EXPERIMENT-3

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

Gradle Project Overview

1: Setting Up a Gradle Project

- **Install Gradle** (If you haven't already):
- **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-applicationgradle 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):

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation
'org.springframework.boot:spring-boot-starter-
web:2.5.4'
}

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

Kotlin DSL: Gradle also supports Kotlin for its build scripts (**`build.gradle.kts`**). Example of a simple **`build.gradle.kts`** file (Kotlin DSL):

```
plugins {
    kotlin("jvm") version "1.5.21"
}

repositories {
    mavenCentral()
}

dependencies {
```

```
implementation("org.springframework.boot:spring-
boot-starter-web:2.5.4")
}

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

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.

Task Block: Tasks define operations in Gradle, and they can be executed from the command line using `gradle <task-name>`.

In Groovy DSL:

```
task hello {
    doLast {
        println 'Hello, Gradle!'
    }
}
```

In Kotlin DSL:

```
tasks.register("hello") {
    doLast {
        println("Hello, Gradle!")
    }
}
```

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.
2. 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 (Kotlin DSL)

- while creating project it will ask necessary requirement:
  - **Enter target Java version (min: 7, default: 21):** 17
  - **Project name (default: program3-kotlin):** kotlinProject
  - **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]** 1
  - **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

```
C:\Users\ItiShree>cd desktop

C:\Users\ItiShree\Desktop>mkdir GRADLEPROJECTNEW

C:\Users\ItiShree\Desktop>cd GRADLEPROJECTNEW

C:\Users\ItiShree\Desktop\GRADLEPROJECTNEW>gradle init --type java-application

Enter target Java version (min: 7, default: 21): 17

Project name (default: GRADLEPROJECTNEW): kotlinProject

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] 1

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


> Task :init
Learn more about Gradle by exploring our Samples at https://docs.gradle.org/8.12/samples/sample_building_java_applications.html

BUILD SUCCESSFUL in 2m 24s
```

Step 2: build.gradle.kts (Kotlin DSL)

```
plugins {
    kotlin("jvm") version "1.8.21"
    application
}
repositories {
    mavenCentral()
}
```

```
dependencies {
    implementation(kotlin("stdlib"))
    testImplementation("junit:junit:4.13.2")
}
application {
    mainClass.set("com.example.MainKt")
}
tasks.test {
    useJUnit()
    testLogging {
        events("passed", "failed", "skipped")
        exceptionFormat =
org.gradle.api.tasks.testing.logging.TestExceptionFormat.FULL
        showStandardStreams = true
    }
    outputs.upToDateWhen { false }
}
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(17))
```

```
    }
}
```

Step 3: `Main.kt` (Change file name and update below code)

- After creating project **change the file name**.
- Manually navigate the folder path like **src/main/java/org/example/**
- Change the file name **App.java** to **Main.kt**
- After then open that file and copy the below code and past it, save it.

```
package com.example

fun addNumbers(num1: Double, num2: Double): Double {
    return num1 + num2
}

fun main() {
    val num1 = 10.0
    val num2 = 5.0
    val result = addNumbers(num1, num2)
    println("The sum of $num1 and $num2 is: $result")
```

}

Step 4: `MainTest.kt` (JUnit Test) (Change file name and update below code)

- After creating project **change the file name**.
- Manually navigate the folder path
  like **src/test/java/org/com/example/**
- Change the file name **MainTest.java** to **`MainTest.kt`**
- After then open that file and copy the below code and past it, save it.

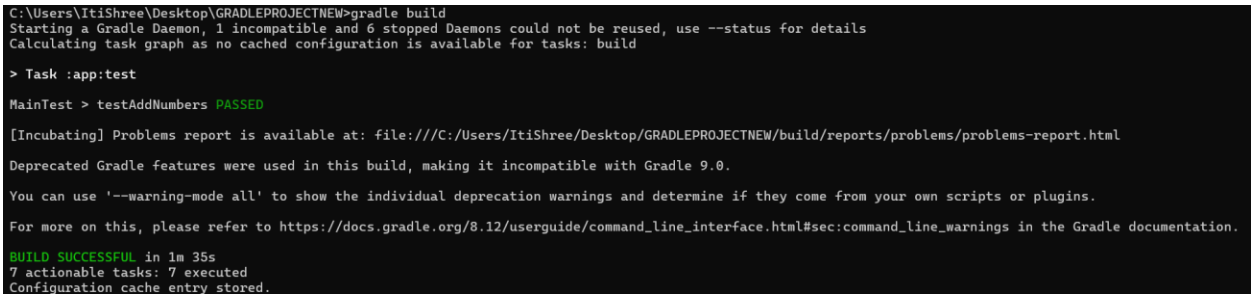
```
package com.example

import org.junit.Assert.*

import org.junit.Test

class MainTest {


    @Test

    fun testAddNumbers() {

        val num1 = 10.0

        val num2 = 5.0

        val result = addNumbers(num1, num2)

        assertEquals("The sum of $num1 and $num2 should be 15.0", 15.0,
result, 0.001)
```

```
    }
}
```

Step 5: Run Gradle Commands

- **To build the project:**

```
gradle build
```



- **To run the project:**

```
gradle run
```

```
C:\Users\ItiShree\Desktop\GRADLEPROJECTNEW>gradle run
Calculating task graph as no cached configuration is available for tasks: run

> Task :app:run
The sum of 10.0 and 5.0 is: 15.0
```

- **To test the project:**

```
gradle test
```

```
C:\Users\ItiShree\Desktop\GRADLEPROJECTNEW>gradle test
Calculating task graph as no cached configuration is available for tasks: test

> Task :app:test

MainTest > testAddNumbers PASSED

[Incubating] Problems report is available at: file:///C:/Users/ItiShree/Desktop/GRADLEPROJECTNEW/build/reports/problems/problems-report.html

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or plugins.

For more on this, please refer to https://docs.gradle.org/8.12/userguide/command_line_interface.html#sec:command_line_warnings in the Gradle documentation.

BUILD SUCCESSFUL in 2s
3 actionable tasks: 1 executed, 2 up-to-date
Configuration cache entry stored.
C:\Users\ItiShree\Desktop\GRADLEPROJECTNEW>
```

`build.gradle` (Groovy DSL)

```
plugins {
    id 'application'
}

application {
    mainClass = 'com.example.AdditionOperation'
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'junit:junit:4.13.2'
}

test {
```

```
    outputs.upToDateWhen { false }

    testLogging {
        events "passed", "failed", "skipped"
        exceptionFormat "full"
        showStandardStreams = true
    }
}
```

Step 3: `AdditionOperation.java`(Change file name and update below code)

- After creating project **change the file name**.
- Manually navigate the folder path like **src/main/java/org/example/**
- Change the file name **App.java** to **AdditionOperation.java**
- After then open that file and copy the below code and past it, save it.

```
package com.example;

public class AdditionOperation {
    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: `AdditionOperationTest.java` (JUnit Test) (Change file name and update below code)

- After creating project **change the file name**.
- Manually navigate the folder path like **src/test/java/org/example/**

- Change the file name **AppTest.java** to **`AdditionOperationTest.java`**
- After then open that file and copy the below code and past it, save it.

```
package com.example;

import org.junit.Test;
import static org.junit.Assert.*;

public class AdditionOperationTest {

    @Test
    public void testAddition() {
        double num1 = 5;
        double num2 = 10;
        double expectedSum = num1 + num2;

        double actualSum = num1 + num2;

        assertEquals(expectedSum, actualSum,
0.01);
    }
}
```

Step 5: Run Gradle Commands

- To **build** the project:

```
gradle build
```

- To **run** the project:

```
gradle run
```

To test the project:

```
gradle test
```