined
```

Q & A

Q: но в панельке IDEA же не было никакого плагина `help`?

A: не все плагины там перечислены, если вы указываете какой-то, которого там нет, Maven вполне в состоянии его скачать.

Общий формат запуска цели любого плагина выглядит вот так: `mvn groupId:artifactId:version:goal`.

Например, `mvn com.github.spotbugs:spotbugs-maven-plugin:3.1.12.2:help`, но:

- можно не указывать версию (тогда будет взята последняя релизная версия);
- можно указывать сокращённое имя (если плагин называется `name-maven-plugin` или `maven-name-plugin`);
- можно не указывать `groupId` (если он равен `org.apache.maven.plugins` или `org.codehaus.mojo`).



MAVEN PLUGINS

Но для большинства придётся указывать всё, кроме версии: `mvn com.github.spotbugs:spotbugs-maven-plugin:help`, если только вы их не укажете в `pom.xml`.

К этому мы и приступим (подключению полезных и популярных плагинов), но давайте сначала напишем немного кода.



ЗАДАЧА

ЗАДАЧА

К нам попал в руки следующий код, доставшийся в наследство от другого программиста (так называемое `legacy` — наследие):

```
package ru.netology.statistic;

public class StatisticsService {
    /**
     * Calculate index of max income
     *
     * @param incomes — array of incomes
     * @return — index of first max value
     */
    public long findMax(long[] incomes) {
        long current_max_index = 0;
        long current_max = incomes[0];
        for (long income : incomes)
            if (current_max < income)
                current_max = income;
        return current_max;
    }
}
```

Конечно же, никаких тестов, Maven'а и всего остального нет и в помине 🐈.



ЗАДАЧА

Поскольку код написан просто безобразно, ваши коллеги попросили вас помочь с внедрением системы CI, которая позволит автоматически запускать автотесты, проверять типичные ошибки и стиль кодирования.

Важно: не внедряйте ничего в командную работу в одностороннем порядке, потому что вы так решили. Обязательно согласуйте это с коллегами!

Ни в коем случае не вмешивайтесь в процессы и инструменты программистов, пока заранее с ними это не обговорите и не «продадите им идею».

ЗАДАЧА

Запускаем `mvn test` и удостоверимся, что сборка зелёная и тесты проходят (т. к. тестов у нас нет).

После чего заливаем всё на GitHub и смотрим, что говорит CI:

▼ ✓ Build with Maven

```
1  ► Run mvn -B package --file pom.xml
7  [INFO] Scanning for projects...
8  [INFO]
9  [INFO] -----< ru.netology:statistics >-----
10 [INFO] Building statistics 1.0-SNAPSHOT
11 [INFO] -----[ jar ]-----
```

Но CI запускает другую команду:

- `-B` — batch mode, неинтерактивный режим (цвета, прогресс);
- `package` — запускаемая фаза (ниже `test` поэтому тесты тоже будут запускаться);
- `--file pom.xml` — указание на используемый файл проекта.

JAR

Попробуем запустить эту команду локально (увидим в конце лога ниже):

```
Building jar: /ci/target/statistics-1.0-SNAPSHOT.jar
```

Jar (Java Archive) — это архив, в который упаковываются скомпилированные Java-приложения* и метаданные для дальнейшего распространения (развёртывания на сервере, подключения к другим проектам, публикации в репозиториях).

Именно таким же образом (через jar-архивы) распространяется JUnit Jupiter и другие библиотеки. Фаза `package` отвечает за создание этого архива.

Затем этот архив можно будет с помощью фазы `install` опубликовать в локальном репозитории (каталог `.m2` в вашем домашнем каталоге), либо в Maven Central или другом репозитории с помощью фазы `deploy`.



VERIFY

Соответственно, `package` запускается для того, чтобы удостовериться, что код не только проходит тесты, но и собирается в `jar`.

Но в большинстве случаев в рамках тестирования запускают фазу `verify`, в которой можно проводить доп.проверки после упаковки.

На данном этапе к этой фазе не прикреплено никаких целей (см. `mvn help:describe -Dverify`), но мы скоро это исправим.

VERIFY

Поэтому давайте переделаем команду сборки в `maven.yml` на следующую:
`mvn -B -e verify`.

После чего снова всё запустим на GitHub и удостоверимся, что сборка по-прежнему проходит.

Maven запускается в формате: `mvn [options] [<goal(s)>] [<phase(s)>]`, справку можно получить с помощью `mvn -h`



SUREFIRE

SUREFIRE

За автотесты у нас отвечает Surefire, поэтому в первую очередь ему можно задать вопрос о том, а были ли тесты вообще.

И если их не было, то можно просто «ронять» сборку.

Поискав на сайте Maven можно найти на описание [конфигурации Surefire](#):

<failIfNoTests>

Set this to "true" to cause a failure if there are no tests to run. Defaults to "false".

- **Type:** java.lang.Boolean
- **Since:** 2.4
- **Required:** No
- **User Property:** failIfNoTests

SUREFIRE

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
      <configuration>
        <failIfNoTests>true</failIfNoTests>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Удостоверяемся, что локально сборка падает (и на CI тоже — после пуша):

```
572 [INFO] -----
573 [INFO] BUILD FAILURE
574 [INFO] -----
575 [INFO] Total time: 14.085 s
576 [INFO] Finished at: 2020-02-27T17:47:05Z
577 [INFO] -----
578 [ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.22.2:test (default-
test) on project statistics: No tests to run! -> [Help 1]
```



Q & A

Q: но что если кто-то изменит конфигурацию сборки и уберёт эти настройки?

A: в небольших компаниях за это просто бьют по рукам, в более крупных — только у ключевых сотрудников есть доступ на push в master, поэтому они будут следить за этим (и не пропустят подобные pull-request'ы).

Примечание*: в GitHub можно настроить Protected Branches (бранчи, в которые нельзя "заливать" код кому попало).

ТЕСТЫ

Напишем небольшой тест, который проверяет работу сервиса (чтобы проверки Surefire проходили):

```
class StatisticsServiceTest {  
    @Test  
    void findMax() {  
        StatisticsService service = new StatisticsService();  
  
        long[] incomesInBillions = {12, 5, 8, 4, 5, 3, 8, 6, 11, 11, 12};  
        long expected = 12;  
  
        long actual = service.findMax(incomesInBillions);  
  
        assertEquals(expected, actual);  
    }  
}
```



ТЕСТЫ

Понятно, что с точки зрения комбинаторики наш тест не покрывает всех возможных сценариев, но можно ли как-то посмотреть, какой код вообще выполняется в результате прогона теста?

Не сидеть же с дебаггером и «прокликать» всё.



JACOCO

CODE COVERAGE

Code Coverage — метрика, показывающая, насколько наш код покрыт автотестами (т.е. % запущенного в результате прогона автотестов).

Для использования её (метрики) в нашем проекте, достаточно использовать [плагин](#), который использует инструмент [JaCoCo](#) (Java Code Coverage):

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.5</version>
</plugin>
```

JACOCO PLUGIN

Но при этом ничего не происходит: плагин не публикует самостоятельно свои цели в фазы. Нам необходимо это настроить самим.

`mvn jacoco:help` (поскольку `jacoco` у нас в `pom.xml` мы не обязаны указывать полное имя).

Плагин выведет справку по своему использованию:

- `jacoco:prepare-agent`;
- `jacoco:report`.

Минимальная конфигурация

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.5</version>
  <executions>
    <execution>
      <!-- id (придумываем сами) -->
      <id>prepare-agent</id>
      <phase>initialize</phase>
      <goals>
        <!-- какую цель выполняем (берём из справки) -->
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <!-- id (придумываем сами) -->
      <id>report</id>
      <goals>
        <!-- какую цель выполняем (берём из справки) -->
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```




Можно не писать фазы, т.к. они зашиты в целях (если нет необходимости их переназначить).

JACOCO

- `prepare-agent` подготавливает агента JVM для отслеживания вызовов кода
- `report` генерирует отчёт

После запуска `verify` отчёт будет в каталоге `target/site/jacoco/index.html`:

statistics

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 ru.netology.statistic		93 %		75 %	1	4	1	7	0	2	0	1
Total	2 of 33	93 %	1 of 4	75 %	1	4	1	7	0	2	0	1

StatisticsService.java

```
1. package ru.netology.statistic;  
2.  
3. public class StatisticsService {  
4.     /**  
5.      * Calculate index of max income  
6.      *  
7.      * @param incomes - array of incomes  
8.      * @return - index of first max value  
9.      */  
10.    public long findMax(long[] incomes) {  
11.        long current_max_index = 0;  
12.        long current_max = incomes[0];  
13.        for (long income : incomes)  
14.            if (current_max < income)  
15.                current_max = income;  
16.        return current_max;  
17.    }  
18. }
```

1 of 2 branches missed.

Created with JaCoCo 0.8.5.201910111838

- зелёный фон — выполнено при прохождении тестов
- жёлтый фон — выполнено не до конца (одна из веток не отработала)
- красный фон — не выполнено

Таким образом, мы видим, что наши тесты не покрывают нескольких участков кода (это необходимо исправить).



СИНДРОМ 100%

Часто в компаниях, активно использующих автотесты можно услышать «Наш код на 100% покрыт автотестами» в формулировке «**Протестировано ВСЁ**».

Вопрос к аудитории: как вы думаете, возможно ли протестировать всё?



СИНДРОМ 100%

Исчерпывающее тестирование, как вы знаете, невозможно.

Поэтому нужно, чтобы вы всегда помнили следующее: **Code Coverage** вам покажет **только участки кода, которые не исполнялись в результате прогона тестов.**

Code Coverage **не говорит о том, что своими тестами вы покрыли все возможные сценарии.**

Покрытие в 100% показывает только то, что все участки кода в результате прогона тестовы были исполнены.



CODE COVERAGE

Вы должны по-прежнему использовать комбинаторику, тест-анализ и тест-дизайн для того, чтобы покрывать **сценарии использования**, а не строки кода.



QUALITY GATE

Но при этом покрытие кода автотестами достаточно часто используют в качестве Quality Gate* для новых возможностей.

Можно установить конкретную цифру для нового кода, а можно требовать, чтобы добавление нового кода/изменение существующего не вело к падению Code Coverage.

Примечание*: Quality Gate — формализованные требования к качеству.



ДРУГИЕ ПЛАГИНЫ



ДРУГИЕ ПЛАГИНЫ

В своей домашней работе вы познакомитесь ещё с двумя плагинами:

1. [SpotBugs](#) — статический анализатор, который ищет типовые ошибки в Java-коде.
2. [CheckStyle](#) — анализатор, проверяющий соответствие установленному в организации стилю написания кода.

Обязательно делайте домашние задания, поскольку именно в них нарабатываются практические навыки.



ИТОГИ



ИТОГИ

Сегодня была одна из ключевых лекций для нас, как автоматизаторов:

- мы узнали о системах CI/CD;
- мы начали серьёзно работать с Maven'ом;
- мы поговорили о Code Coverage и заблуждениях, связанных с ним.

Обязательно практикуйтесь в работе с Maven — он будет одним из ваших ключевых инструментов работы.



ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты **все задачи**.



Задавайте вопросы и напишите отзыв о лекции!

ВАСИЛИЙ ДОРОХИН

