1. Introduction

With the Gradle project, understand build scripts (using both Groovy and Kotlin DSL), manage dependencies, and automate tasks.

2. Prerequisites

- IntelliJ IDEA installed on your system.
- Java Development Kit (JDK) installed and configured.

3. Setting Up a New Gradle Project in IntelliJ IDEA

1. **Open IntelliJ IDEA:** Launch the IntelliJ IDEA application.

2. **Create New Project:** On the welcome screen, click on "New Project".

3. **Select Gradle:** In the left-hand panel of the "New Project" window, choose "Gradle".

4. **Project SDK:** Ensure that a valid Java SDK is selected in the "Project SDK" dropdown menu. If no SDK is listed, click "Add JDK..." and navigate to your Java installation directory to select it.

5. **Language Selection:** At this stage, you can choose between Groovy and Kotlin DSL for your build scripts. For illustrative purposes, we will outline the process for both.

    - **Groovy DSL:** To use Groovy DSL, leave the "Language" dropdown set to "Groovy" (which is often the default).
    - **Kotlin DSL:** To use Kotlin DSL, select "Kotlin" from the "Language" dropdown menu.
6. **Group and Artifact Configuration:**

    - **Group:** Enter the group ID for your project. This is typically a reverse domain name of your organization (e.g., `com.example`).
    - **Artifact:** Enter the name of your project (e.g., `my-gradle-project`).
7. **Project Naming and Location:**

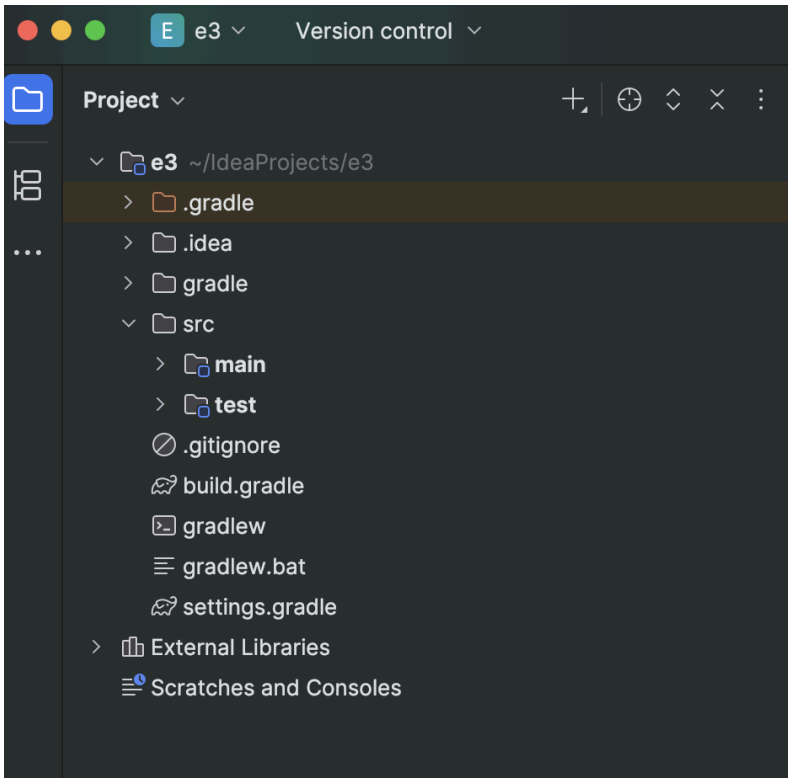
    - **Name:** Specify a name for your project. This will also be the name of the project directory by default.
    - **Location:** Choose the directory on your system where you want to save the project files.

8. **Generate Build Files:** Ensure that the option to "Generate build.gradle(.kts) and settings.gradle(.kts)" (the exact wording might vary slightly based on your DSL choice) is checked. This will instruct IntelliJ IDEA to create the basic Gradle build files.

Click "Create": Once you have configured all the settings, click the "Create" button. IntelliJ IDEA will then generate the basic Gradle project structure.

You should observe a project structure similar to the following in your project window:

```
my-gradle-project/ (or your chosen name)
├── .gradle/
├── .idea/
├── build.gradle (Groovy DSL) OR build.gradle.kts (Kotlin DSL)
├── gradle/
│       └── wrapper/
│               ├── gradle-wrapper.jar
│               └── gradle-wrapper.properties
├── settings.gradle (Groovy DSL) OR settings.gradle.kts (Kotlin DSL)
└── src/
    └── main/
            └── java/ (or kotlin/)
    └── test/
            └── java/ (or kotlin/)
```



4. Understanding Build Scripts (`build.gradle` / `build.gradle.kts`)

The `build.gradle` (for Groovy) or `build.gradle.kts` (for Kotlin) file is the core configuration file for your Gradle project. It defines plugins, dependencies, and tasks that Gradle uses to build your project.

4.1 Groovy DSL (`build.gradle`)

Groovy

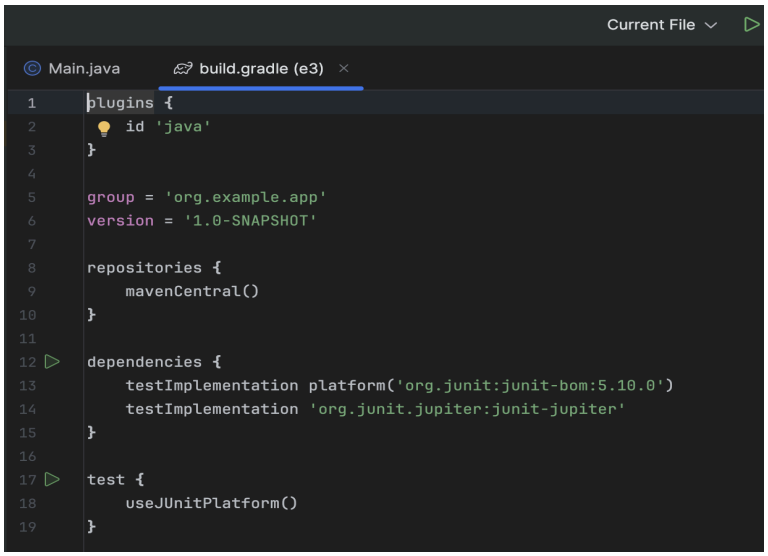
```
plugins {
    id 'java'
}

group = 'com.example'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.apache.commons:commons-lang3:3.12.0'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.10.2'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.10.2'
}

tasks.named('test') {
    useJUnitPlatform()
}
```



Explanation:

- **`plugins { ... }`**: This block declares Gradle plugins. The `java` plugin provides essential Java development capabilities like compilation and testing.
- **`group = '...'` and `version = '...'`**: These define the project's unique identifier and version, crucial for dependency management and publishing.
- **`repositories { ... }`**: This section specifies where Gradle should look for external libraries (dependencies). `mavenCentral()` is a widely used public Maven repository.
- **`dependencies { ... }`**: This block lists the external libraries your project relies on.
    - `implementation`: Dependencies required for the main source code of your project.
    - `testImplementation`: Dependencies needed for compiling and running your unit tests.
    - `testRuntimeOnly`: Dependencies required only during the execution of tests.
- **`tasks.named('test') { ... }`**: This configures an existing Gradle task named `test`. Here, it specifies that the JUnit Platform should be used to execute tests.

4.2 Kotlin DSL (`build.gradle.kts`)

Kotlin

```
plugins {
    java
}

group = "com.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.apache.commons:commons-lang3:3.12.0")
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.10.2")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.10.2")
}

tasks.named("test") {
    useJUnitPlatform()
}
```

Explanation:

The Kotlin DSL serves the same purpose as the Groovy DSL but utilizes Kotlin syntax. Key differences include:

- **Plugin Application:** Plugins are applied as properties (e.g., `java`).
- **String Literals:** Strings are enclosed in double quotes (`"`).
- **Dependency Declaration:** Dependency configurations are invoked as functions, with the dependency coordinates as string arguments within parentheses.

5. Understanding the `settings.gradle` / `settings.gradle.kts` File

The `settings.gradle` (Groovy) or `settings.gradle.kts` (Kotlin) file defines the structure of a multi-project Gradle build. For single-project builds, it primarily specifies the root project name.

5.1 Groovy DSL (`settings.gradle`)

Groovy
rootProject.name = 'my-gradle-project'

5.2 Kotlin DSL (`settings.gradle.kts`)

Kotlin
rootProject.name = "my-gradle-project"

In more complex projects with sub-projects, you would use the `include` keyword to define them (e.g., `include 'subproject1', 'subproject2'`).

6. Dependency Management

Gradle's dependency management system simplifies the inclusion of external libraries into your project. Dependencies are declared in the `dependencies` block of your `build.gradle` or `build.gradle.kts` file.

6.1 Adding Dependencies

To add a new dependency, add a line within the `dependencies` block, specifying the configuration (e.g., `implementation`, `testImplementation`) and the dependency coordinates (Group ID, Artifact ID, Version).

Example (Adding the Guava library):

Groovy DSL (`build.gradle`):

Groovy

```
dependencies {
    implementation 'com.google.guava:guava:33.0.0-jre'
    // ... other dependencies
}
```

Kotlin DSL (`build.gradle.kts`):

```
Kotlin
dependencies {
    implementation("com.google.guava:guava:33.0.0-jre")
    // ... other dependencies
}
```

6.2 Refreshing Dependencies

After modifying your build script, IntelliJ IDEA will typically prompt you to synchronize your Gradle project. You can also manually trigger this by:

- Clicking the "Reload All Gradle Projects" button in the Gradle tool window (usually located on the right side of the IDE). [1]
- [1. codingtechroom.com]
- [codingtechroom.com]
-
- Right-clicking on your `build.gradle` or `build.gradle.kts` file in the Project view and selecting "Load Gradle Changes".

Gradle will then download the specified dependencies and make them available for your project.

7. Task Automation

Gradle uses tasks to define and execute build processes. The `java` plugin automatically provides several common tasks.

7.1 Common Gradle Tasks

- `compileJava`: Compiles the Java source code in the `src/main/java` directory.
- `processResources`: Copies resources from the source sets to the output directories.
- `test`: Runs the unit tests located in the `src/test/java` directory.
- `jar`: Packages the compiled code and resources into a Java Archive (JAR) file.
- `build`: Assembles and tests the project. This task usually depends on other tasks like `compileJava`, `processResources`, and `test`.
- `clean`: Deletes the build output directory, allowing for a fresh build.

7.2 Viewing Available Tasks

You can view the list of available Gradle tasks in the "Gradle" tool window in IntelliJ IDEA. Expand your project, then navigate to the "Tasks" directory to see the categorized list of tasks.

7.3 Executing Tasks

You can execute Gradle tasks in several ways within IntelliJ IDEA:

- **Double-click:** Double-click on a task in the "Gradle" tool window to execute it.
- **Right-click and Run:** Right-click on a task and select "Run" from the context menu.
- **Gradle Console:** Open the "Gradle Console" (View -> Tool Windows -> Gradle) and type the task name followed by Enter (e.g., `gradle clean build`).

7.4 Creating Custom Tasks

You can define your own tasks in the `build.gradle` or `build.gradle.kts` file to automate specific actions.

Example (Creating a simple greeting task):

Groovy DSL (`build.gradle`):

Groovy
```
task greeting {
    doLast {
        println 'Hello from Gradle!'
    }
}
```

Kotlin DSL (`build.gradle.kts`):

Kotlin
```
tasks.register("greeting") {
    doLast {
        println("Hello from Gradle!")
    }
}
```

You can now run this custom task by double-clicking it in the Gradle tool window or using the Gradle console: `gradle greeting`.

7.5 Task Dependencies

Tasks can be configured to depend on other tasks, ensuring a specific order of execution.

Example (Making the `build` task depend on the `greeting` task):

Groovy DSL (`build.gradle`):

Groovy
```
build.dependsOn greeting
```

Kotlin DSL (`build.gradle.kts`):

```
tasks.named("build") {
    dependsOn(tasks.named("greeting"))
}
```

Now, when you execute the `build` task, the `greeting` task will be executed before the standard build process.

8. Running Your Code

For Java or Kotlin projects with a `main` method, the `java` plugin provides a `run` task to execute your application.

8.1 Executing the `run` Task

You can execute the `run` task from the Gradle tool window or by using the Gradle console: `gradle run`.