ASTON.

# Инструменты сборки проектов

ISO 45001 ISO 50001 ISO 9001 ISO 14001 IATF 16949:2016 CERTIFIED COMPANY

astondevs.ru

# Понятие и назначение

В жизни каждого программиста возникает момент, когда маленьких и простых приложений недостаточно для решения поставленных задач. Иногда даже стандартного инструментария Java недостаточно. Тогда используются чужие библиотеки и фреймворки.
Раньше разработчики просто делились файлами своих библиотек друг с другом и подключали их к своему проекту. Проще говоря, клали файлы в соответствующую директорию.
У такого подхода есть три проблемы:

- Сложно следить за версиями библиотек. Если твой друг выпустил версию 2.0, а тебе не сказал, это полбеды. А вот если ты уже скачал новую версию → удалил старый файл библиотеки → оказалось, что новый файл требует ещё каких-то других библиотек.
- Проблема коллаборации. Если над проектом работали 50 программистов, и один стал использовать функции из новой библиотеки, а его коллеги из другого модуля ещё нет, возникали конфликты — сначала в коде, а затем — в отношениях.
- Сложно управлять кодом чужих библиотек. Нельзя выкачать только часть библиотеки или настроить этапы, на которых эта зависимость используется, запрещается автоматизировать рутинные задачи при сборке проекта.

# Понятие и назначение

Сборщики проектов — это фреймворки, которые автоматизируют сборку. Их можно подключить практически в любом проекте. Даже загрузка дополнительных jar-библиотек занимает время, а со сборщиком все требуемые библиотеки подтягиваются из удалённого репозитория.

# Виды сборщиков проектов

Первой утилитой сборки можно назвать make. Эта утилита собирала проекты на языках C и C++. Она до сих пор популярна, хоть и не приносит Java-разработчику большой пользы. Но именно с этой утилиты началось становление платформонезависимых сборщиков.

В отличие от make, утилита Ant полностью независима от платформы. Требуется только установка на применяемой системе рабочей среды Java — JRE. Отказ от использования команд операционной системы и формат XML обеспечивают переносимость сценариев.

Управление процессом сборки происходит посредством XML-сценария или Build-файла. Он содержит определение проекта, состоящего из отдельных целей (Targets). Эти цели сравнимы с процедурами в языках программирования и содержат вызовы команд-заданий (Tasks). Каждое задание — это неделимая, атомарная команда, выполняющая элементарное действие.

Во фреймворке Maven управление процессом сборки происходит также средствами XML-файлов — POM-файлов, от Project Object Model.

# Виды сборщиков проектов

Между целями определяются зависимости — каждая выполнится только после тех, от которых она зависит. Если эти зависимости выполнялись ранее, повторно они не производятся.

Типичные примеры целей:

- clean — удаление промежуточных файлов;
- compile — компиляция всех классов;
- deploy — развёртывание приложения на сервере.

Конкретный набор целей и их взаимосвязи зависят от специфики проекта.Сейчас устаревающий Apache Ant практически не используется в проектах. На его место пришли два гиганта: **Maven и Gradle**.

# Понятие обобщения

Подводя итог, можно выделить три проблемы при использовании такого подхода:

- Каждый раз, когда мы хотим вытащить данные из нашей универсальной коробки, нам необходимо выполнять приведение типов;
- Чтобы не получить ClassCastException, перед каждым приведением типов, необходимо делать проверку типов данных с помощью instanceof;
- Если мы где-то будем применять приведение типов, и забудем прописать instanceof, то появится вероятность появления ClassCastException в этой части кода.

# Maven

Если установлена Intellij IDEA, то ничего дополнительно устанавливать не нужно. Maven поставляется вместе с нашей IDE.

Если же выбор пал на другую IDE, возможно, потребуется установить плагин Maven. Установка Maven не занимает много времени. Достаточно скачать с сайта Maven — Download Apache Maven архив с последней стабильной версией. Затем распаковать его и указать системные переменные среды M2_HOME в пути ~\maven\ в зависимости от того, куда его распаковали.

При создании нового проекта (File → New → Project) нужно выбрать Maven и версию SDK. Появится опция Create From Archetype. Разумеется, создать проект можно и без использования архетипов, но остановимся на этой теме подробнее.

# Архетип

Самый простой и удобный способ создания нового проекта в Apache Maven. Это создание его из архетипа.

**Архетип** — это шаблон будущего проекта. Всего имеется порядка 1800 известных архетипов.

Очевидно, что архетипы не даны нам свыше и были кем-то написаны. А следовательно, можно написать свой архетип. В Maven даже есть готовый архетип для написания архетипов: maven-archetype-archetype.

# Создание собственного архетипа

В каталоге src/main/resources/archetype-resources располагается содержимое будущего архетипа, включая его pom.xml. В отличие от «настоящего» pom.xml, во внутреннем координаты проекта заданы переменными:

```
<groupId>${groupId}</groupId>
<artifactId>${artifactId}</artifactId>
<version>${version}</version>
```

В файле src/main/resources/META-INF/maven/archetype.xml находится дескриптор архетипа.

```
<archetype>
  <id>archetype</id>
  <sources>
    <source>src/main/java/App.java</source>
  </sources>
  <testSources>
    <source>src/test/java/AppTest.java</source>
  </testSources>
</archetype>
```

# Создание собственного архетипа

Используя теги <sources/>, <resources/>, <testSources/>, <testResources/>, <siteResources/>, можно задать расположение соответствующих исходников артефакта. «Внешний» pom.xml задаёт координаты и прочие параметры архетипа.

Созданный таким образом архетип собирается, загружается в репозиторий и использоваться в дальнейшем.
Все архетипы знать не обязательно, обратим внимание на следующие:

- **maven-archetype-plugin.** Шаблон плагина к Apache Maven. Включает в себя образец плагина.
- **maven-archetype-quickstart.** Один из наиболее удобных и часто используемых архетипов. Создаёт java-приложение, состоящее из классического Hello world, образца теста и зависимости от JUnit.
- **maven-archetype-site.** Apache Maven поддерживает генерацию сайта проекта, включая туда статистику по исходникам, собранные артефакты и так далее. Такая функциональность в «дикой природе» используется редко, но архетип есть.
- **gmaven-archetype-basic.** Шаблон проекта, поддерживающий интеграцию Maven с Groovy.

# Создание проекта с архетипом

Создадим проект с архетипом maven-archetype-quickstart и внимательно его рассмотрим. После выбора архетипа укажем ещё название проекта, например, JavaUIAutomation, и, опционально, «реквизиты» проекта (Artifact coordinates): GroupId, ArtifactId, Version. О них мы поговорим ниже.

После нажатия кнопки Finish использованный для создания проекта архетип самостоятельно сгенерировал каталоги для Java package, применяя координаты проекта для его названия.

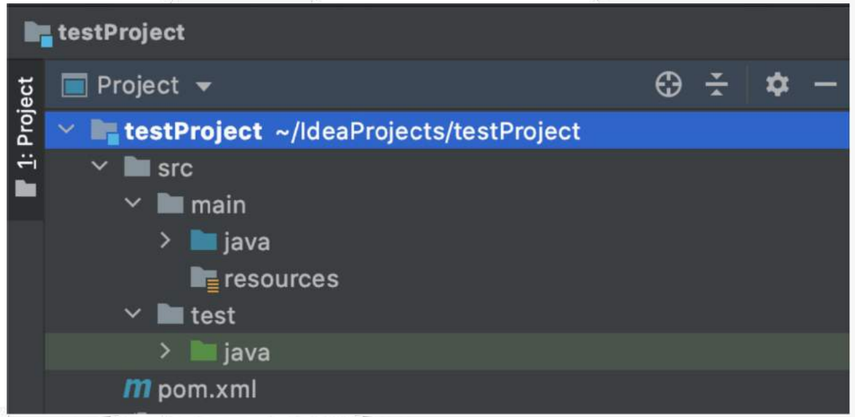
Конечно, проект можно создать и без пользовательского интерфейса IDE, с использованием командной строки. Для этого наберём в терминале:

**$ mvn archetype:generate -DgroupId=ru.geekbrains.lesson -DartifactId=hellomaven -Dversion=1 -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false.**

- **DarchetypeArtifactId=maven-archetype-quickstart** — выбирает архетип maven-archetype-quickstart.
- **DinteractiveMode=false** — говорит Maven, что мы знаем что делаем и что мы не хотим вести диалог.
- **DgroupId=ru.geekbrains.lesson** — группа. Обычно группа связана с названием компании, разработчика или большого продукта. Например, сам Maven и связанные с ним проекты используют группу org.apache.maven.
- **DartifactId=hellomaven** — имя артефакта. То, как мы называем проект.
- **Dversion=1** — версия артефакта.

# Структура проекта

Maven использует подход, известный как соглашение по конфигурации (convention over configuration), и ожидает, что мы расположим файлы исходного кода определённым образом:




В частности, maven ожидает, что все исходники расположатся в каталоге src, причём исходный код программы — в подкаталоге main/java, а исходный код тестов — в каталоге test/java. Есть ещё несколько стандартных каталогов, но они обычно специфичны для той или иной платформы, например, для Android.

# Описание примерной базовой иерархии файлов в проекте

Подробное описание всех каталогов:

- корневой каталог проекта: файл pom.xml и все дальнейшие подкаталоги;
- src: все исходные файлы;
- src/main: исходные файлы для продукта;
- src/main/java: Java — исходный код;
- src/main/resources: другие файлы, которые используются при компиляции или исполнении (например, Properties-файлы);
- src/test: исходные файлы, необходимые для организации автоматического тестирования;
- src/test/java: JUnit-тест для автоматического тестирования;
- target: все создаваемые в процессе работы Maven файлы, на скриншоте этой папки нет, почему — смотри ниже;
- target/classes: скомпилированные Java-классы.

# Описание примерной базовой иерархии файлов в проекте

В IDEA всё помечается конкретным цветом:

- голубым — исходники;
- зелёным — тесты;
- папки с ресурсами имеют дополнительный значок.

Если автоматически так не случилось, можно сделать это самостоятельно через контекстное меню папки → Mark Directory As... → выбрать тип папки. Ещё это можно сделать через Project Structure → Modules.

Заглянем в файл POM.xml, который расположился в корне нашего проекта.

# Файл POM

Пример POM.xml:

Этот файл — ключевой при автоматической сборке проекта. По сути, это такой же набор инструкций, как и Ant. Maven ушёл достаточно далеко от Ant, поэтому в проекте используем именно его.

Всё приложение — это один большой артефакт сборки. Поэтому всё, что касается приложения, нужно описывать между тегами <project/>.

```
<project>
  <!-- версия модели для POM-ов Maven 2.x всегда 4.0.0
-->
  <modelVersion>4.0.0</modelVersion>
  <!-- координаты проекта, то есть набор значений,
который позволяет однозначно идентифицировать этот
проект -->
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <!-- зависимости от библиотек -->
  <dependencies>
    <dependency>
      <!-- реквизиты необходимой библиотеки -->
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <!-- эта библиотека используется только для
запуска и компилирования тестов -->
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

# Файл POM

Все реквизиты требуется объявить в специальных тегах:

- **<modelVersion>** — версия Maven.
- **<groupId>** — что-то вроде идентификатора производителя. На самом деле принято использовать перевёрнутый адрес сайта компании, которая разработала библиотеку. Идентификатор разработчика нужен из-за нескольких библиотек с одинаковым названием.
- **<artifactId>** — название нашей библиотеки.
- **<version>** — версия проекта. Нужная вещь, если пишем вспомогательную библиотеку и собираемся выложить её в популярный репозиторий для других программистов. В репозитории файл pom.xml станет парситься, и версия нашей библиотеки обновится в соответствии с разработкой новых версий. Также помогает вспомнить текущую версию.
- **<packaging>** — вид, в который соберётся проект. Это может быть jar, war и другие. В этом случае устанавливается флаг pom для сборки maven-проекта. Последний используется как дополнительный модуль в других приложениях.
- Тег **<dependencies>** управляет зависимостями, которые будут включены в проект. Отдельная зависимость указывается своим тегом <dependency>, и в ней прописываются реквизиты библиотеки.

# Файл POM

**NB: Важно!** Способ указания используемой библиотеки полностью совпадает с методом описания реквизитов к приложению — те же теги, кроме тега <scope>.

В центральном репозитории любой пользователь найдёт библиотеку нужной версии и подсмотрит groupId и artifactId.
Важно учесть версию подтягиваемой библиотеки. Не всегда получается определить стабильную версию. Если мы не знаем библиотеку, то тег <version> обозначаем как SNAPSHOT. Этот флаг указывает maven, что нужно взять последнюю стабильную версию библиотеки.

Когда библиотек в проекте становится слишком много, например, список dependencies занимает больше 2 экранов на мониторе, удобно вынести версии библиотек в properties. Это нужно, чтобы все они находились в одном месте.

# Файл POM

Пример:

```
<properties>
  <junit.version>4.11</junit.version>
</properties>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Тег <scope> нужен, чтобы обозначить область видимости использования библиотеки. В этом случае прописываем между ними параметр test — Maven узнает, что библиотека JUnit подтягивается в наш проект, и скомпилируется только на этапе тестирования.

# Файл POM

Значения тега **<scope>**

- **compile** — устанавливается для всех по умолчанию. Означает, что зависимость подтягивается из репозитория для компиляции.
- **provided** — по функциональности близка с compile, но указывает JDK, что зависимость используется ещё и во время выполнения.
- **runtime** — зависимость используется только на этапе выполнения.
- **system** — зависимость, помеченная этим флагом, находится в папке с проектом, и никогда не ищется в репозиториях. Для этого также указывается путь тегом.

Бывают и другие флаги для тега <scope>, но на практике редко используются. Эти флаги есть в официальной документации.
Бывает, что одна зависимость содержит составные, но они нам не нужны. Чтобы не отягощать свой проект, эти зависимости не загружаются. Для этого между тегами <dependency> нужно расположить тег <exclusions>, а в нём — <exclusion>. Используя реквизиты, укажем, какие зависимости не надо подтягивать в проект.

# Файл POM

Как и версию, выбирается вид, в котором зависимость подгрузится в проект. Для веб-приложения тип jar не подойдёт, а зависимость типа war понадобится. В теге <dependency> указывается тег <type>, между которыми и устанавливается тип зависимости.

Самые известные — jar и war. Но если зависимость имеет специфический тип, например, sword, использовать его не запрещается. Главное, чтобы в своём проекте мы могли с ней работать.

# Жизненный цикл сборки

Жизненный цикл сборки продукта в Maven — это строго определённая последовательность фаз. Во время их выполнения должны достигаются конкретные цели. В каждой фазе указываются задачи, в некоторых переводах они называются «цели».

Задача (goal) — это специальная команда, относящаяся к сборке проекта и его управлению. Она привязывается как к нескольким фазам, так и ни к одной. Не привязанная ни к одной фазе задача запускается вне фаз сборки с прямым вызовом.

Порядок выполнения зависит от порядка вызова целей и фаз. Например, рассмотрим команду ниже. Аргументы clean и package — фазы сборки до тех пор, пока dependency:copy-dependencies считается задачей.

```
mvn clean dependency:copy-dependencies package
```

# Жизненный цикл сборки

В этом случае сначала выполнится фаза clean, после этого — задача dependency: copy-dependencies. Затем выполнится фаза package.

**Фазы жизненного цикла сборки:**

- **compile:** компилирование проекта.
- **test:** тестирование с помощью JUnit-тестов.
- **package:** создание .jar файла или war, ear в зависимости от типа проекта.
- **integration-test:** запуск интеграционных тестов.
- **install:** копирование .jar (war, ear) в локальный репозиторий.
- **deploy:** публикация файла в удалённый репозиторий.

# Жизненный цикл сборки

К примеру, нам нужно создать jar-файл проекта. Чтобы его создать, набираем:

```
mvn package
```

Но перед созданием jar-файла выполнятся все предыдущие фазы compile и test, а фазы integration-test, install, deploy не выполнятся. Если набрать:

```
mvn deploy
```

то выполняются все приведенные выше фазы.

Особняком стоят фазы **clean** и **site**. Они не выполняются, если специально не указаны в строке запуска.

- **clean:** удаление всех созданных в процессе сборки артефактов: .class, .jar и других файлов. В простейшем случае результат — просто удаление каталога target.
- **site:** предназначена для создания документации — javadoc+сайт описания проекта.

Так как команда **mvn** понимает, когда ему передают несколько фаз, то для сборки проекта создания документации «с нуля» выполняют:

```
mvn clean package site
```

# Плагины

Если говорить в целом, то Maven — это фреймворк, выполняющий плагины. В этом фреймворке каждая задача, по сути, выполняется плагинами.

Плагины Maven используются:

- для создания jar-файла;
- создания war-файла;
- компиляции кода файлов;
- юнит-тестирования кода;
- создания отчётов проекта;
- создания документации проекта.

В общей форме плагин обеспечивает набор целей. Эти цели выполняются с применением такого синтаксиса:
mvn [имя-плагина]:[имя-цели]

Например, чтобы выполнить компиляцию проекта, нужно использовать следующую команду:
mvn compiler:compile

# Типы плагинов в Maven

- Плагины сборки. Выполняются в процессе сборки и конфигурируются внутри блока <build></build> файла pom.xml.
- Плагины отчётов. Выполняются в процессе генерирования сайта и конфигурируются внутри блока <reporting></reporting> файла pom.xml.

Список наиболее используемых плагинов:

- **clean:** очищает цель после сборки. Удаляет директорию target.
- **compiler:** компилирует исходные Java-файлы.
- **surefire:** запускает тесты JUnit. Создаёт отчёты о тестировании.
- **jar:** собирает JAR-файл текущего проекта.
- **war:** собирает WAR-файл текущего проекта.
- **javadoc:** генерирует Javadoc проекта.

# Типы плагинов в Maven

Например:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.1</version>
</plugin>
```

Плагины указываются в файле pom.xml внутри блока <plugins></plugins>.
Каждый плагин может иметь несколько целей.
Мы можем определять фазу и из неё начать автоматическое выполнение плагина. В примере ниже используем фазу package.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>2.6</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

# Типы плагинов в Maven

Для работы большинства плагинов обычно требуются дополнительные настройки. Они специфичны для конкретного плагина. Настройки задаются в тегах <configuration>. Например, так настраивается tomcat-плагин:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>tomcat-maven-plugin</artifactId>
    <version>1.1</version>
    <configuration>
        <fork>false</fork>
        <server>test-server</server>
        <url>http://test-server/manager</url>
    </configuration>
</plugin>
```

# Репозитории

В мире Maven мы оперируем репозиториями. Под репозиторием понимается директория, где хранятся все JAR, библиотеки, плагины и любые артефакты, которыми пользуется Maven.

Типы репозиториев Maven:

- локальные (local);
- центральные (central);
- удалённые (remote).

# Локальные

**Локальный репозиторий** — это директория, которая хранится на нашем компьютере. Она создаётся в момент первого выполнения любой команды Maven.

Локальный репозиторий Maven хранит все зависимости проекта: библиотеки, плагины и т. д. Когда мы выполняем сборку проекта с применением Maven, то все зависимости (JAR-файлы) автоматически загружаются в локальный репозиторий. Так мы избегаем использование ссылок на удалённый репозиторий при каждой сборке проекта.

После выполнения команды run Maven все зависимости автоматически загрузятся в локальный репозиторий.
По умолчанию, локальный репозиторий создаётся Maven в директории %USER_HOME%. Чтобы изменить директорию, нужно указать её в файле settings.xml. Этот файл находится в папке %M2_HOME%\conf.

# Локальные

```
<?xml version="1.0" encoding="UTF-8"?>
<settings
xmlns="http://maven.apache.org/SETTINGS/1.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instan
ce"

xsi:schemaLocation="http://maven.apache.org/SETTIN
GS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">

<localRepository>/home/Documents/MyLocalRepository
</localRepository>
</settings>
```

Чтобы создать файл **Settings.xml**, нужно нажать правой кнопкой на корневую папку проекта → Maven → Create settings.xml. Однажды создав и сконфигурировав его, он всегда будет применяться по умолчанию в настройках Maven.

# Центральный репозиторий

Центральный репозиторий **Maven** — это репозиторий, который обеспечивается сообществом Maven. Он содержит большое число часто используемых библиотек.

Если **Maven** не может найти зависимости в локальном репозитории, то автоматически начинается поиск необходимых файлов в центральном репозитории.

# Удаленный репозиторий

Иногда Maven не может найти нужные зависимости в центральном репозитории. В этом случае процесс сборки прерывается, и в консоль выводится сообщение об ошибке.

Чтобы предотвратить подобную ситуацию, в Maven предусмотрен механизм Удалённого (remote) репозитория. Это репозиторий, который определяется самим разработчиком. Там часто хранятся все нужные зависимости.

Для настройки удалённого репозитория требуется внести следующие изменения в файл 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>
  <groupId>com.companyname.projectgroup</groupId>
  <artifactId>Tutorials</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>net.your-name.private-lib</groupId>
      <artifactId>private-lib</artifactId>
      <version>1.0.0</version>
    </dependency>
  <dependencies>
  <repositories>
    <repository>
      <id>your-name.lib1</id>
      <url>http://download.your-site.net/maven2/lib1</url>
    </repository>
    <repository>
      <id>your-name.lib2</id>
      <url>http://download.your-site.net/maven2/lib2</url>
    </repository>
  </repositories>
</project>
```

# Использование метасимвольных аргументов

Если для скачивания важных пакетов из удалённого репозитория требуется аутентификация, нужно указать реквизиты с информацией о логине или пароле и id репозитория в файле **settings.xml**.

```
<settings>
...
    <servers>
        <server>
            <id>your-name.lib2</id>
            <username>myMavenRepo</username>

<password>${yourHttpAuthReadPassword}</password>
        </server>
    </servers>
...
</settings>
```

# Порядок поиска зависимостей Maven

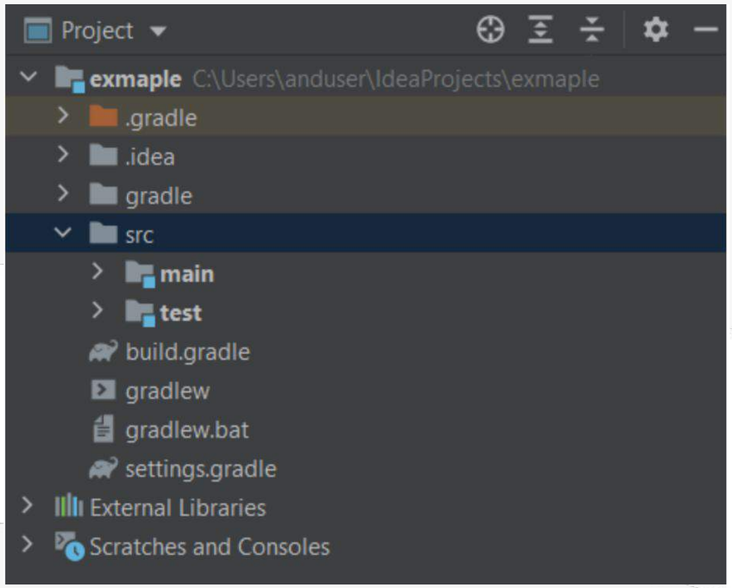
После выполнения сборки проекта в **Maven** автоматически начинается поиск зависимостей в следующем порядке:

- Поиск зависимостей в локальном репозитории. Если зависимости не обнаружены, происходит переход к шагу 2.
- Поиск зависимостей в центральном репозитории. Если они не обнаружены и удалённый репозиторий определён, то происходит переход к шагу 4.
- Если удалённый репозиторий не определён, то процесс сборки прекращается и выводится сообщение об ошибке.
- Поиск зависимостей на удалённом репозитории. Если они найдены, происходит их загрузка в локальный репозиторий, наоборот — выводится сообщение об ошибке.

# Gradle

**Gradle** - система автоматической сборки проектов, менеджер зависимостей, аналог Maven. Gradle основан на структуре (графе) т.н. тасок (task), которые могут зависеть друг от друга. Эти таски выполняют какую-то работу (например, сборку проекта).

Установить gradle можно по-старинке, скачав с официального сайта https://gradle.org/, либо создав проект в IDEA, указав его, как сборщик (это приведет к скачиванию самого Gradle и формированию дефолтной структуры проекта необходимыми gradle-файлами).




Основной файл для конфигурации зависимостей и тасок - build.gradle. Этот файл (как и другие файлы конфигураторы для Gradle) по дефолту понимает groovy (при желании/необходимости можно писать на kotlin-е, сделав для этого расширение у файла .kts)

# Дефолтное наполнение файла build.gradle

```
plugins {

    id 'java'

}

group 'org.example'
version '1.0-SNAPSHOT'
repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine'
}

test {
    useJUnitPlatform()
}
```

# Порядок поиска зависимостей Maven

**Плагин** — это набор задач, который мы “прикручиваем” к своему проекту. Почти все необходимые таски (компиляция задач, настройка объектов домена, настройка исходных файлов) выполняются плагинами. Применение плагина к проекту означает, что он позволяет расширять возможности проекта.

Примеры подключения плагина к проекту:

Groovy:

```
apply plugin: JavaPlugin
 plugins {
    id 'java'

}
plugins {
    id "com.jfrog.bintray" version "0.4.1"
}
```

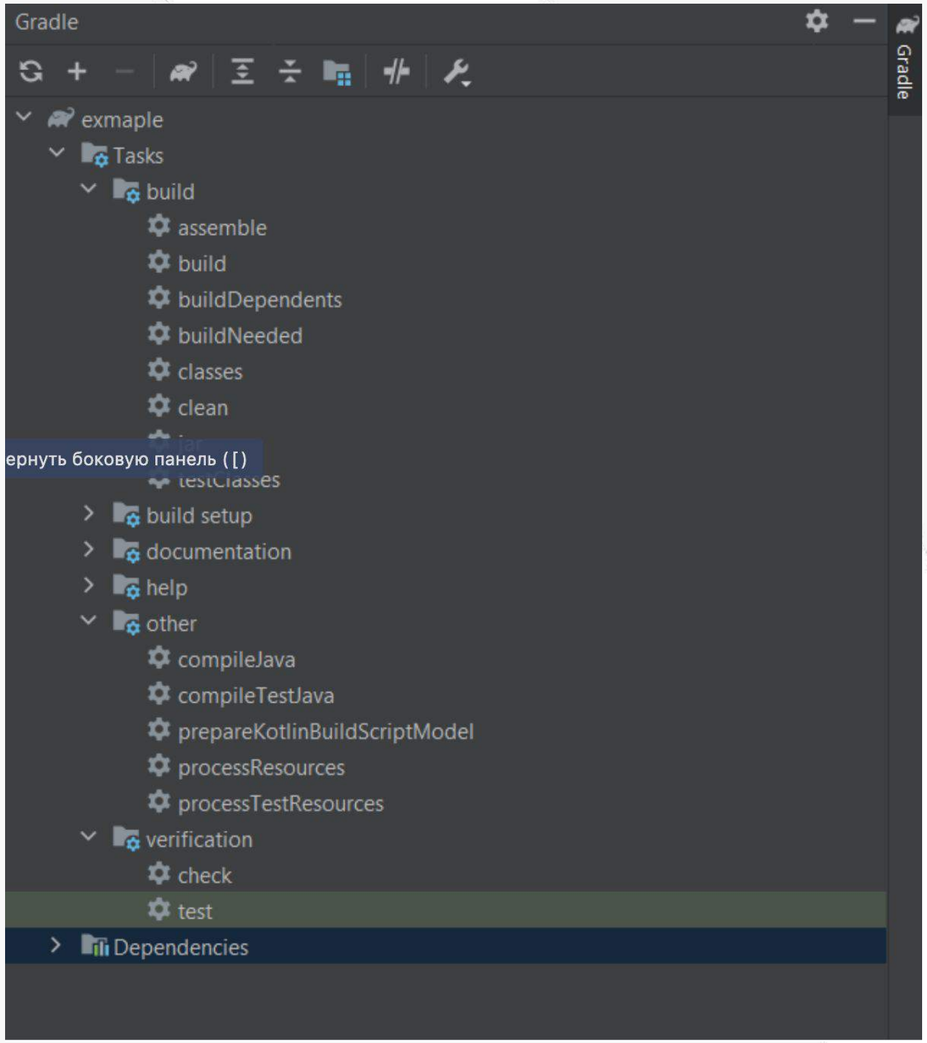
Kotlin:

```
plugins {
    id("io.qameta.allure") version "2.8.1"
    kotlin("jvm") version DefaultPlugin.Kotlin
    id("org.jlleitschuh.gradle.ktlint") version "9.4.1"
}
apply("${project.projectDir}/build-wsdl.gradle")
```

При наличии плагинов, подключенных к проекту, появляются первые таски.

# Порядок поиска зависимостей Maven

Например, указанные таски появляются при подключении плагина Java и нужны для компиляции/сборки проекта, использующего jvm.



# Раздел repositories

Отвечает за место, откуда будут подтягиваться плагины/библиотеки. **Gradle** не имеет своего центрального репозитория, по умолчанию используется дефолтный репозиторий maven-а (естественно, можно указать какие-то кастомные url-ы).

Примеры наполнения repositories:

```
repositories {
   mavenCentral()
}
repositories {

maven("http://nexus-new.domen.ru/repository/mvn-pro
xy/")
}
```

# Раздел dependencies

Раздел/блок, отвечающий за подключение сторонних библиотек к проекту (зависимостей).

Пример наполнения блока зависимостей

Groovy:

```
dependencies {
    testImplementation
'org.junit.jupiter:junit-jupiter-api:5.6.0 '
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine '
}
```

Kotlin:

```
dependencies {
    implementation("org.slf4j:slf4j-simple:1.7.30")
    runtimeOnly("org.postgresql:postgresql")
    runtimeOnly("com.oracle:ojdbc8:18.3.0.0")

testImplementation("org.junit.jupiter:junit-jupiter:5.7.0")

implementation("io.github.microutils:kotlin-logging:2.0.3")
}
```

Зависимости подключаются для разных этапов жизни проекта. Например, runtimeOnly зависимость используется только в runtime, testImplementation только для тестов и т.д.

# Раздел тасков

Задача (task) является основным компонентом процесса сборки в файле билда Gradle. Задачи представляют собой именованные наборы инструкций билда, которые Gradle запускает выполняя сборку приложения.
Пример изменения существующего таска (из какого-либо плагин):

```
tasks {
    test {
        maxParallelForks =
Runtime.getRuntime().availableProcessors()
        doFirst {
            logger.quiet("Max availableProcessors is
$maxParallelForks")
            useJUnitPlatform {
                excludeTags = setOf("integration")
            }
        }
        reports {
            junitXml.isEnabled = true
            html.isEnabled = false
        }
    }
}
```

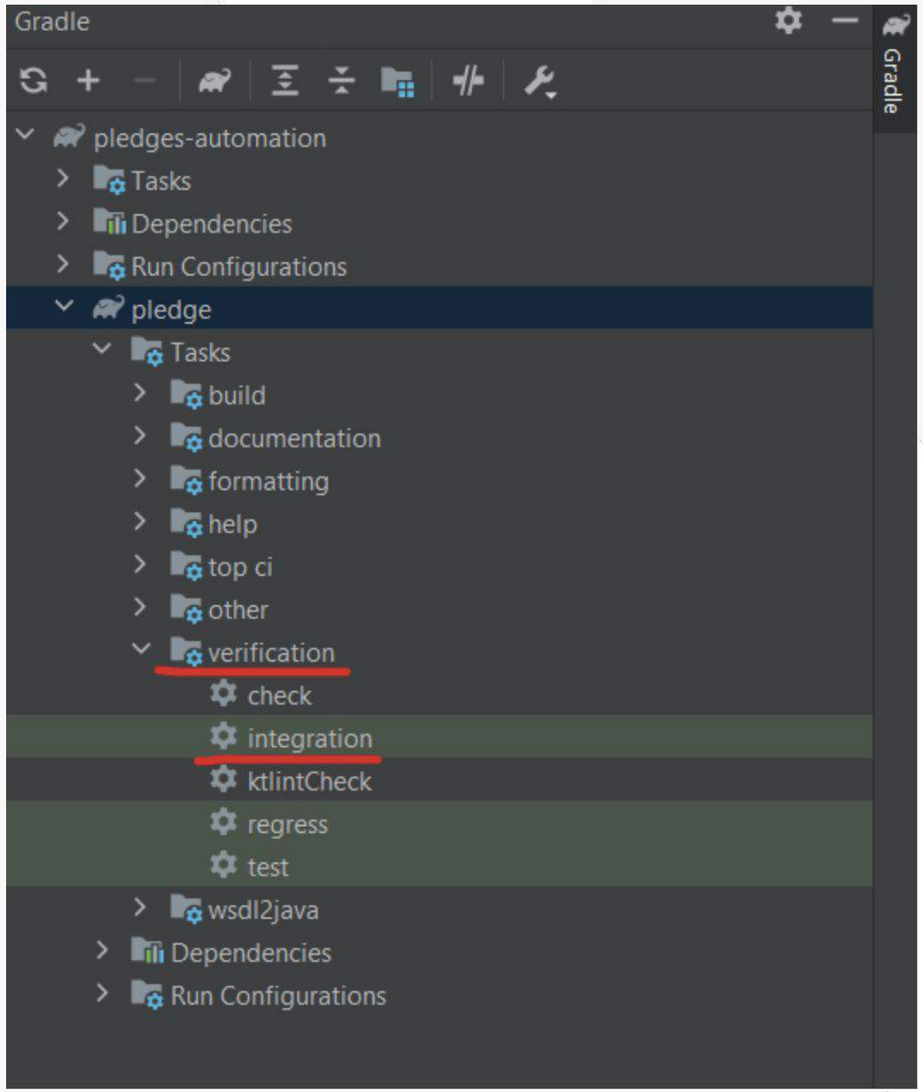
# Раздел тасков

Создание тасков

```
tasks.register("integration", Test::class.java) {
    group = "verification"
    useJUnitPlatform {
        includeTags = setOf("integration")
    }
    logger.info("Integration tests running")
}
```

Для вышеприведенного кода создаем таск с именем integration, который имплементирует класс Test (т.е. работает с тестовыми методами), относится к группе verification, для запуска тестовых методов использует платформу JUnit и запускает только те методы, у которых есть тег integration.

Достаточно подробно о Gradle можно также почитать здесь.



# Практическое задание

Создайте две ветки: **Lesson_11_maven** и **Lesson_11_gradle**.

В ветке **Lesson_11_maven** создайте проект, используя Maven с архетипом quickstart.
В ветке **Lesson_11_gradle** создайте проект, используя Gradle.

В каждый проект добавьте зависимости на такие инструменты как **JUnit 5, Selenium**.