# Лекция 5
# Разработка АС реального времени (часть 2)

ФИО преподавателя: Зорина Наталья Валентиновна

e-mail:zorina_n@mail.ru

# Тема лекции:

## «Система сборки Gradle. Конфигурация проекта. Управление зависимостями. Сложные проекты»

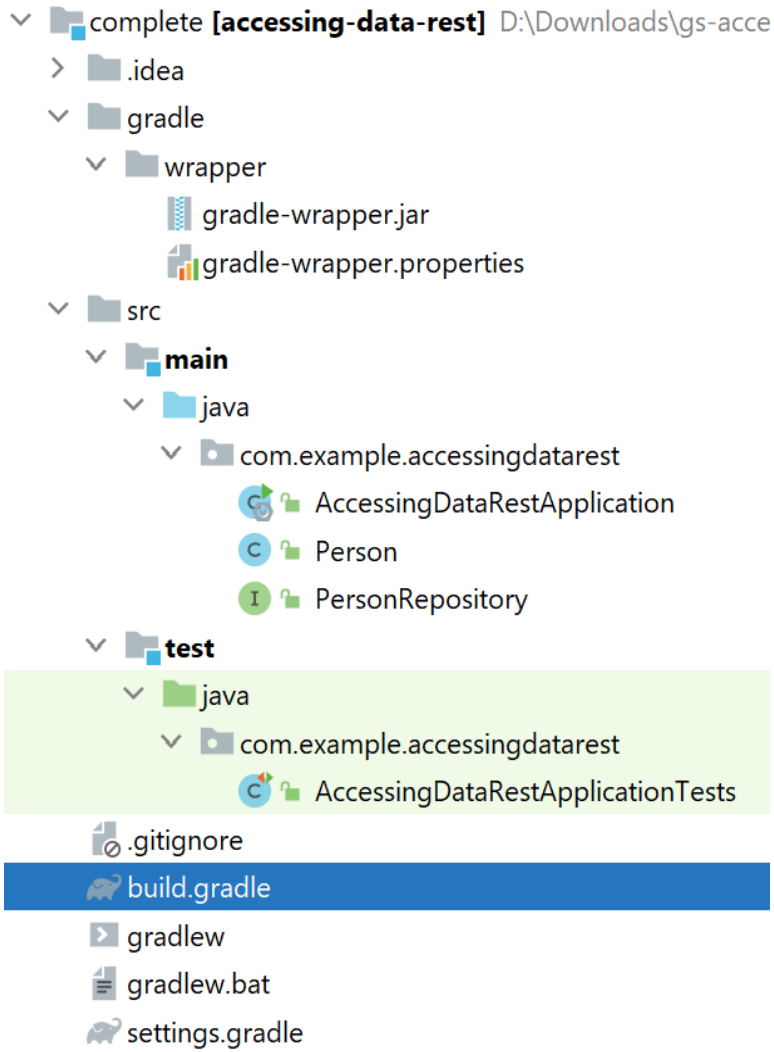
# Пример приложения

Анатомия простого примера приложения Spring Boot:

https://spring.io/guides/gs/rest-service/

(Список примеров: https://spring.io/guides)

# Пример приложения




Gradle wrapper (+файлы gradlew и gradlew.bat из корня проекта)

Исходный код приложения

Код тестов приложения

settings.gradle и build.gradle – описание структуры проекта

# Пример приложения

В большинстве проектов Java исходные коды хранятся в папках:

- src/main/java – основной код
- src/test/java – тестовый код

# Пример приложения

Файл .gitignore используется системой контроля версий Git для указания файлов, которые не нужно хранить в системе контроля версий, такие как результат сборки приложения (файлы .class и .jar).

Как правило папка .idea тоже не хранится в VCS, так как структура проекта уже описана в Gradle или Maven, и IDEA (или другая среда разработки) может импортировать проект из этих описаний.

# Пример приложения

Пример файла .gitignore:

```
.gradle/
build/
!**/src/main/**/build/
!**/src/test/**/build/

### IntelliJ IDEA ###
.idea/
out/
!**/src/main/**/out/
!**/src/test/**/out/
```

# Пример приложения

Остальные файлы:

- папка gradle – Gradle wrapper
  - файл gradlew – запуск Gradle wrapper для Linux
  - файл gradlew.bat – запуск Gradle wrapper для Windows
- файлы settings.gradle и build.gradle – описание структуры проекта и того, как его собирать

# Gradle

Основные используемые системы сборки – Maven и Gradle.

Gradle создан в 2007 году, текущая версия – 7.0.

Gradle использует для конфигурации проекта полноценный язык программирования; изначально Groovy, с версии 3.0 также поддерживается Kotlin.

# Gradle

Установка:

- скачать дистрибутив: https://gradle.org/releases/
- распаковать (например, в C:\Gradle)
- установить переменную среды JAVA_HOME:
  - **`export JAVA_HOME=/opt/jdk`** для Linux
  - **`set JAVA_HOME=C:\java\jdk-16+36`** для Windows
- запуск: **`C:\Gradle\bin\gradle.bat <задача>`**

# Gradle

Описание проекта Gradle находится в файлах:

- settings.gradle – из каких (под)проектов состоит сборка. В простейшем случае это один проект, но в большинстве случаев сборка состоит из нескольких (под)проектов
- build.gradle – для каждого (под)проекта указывает, как его собирать

# Gradle

Hello world:

- у нас только один (под)проект, называемый корневым проектом (root project). В этом случае settings.gradle может быть пустым, или содержать указание имени корневого проекта: rootProject.name = “HelloWorld”
- описание корневого проекта – в единственном файле build.gradle в корне проекта

# Gradle

build.gradle (https://github.com/osobolev/grademo/tree/master/1):

```
plugins {
    id("application")
}

application {
    mainClass.set("ru.mirea.example.Example1")
}
```

# Gradle – терминология

| | | |
|---|---|---|
| build | | Набор библиотек и приложений. Большие проекты, как правило, состоят из нескольких подпроектов |
| project | (под)проект | Ваша библиотека или приложение. Кроме исходного кода, для проекта должно быть описание (build.gradle) |
| task | задача | Задача, которую можно выполнить над проектом. Например, **`compileJava`** – скомпилировать исходный код на Java |
| plugin | плагин | Модуль Gradle, предназначенный для выполнения класса задач. Например, **`application`** – плагин для сборки приложений Java. Плагин добавляет описания задач, которые он умеет выполнять. |

# Gradle

Блок plugins служит для подключения плагинов для (под)проекта и по сути указывает тип этого (под)проекта:

```
plugins {
    id("application")
}
```

описывает, что проект является Java-приложением. Приложение отличается об библиотеки тем, что может быть запущено, т.е. имеет метод main.

# Gradle

Для указания того, метод main какого класса нужно запускать, используется блок конфигурации плагина:

```
application {
    mainClass.set("ru.mirea.example.Example1")
}
```

mainClass – это свойство плагина application.

# Gradle

В данном случае все остальные настройки берутся по умолчанию:

- исходные коды проекта находятся в папке src/main/java
- кодировка исходных файлов – кодировка по умолчанию
- версия языка Java – версия Java, которая указана в JAVA_HOME

# Gradle

Список задач, которые Gradle может выполнить для проекта, можно вывести так (нужно запускать в корне проекта, где есть settings.gradle):

```
C:\Gradle\bin\gradle tasks
```

# Gradle

Вывод:

```
Application tasks
-----------------
run - Runs this project as a JVM application
...
Build Setup tasks
-----------------
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.
```

# Gradle

Пример вызова:

```
$ C:\Gradle\bin\gradle run
> Task :run
Hello

BUILD SUCCESSFUL in 2s
2 actionable tasks: 1 executed, 1 up-to-date
```

# Gradle wrapper

Запуск gradle.bat из скачанного дистрибутива Gradle неудобен:

- нужно знать, какую версию Gradle скачивать
- при запуске сборки проекта нужно указывать путь к скачанному Gradle

# Gradle wrapper

Для решения этих проблем используется Gradle wrapper – загрузчик Gradle:

- он лежит в папке gradle/wrapper, хранящей файлы:
  - gradle-wrapper.jar – код загрузчика
  - gradle-wrapper.properties – конфигурация загрузчика; главное свойство – distributionUrl: включает версию Gradle:
    distributionUrl=https\://services.gradle.org/distributions/gradle-7.0-all.zip
- запускается файлами gradlew.bat (Windows) или gradlew (Linux). При запуске сам скачивает при необходимости Gradle нужной версии и запускает.

# Gradle wrapper

Gradle wrapper как правило добавляется в систему хранения версий (так как его размер небольшой, это не создает проблем).

Как только вы скачаете проект с Gradle wrapper из VCS (например, с GitHub), вы можете просто выполнить

**`gradlew build`**

и у вас автоматически скачается и запустится нужная версия Gradle (при условии, что у вас установлена переменная JAVA_HOME).

# Gradle wrapper

Создать Gradle wrapper можно командой

```
C:\Gradle\bin\gradle wrapper
```

или

```
C:\Gradle\bin\gradle wrapper --distribution-type=all
```

Последний вариант при скачивании Gradle скачивает также и его исходный код (полезно для отладки сложных скриптов сборки).

# Gradle wrapper

Если в одном проекте у вас уже есть Gradle wrapper, можно просто перенести его в другой проект, скопировав папку gradle и файлы gradlew/gradlew.bat.

# Gradle wrapper

Шаги инициализации проекта с Gradle wrapper:

1. скачать и распаковать Gradle
2. установить JAVA_HOME
3. создать в папке проекта пустой settings.gradle
4. запустить **`gradle wrapper`**

Как вариант, вместо шагов 3, 4 можно выполнить Wizard для создания проекта **`gradle init`**

# Gradle tasks

Как правило, задачи (tasks) создаются плагинами (рекомендуемый способ). Но можно их создавать и вручную:

```
tasks.register(“hello”) {
    doLast {
        println(“Hello task”)
    }
}
```

создает задачу hello, которую можно выполнить:

gradlew hello

# Gradle tasks

Между задачами могу быть зависимости:

```
tasks.register(“hello”) {
    dependsOn(“beforeHello”)
    doLast {
        println(“Hello task”)
    }
}
tasks.register(“beforeHello”) {
    doLast {
        println(“Before hello”)
    }
}
```

Задача beforeHello должна быть выполнена перед hello, или иными словами выполнение задачи hello зависит от завершения задачи beforeHello

# Gradle tasks

Есть предопределенные типы задач:

```
tasks.register(“copyConfig”, type: Copy) {
    from(“config”)
    into(“distr”)
    exclude(“README.md”)
}
```

создает задачу copyConfig, копирующую файлы из папки config в папку distr, кроме файла README.md.

Кроме Copy есть также тип Zip, копирующий файлы в архив.

# Фазы выполнения

Файлы settings.gradle и build.gradle содержат обычный исполняемый код на языке Groovy (для Kotlin эти файлы должны называться settings.gradle.kts и build.gradle.kts).

# Фазы выполнения

При запуске Gradle эти файлы выполняются. Но выполнение идет в три фазы:

- фаза инциализации: выполнение settings.gradle
- фаза конфигурации: при выполнении build.gradle создаются и конфигурируются задачи
- фаза выполнения: запускается указанная при запуске Gradle задача и те задачи, от которых она зависит

# Фазы выполнения

Описание сборки обычным исполняемым кодом (в отличие от декларативного описания) имеет и плюсы, и минусы.

Плюсы:

- можно отлаживать; как через println, так и отладчиком IDE
- максимальная гибкость

Минусы:

- сложная логика сборки затрудняет понимание

# Фазы выполнения

Предпочитаемый подход в современном Gradle – выносить сложную логику в плагины, а в файлах build.gradle оставлять только декларативную конфигурацию этих плагинов.

# Фазы выполнения

settings.gradle:

```
println("Инициализация")
```

build.gradle:

```
println("Конфигурация")

tasks.register("configured") {
    println("Конфигурация задачи configured")
}

tasks.register("test") {
    doLast {
        println("Выполнение задачи test")
    }
}

tasks.register("testBoth") {
    doFirst {
        println("Выполнение задачи testBoth 1")
    }
    doLast {
        println("Выполнение задачи testBoth 2")
    }
    println("Конфигурация задачи testBoth")
}
```

# Фазы выполнения

gradlew testBoth:

```
Инициализация

> Configure project :
Конфигурация
Конфигурация задачи testBoth

> Task :testBoth
Выполнение задачи testBoth 1
Выполнение задачи testBoth 2

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

# Фазы выполнения

При сложных скриптов важно понимать, какая часть кода выполняется при конфигурации, а какая – при выполнении.

Те части задачи, которые требуется выполнять при ее выполнении, должны находиться в блоках doLast/doFirst.

# Языки Groovy и Kotlin

Язык Groovy – расширение языка Java с более свободным синтаксисом. Поэтому во многих примерах вы можете увидеть

id ‘application’

вместо

id(“application”)

так как Groovy позволяет использовать одинарные кавычки для строк и не писать скобочки при вызове методов.

# Языки Groovy и Kotlin

Я будут придерживаться более строгого синтаксиса – только с двойными кавычками и наличием скобок при вызове методов.

Это при необходимости упростит миграцию скриптов на более строгий язык Kotlin – в большинстве случаев будет достаточно просто переименовать файлы *.gradle в *.gradle.kts.

# Языки Groovy и Kotlin

Плюсы Kotlin:

- это статически типизированный язык, и IDE может показать ошибки в вашем скрипте до его выполнения
- если вы пишете для Android, вы скорее всего и так используете Kotlin

Минусы Kotlin:

- при первом запуске компилируется медленнее
- так как изначально Gradle рассчитан на Groovy, иногда требуются адаптеры типа closureOf

# Языки Groovy и Kotlin

Оба языка определяют краткую форму записи параметров методов-замыканий. Так код,

```
plugins {
    id("application")
}
```

означает вызов метода plugins с передачей ему параметра-замыкания, которое в свою очередь вызывает метод id с параметром “application”.

# Конфигурация проектов

Более расширенный Hello World (https://github.com/osobolev/grademo/tree/master/2):

settings.gradle:

rootProject.name = "demo2“

Таким образом задается имя корневого проекта (по уиолчанию имя проекта = имени папки, в которой он находится).

# Конфигурация проектов

build.gradle:

```
plugins {
    id("application")
}

group = "ru.mirea.example"
version = "0.1"
description = "Пример приложения"

java {
    sourceCompatibility = JavaVersion.VERSION_11
}

tasks.withType(JavaCompile) {
    options.encoding = "UTF-8"
}
```

# Конфигурация проектов

- параметры group и version описывают группу и версию, под которой проект будет публиковаться. Если вы планируете использование вышего проекта другими разработчиками, эти параметры нужно указывать
- параметр description – человекочитаемое описание проекта
- sourceCompatibility указывает, какую версию Java использует проект
- options.encoding указывает кодировку исходного кода

# Конфигурация проектов

Параметры sourceCompatibility и options.encoding лучше описывать явно и не полагаться на умолчания.

Для задания нестандартного пути к исходным файлам можно использовать

```
sourceSets {
    main {
        java {
            srcDir("src")
        }
    }
}
```

# Управление зависимостями

Кроме запуска компиляции, Gradle умеет автоматически скачивать нужные для сборки приложения библиотеки. Для этого нужно указать:

- репозиторий, из которого качать библиотеки (обычно это mavenCentral: https://repo1.maven.org/maven2)
- “адрес” библиотеки в виде тройки
  - group – группа, в которую входит библиотека
  - name – имя библиотеки
  - version – версия библиотеки

# Управление зависимостями

```
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.apache.commons:commons-lang3:3.12.0")
}
```

- group=org.apache.commons
- name=commons-lang3
- version=3.12.0

# Управление зависимостями

При таком объявлении в build.gradle библиотека Apache Commons Lang будет доступна при компиляции и выполнении приложения:

```
import org.apache.commons.lang3.StringUtils;


public class Example2 {
    public static void main(String[] args) {
        String message = StringUtils.center(" Hello world ", 80, '*’);
        System.out.println(message);
    }
}
```

# Управление зависимостями

При объявлениях зависимостей в блоке dependencies указываются конфигурации зависимостей (implementation – одна из возможных конфигураций).

Конфигурация зависимостей (dependency configuration) указывает, на что распространяется указанная библиотека. В случае конфигурации implementation библиотека используется при:

- компиляции как основного, так и тестового кода
- выполнении как основного, так и тестового кода

# Стандартные конфигурации

Большинство задач сборки проектов Java и конфигурации зависимостей определяются плагином “java”. Плагин “application” неявно применяет плагин “java” и добавляет свои собственные задачи (такие как “run”).

# Стандартные конфигурации

| | main компиляция | main runtime | test компиляция | test runtime |
|---|---|---|---|---|
| implementation | + | + | + | + |
| compileOnly | + | – | – | – |
| runtimeOnly | – | + | – | + |
| testImplementation | – | – | + | + |
| testCompileOnly | – | – | + | – |
| testRuntimeOnly | – | – | – | + |

# Стандартные конфигурации

- compileOnly используется для написания библиотек/приложений, работающих в некоторой среде, которая сама предоставляет эту библиотеку. Пример: сервлет в контейнере сервлетов.
- runtimeOnly используется для выбора реализации некоторого сервиса. Примеры: драйвер JDBC или реализация SLF4J.

# Сложные проекты

Большинство реальных проектов состоят не только из корневого проекта, но из нескольких под-проектов.

При этом необходимо описание в settings.gradle информации о:

- расположении этих под-проектов (свойство project.projectDir)
- именах этих под-проектов (свойство project.name)

# Сложные проекты

Пример сложного проекта (https://github.com/osobolev/grademo/tree/master/3):

settings.gradle:

```
rootProject.name = "demo3"

include("lib")
include("app")

for (project in rootProject.children) {
    project.projectDir = file("subprojects/${project.name}")
}
```

# Сложные проекты

Метод include(name) создает новый под-проект с логическим именем name.

По умолчанию папка проекта совпадает с его логическим именем. Если это не так, нужно назначить папку вручную:

project(“:lib”).projectDir = file(“subprojects/lib”)

Стандартный метод file() возвращает путь относительно текущего проекта.

# Сложные проекты

Получение под-проекта с именем name в скриптах Gradle осуществляется с помощью

project(“:name”)

Также при необходимости мы можем сослаться на задачу task проекта name с использованием синтаксиса “:name:task”.

# Сложные проекты

Gradle позволяет при запуске задач указывать конкретный проект, задачу которого мы хотим запустить:

gradlew :lib:build

По умолчанию запускаются задачи всех под-проектов.

# Сложные проекты

Так как часть конфигурации у всех под-проектов одинаковая, ее можно задать в build.gradle корневого проекта:

```
subprojects {
    apply(plugin: "java")

    group = "ru.mirea.example"
    version = "0.1"

    java {
        sourceCompatibility = JavaVersion.VERSION_11
    }

    tasks.withType(JavaCompile) {
        options.encoding = "UTF-8"
    }

    repositories {
        mavenCentral()
    }
}
```

# Сложные проекты

Для описания Java-библиотек используется плагин “java-library”:

```
plugins {
    id("java-library")
}

dependencies {
    api("com.google.guava:guava:30.1.1-jre")
    implementation("org.apache.commons:commons-lang3:3.12.0")
}
```

# Сложные проекты

Библиотека отличается от приложения тем, что ее нельзя запустить, но ее классы можно использовать в других проектах. Плагин java-library создает дополнительные конфигурации зависимостей в дополнене к стандартным, определяемым плагином java:

- api
- compileOnlyApi
- testCompileApi

# Сложные проекты

Конфигурация api, так же как implementation, позволяет использовать библиотеку при компиляии и при выполнении кода. Разница между ними в том, что если другой под-проект app использует нашу библиотеку lib, то:

- те зависимости, которые lib определила как api, будут также доступны при компиляции app
- те зависимости, которые lib определила как implementation, НЕ будут также доступны при компиляции app

# Сложные проекты

Иными словами, implementation – это то, что библиотека использует для своей внутренней реализации, а api – это то, что часть внешнего интерфейса библиотеки.

Как правило, те типы, которые используются в public-методах библиотеки (типы параметров, типы возвращаемых значений, типы исключений throws) должны подключаться как api.

Если же какой-то класс используется только внутри тела public-метода или только в private-методах, его можно подключать как implementation.

# Сложные проекты

В нашем примере guava подключается как api, а commons-lang3 – как implementation:

```
import com.google.common.collect.ImmutableList;
import org.apache.commons.lang3.StringUtils;

public final class MyLibrary {

    public static ImmutableList<String> getGreeting() {
        return ImmutableList.of("Hello", "world");
    }

    public static String createMessage(Iterable<String> strings) {
        return StringUtils.center(" " + String.join(" ", strings) + " ", 80, '*');
    }
}
```

api – используется в сигнатуре public-метода

implementation – используется только внутри метода

# Сложные проекты

Другие под-проекты могут использовать под-проект lib следующим образом:

```
dependencies {
    implementation(project(":lib"))
}
```

# Сложные проекты

В последние несколько лет Gradle не рекомендует использовать блок subprojects и apply для конфигурации под-проектов.

Для вынесения общего кода под-проектов рекомендуется писать собственные плагины.

В общем случае плагин – это класс, реализующий интерфейс Plugin, который конфигурирует переданный ему проект (создает в нем задачи).

# Сложные проекты

К счастью, есть простой способ написания своих плагинов. Для этого нужно в корне проекта создать папку buildSrc, в ней создать:

- пустой файл settings.gradle
- файл build.gradle
- подпапку src/main/groovy, в которую помещать файлы вида
  - my-plugin.gradle

В этом случае скрипт my-plugin.gradle будет доступен как плагин с именем “my-plugin”

# Сложные проекты

Файл build.gradle для buildSrc должен содержать

```
plugins {
    id("groovy-gradle-plugin")
}
```

# Сложные проекты

Для Kotlin должно быть так:

- пустой файл settings.gradle.kts
- файл build.gradle.kts:

```
plugins {
    `kotlin-dsl`
}
repositories {
    mavenCentral()
}
```

- подпапку src/main/kotlin, в которую помещать файлы вида
  - my-plugin.gradle.kts

# Сложные проекты

Пример такого подхода:
https://github.com/osobolev/grademo/tree/master/4

Он определяет вместо стандартных плагинов java-library и application свои: my-lib и my-app. Оба своих плагина в свою очередь подключают плагин my-code, который содержит логику, общую для библиотек и приложений. При этом my-lib также подключает “java-library”, а my-app – “application”.

В под-проектах мы уже используем для библиотек my-lib, а для приложения my-app.