

Lab 3. Working with Gradle: Setting Up a Gradle Project, Understanding Build Scripts (Groovy and Kotlin DSL), Dependency Management and Task Automation.

Gradle is a modern build automation tool designed to be highly flexible, fast, and scalable. It is widely used in Java projects, Android development, and many multi-language projects. Here's what makes Gradle stand out:

- **Flexible Build Scripts:** Gradle uses a Domain Specific Language (DSL) based on either Groovy (by default) or Kotlin. This provides a more dynamic and expressive way to define build logic compared to static XML configurations (as used in Maven).
- **Incremental Builds:** Gradle optimizes build times by determining what parts of the project have changed and rebuilding only those parts.
- **Task Automation:** Everything in Gradle is treated as a task, allowing you to create custom tasks or reuse existing ones for compiling code, running tests, packaging, and more.
- **Dependency Management:** Like Maven, Gradle can automatically download and manage dependencies from remote repositories (e.g., Maven Central, JCenter, or custom repositories).

1: Setting Up a Gradle Project

- **Install Gradle** (If you haven't already):
 - Follow Gradle installation Program 1
- **Create a new Gradle project:** You can set up a new Gradle project using the Gradle Wrapper or manually. Using the Gradle Wrapper is the preferred approach as it ensures your project will use the correct version of Gradle.
- **To create a new Gradle project using the command line:**

gradle init --type java-application

This command creates a new Java application project with a sample **build.gradle** file.

2: Understanding Build Scripts

Gradle uses a DSL (Domain-Specific Language) to define the build scripts. Gradle supports two DSLs:

- Groovy DSL (default)
- Kotlin DSL (alternative)

Groovy DSL: This is the default language used for Gradle build scripts (build.gradle). Example of a simple build.gradle file (Groovy DSL):

Groovy DSL: This is the default language used for Gradle build scripts (**build.gradle**).

*Example of a simple **build.gradle** file (Groovy DSL):*

Kotlin DSL: Gradle also supports Kotlin for its build scripts (**build.gradle.kts**).

*Example of a simple **build.gradle.kts** file (Kotlin DSL):*

Difference between Groovy and Kotlin DSL:

- **Syntax:** Groovy uses a more concise, dynamic syntax, while Kotlin offers a more structured, statically-typed approach.
- **Error handling:** Kotlin provides better error detection at compile time due to its static nature.

3: Dependency Management

Gradle provides a powerful dependency management system. You define your project's dependencies in the dependencies block.

1. Adding dependencies:

- Gradle supports various dependency scopes such as implementation, compileOnly, testImplementation, and others.

Example of adding a dependency in **build.gradle** (Groovy DSL):

```
dependencies {  
  
    implementation 'com.google.guava:guava:30.1-jre'  
  
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.1'  
  
}
```

Example in **build.gradle.kts** (Kotlin DSL):

```
dependencies {  
  
    implementation("com.google.guava:guava:30.1-jre")  
  
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.7.1")  
  
}
```

2. Declaring repositories:

To resolve dependencies, you need to specify repositories where Gradle should look for them. Typically, you'll use Maven Central or JCenter, but you can also configure private repositories.

Example (Groovy):

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

Example (Kotlin):

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

4: Task Automation

Gradle tasks automate various tasks in your project lifecycle, like compiling code, running tests, and creating builds.

1. **Using predefined tasks:** Gradle provides many predefined tasks for common activities, such as:

- **build** – compiles the project, runs tests, and creates the build output.
- **test** – runs tests.
- **clean** – deletes the build output.

Example of running the build task:

```
gradle build
```

3. **Creating custom tasks:** You can define your own tasks to automate specific actions. For example, creating a custom task to print a message.

- **Example Groovy DSL:**

```
task printMessage {
```

```
doLast {  
    println 'This is a custom task automation'  
}  
}
```

- **Example Kotlin DSL:**

```
tasks.register("printMessage") {  
    doLast {  
        println("This is a custom task automation")  
    }  
}
```

5: Running Gradle Tasks

To run a task, use the following command in the terminal:

```
gradle <task-name>
```

For example:

- To run the build task: **gradle build**
- To run a custom task: **gradle printMessage**

6: Advanced Automation

You can define task dependencies and configure tasks to run in a specific order. Example of task dependency:

```
task firstTask {  
  
    doLast {  
  
        println 'Running the first task'    }  
}
```

```

    }

}

task secondTask {

    dependsOn firstTask

    doLast {

        println 'Running the second task'

    }

}

```

In this case, **secondTask** will depend on the completion of **firstTask** before it runs.

Working with Gradle Project (Groovy DSL):

Step 1: Create a new Project

(NOTE: Create a new directory for your project and navigate to it:
mkdir LAB3 and cd LAB3)

gradle init --type java-application

- while creating project it will ask necessary requirement:
 - **Enter target Java version (min: 7, default: 21):** 17
 - **Project name (default: program3-groovy):** groovyProject
 - **Select application structure:**
 - 1: Single application project
 - 2: Application and library project
 - **Enter selection (default: Single application project) [1..2]** 1
 - **Select build script DSL:**
 - 1: Kotlin
 - 2: Groovy
 - **Enter selection (default: Kotlin) [1..2]** 2
 - **Select test framework:**

- 1: JUnit 4
- 2: TestNG
- 3: Spock
- 4: JUnit Jupiter
 - Enter selection (default: JUnit Jupiter) [1..4] 1
- Generate build using new APIs and behavior (some features may change in the next minor release)? (default: no) [yes, no]
 - no

Step 2: **build.gradle** (Groovy DSL) *(just open and refer this file for sample)*

After project creation, the directory structure typically looks like this:

```
LAB3/
├── build.gradle // The primary build script (Groovy DSL by default)
├── gradle/ // Contains Gradle wrapper files (if generated)
├── gradlew // Unix shell script to run Gradle wrapper
├── gradlew.bat // Windows batch script for Gradle wrapper
├── settings.gradle // Contains project settings and names
└── src
    ├── main
    │   ├── java
    │   └── App.java // Your main application source file
    └── test
        └── java
            └── AppTest.java // Your test cases
```

Explanation of Components:

- **build.gradle:** This is the main build script written in Groovy (or Kotlin if you choose). It defines plugins, repositories, dependencies, and tasks.
- **settings.gradle:** A small script that defines the project's name and, in multi-project builds, the included subprojects.
- **gradlew / gradlew.bat:** The Gradle wrapper scripts. They allow you to run Gradle without requiring a separate installation on every machine by automatically downloading the correct Gradle version.
- **src/main/java:** Contains your application's source code.
- **src/test/java:** Contains your unit tests.

Step 3: App.java (**Edit file and update below code**)

```
package org.example;

public class App {

    public static void main(String[] args) {

        double num1 = 5;

        double num2 = 10;

        double sum = num1 + num2;

        System.out.printf("The sum of %.2f and %.2f is %.2f\n", num1, num2, sum);

    }

}
```

Step 4: Run Gradle Commands

gradle build

- To **run** the project:

gradle run

- To **test** the project:

gradle test