Step 2: Verify Installation

```
gradle -v
gradle --version
```

This should display the Gradle version.

3️⃣ Creating a Gradle Project in IntelliJ IDEA

Step 1: Open IntelliJ IDEA and Create a New Project

1. Click on **"New Project"**.
2. Select **"Gradle"** (under Java/Kotlin).
3. Choose **Groovy or Kotlin DSL (Domain Specific Language)** for the build script.
4. Set the **Group ID** (e.g., `com.example`).
5. Click **Finish**.

Step 2: Understanding Project Structure

```
my-gradle-project
├── build.gradle  (Groovy Build Script)
├── settings.gradle
├── src
│   ├── main
│   │   ├── java
│   │   └── resources
│   ├── test
│   │   ├── java
│   │   └── resources

```

4️⃣ Build and Run a Simple Java Application

Step 1: Modify `build.gradle` (Groovy DSL)

```
plugins {
    id 'application'
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.1'
}

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

```

Step 2: Create `Main.java` in `src/main/java/com/example`

```
package com.example;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello from Gradle!");
    }
}

```

Step 3: Build and Run the Project

- **In IntelliJ IDEA**, open the **Gradle tool window** (View → Tool Windows → Gradle).
- Click `Tasks > application > run`.
- Or run from terminal:

```
gradle run

```

5️⃣ Hosting a Static Website on GitHub Pages

Step 1: Create a `/docs` Directory

- Create `docs` inside the **root folder** (not in `src`).
- Add your **HTML, CSS, and images** inside `/docs`.

Step 2: Modify `build.gradle` to Copy Website Files (This is optional)

```
task copyWebsite(type: Copy) {
    from 'src/main/resources/website'
    into 'docs'
}

```

Step 3: Commit and Push to GitHub

```
git add .
git commit -m "Deploy website using Gradle"
git push origin main

```

Step 4: Enable GitHub Pages

- Go to **GitHub Repo → Settings → Pages**.
- Select the `/docs` **folder** as the source.

Your website will be hosted at:

```
https://yourusername.github.io/repository-name/

```

6️⃣ Testing the Website using Selenium & TestNG in IntelliJ IDEA

Step 1: Add Selenium & TestNG Dependencies in `build.gradle`

```
dependencies {
    testImplementation 'org.seleniumhq.selenium:selenium-java:4.28.1' // use the latest stable version
    testImplementation 'org.testng:testng:7.4.0' // use the latest stable version
}

test {
    useTestNG()
}

```

Step 2: Write a Test Script (`src/test/java/org/test/WebpageTest.java`)

```
package org.test;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import static org.testng.Assert.assertTrue;

public class WebpageTest {
    private static WebDriver driver;

    @BeforeTest
    public void openBrowser() throws InterruptedException {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        Thread.sleep(2000);
        driver.get("https://sauravsarkar-codersarcade.github.io/CA-GRADLE/");
    }

    @Test
    public void titleValidationTest(){
        String actualTitle = driver.getTitle();
        String expectedTitle = "Tripillar Solutions";
        Assert.assertEquals(actualTitle, expectedTitle);
        assertTrue(true, "Title should contain 'Tripillar'");

    }

    @AfterTest
    public void closeBrowser() throws InterruptedException {
        Thread.sleep(1000);
        driver.quit();
    }


}

```

Step 3: Run the Tests

- Open the **Gradle tool window** in IntelliJ.
- Click `Tasks > verification > test`. **"Recommended"**
- Or run from terminal:

```
gradle test // Fails sometimes due to terminal issues

```

7️⃣ Packaging a Gradle Project as a JAR

Step 1: Modify `build.gradle` for JAR Packaging

```
plugins {
    id 'java'
    id 'application'
}

application {
    mainClass = 'com.example.Main'
}
jar {
    manifest {
        attributes 'Main-Class': 'com.example.Main'  // This tells Java where to start execution
    }
}

```


Step 2: Build and Package the JAR

```
gradle jar

```

The JAR file will be generated in `build/libs/`.

Step 3: Run the JAR

```
java -jar build/libs/<my-gradle-project>.jar

Expected output:
Hello from Gradle!

```

8️⃣ Gradle Lifecycle & Common Commands

Task	Command	Description
Initialize Project	`gradle init`	Creates a new Gradle project.
Compile Code	`gradle build`	Compiles the project.
Run Application	`gradle run`	Runs the application.
Clean Build Files	`gradle clean`	Deletes old build files.

Run Tests	`gradle test`	Executes unit tests.
Generate JAR	`gradle jar`	Packages the project into a JAR.
Deploy to GitHub Pages	`git push origin main`	Pushes website to GitHub.

9️⃣ Conclusion

Gradle provides a **fast, flexible, and scalable build automation system**.

We covered:

1️⃣ **Project Setup** in IntelliJ IDEA
2️⃣ **Building & Running a Java Application**
3️⃣ **Hosting a Static Website on GitHub Pages**
4️⃣ **Testing with Selenium & TestNG**
5️⃣ **Packaging as a JAR**

✅ Now you are **ready to use Gradle for build automation!** 🚀

🌟 Gradle Build Lifecycle: A Simple & Colorful Guide 🚀

Gradle follows a **flexible, task-based lifecycle**, unlike Maven's fixed phases. The build lifecycle in Gradle is divided into three key stages:

🔹 1. Initialization Phase

📌 **What happens here?**

- Determines which projects are part of the build (for multi-project builds).
- Creates an instance of each project.

📌 **Example Command:**

```
gradle help

```

(Shows project details and verifies Gradle setup.)

🔹 2. Configuration Phase

📌 **What happens here?**

- Gradle loads and executes the `build.gradle` file.
- It **configures** tasks but does **not** execute them yet.

📌 **Example Command:**

```
gradle tasks

```

(Lists all available Gradle tasks in the project.)

🔹 3. Execution Phase

📌 **What happens here?**

- Gradle executes the **tasks requested by the user**.
- Dependencies are resolved dynamically.
- Supports **incremental builds** (only modified files are compiled).

📌 **Example Command:**

```
gradle build

```

(Builds the project by compiling and packaging files.)

🎯 Key Gradle Tasks & Commands

Task 🛠️	Command	Description
🧹 **Clean**	`gradle clean`	Deletes previous build files.
🛠️ **Compile**	`gradle compileJava`	Compiles the Java source code.
📦 **Package (JAR/WAR)**	`gradle jar`	Packages the project into a JAR file.
✅ **Test**	`gradle test`	Runs unit tests.
🚀 **Run Application**	`gradle run`	Executes the main Java application.
📥 **Dependency Resolution**	`gradle dependencies`	Displays dependency tree.
⏩ **Parallel Execution**	`gradle build --parallel`	Speeds up builds by running tasks in parallel.

🎨 Gradle Lifecycle Visualized

```
Initialization  →  Configuration  →  Execution
(Project setup)    (Tasks loaded)    (Tasks executed)

```

✅ **Gradle is flexible** – tasks can be **customized or skipped** based on project needs. Unlike Maven, Gradle allows **on-demand task execution** rather than following a strict lifecycle.

🌟 Conclusion

🎯 **Gradle is powerful because:**
✅ It **only executes what's necessary** (faster builds).
✅ It **supports parallel execution** for large projects.
✅ It gives developers **more control over task execution**.

With Gradle, you can **automate builds efficiently** and **improve performance** for Java, Kotlin, and Android projects. 🚀

🔄 Maven vs. Gradle: A Detailed Comparison

Feature	Maven 🏗️	Gradle 🚀
Build Script Language	XML (`pom.xml`)	Groovy/Kotlin (`build.gradle` or `build.gradle.kts`)
Performance	Slower due to full rebuilds	Faster with **incremental builds** and parallel execution
Dependency Management	Uses Maven Central & local repository	Supports Maven, Ivy, and custom repositories
Configuration Style	**Declarative** (XML-based, verbose)	**Declarative + Imperative** (more concise, script-based)
Ease of Use	More structured, but verbose	More flexible, but has a learning curve
Incremental Builds	No support (always rebuilds everything)	Supports **incremental builds** (only recompiles changed files)
Build Performance	Slower, builds from scratch	Faster due to task caching and incremental execution
Customization & Extensibility	Limited custom scripts, plugin-based	Highly customizable with dynamic tasks and scripts
Multi-Project Builds	Complex and slower	Native support, optimized for large projects
IDE Support	Supported by IntelliJ IDEA, Eclipse, VS Code	Supported by IntelliJ IDEA, Eclipse, VS Code
Dependency Resolution	Resolves dependencies sequentially	**Parallel dependency resolution** (faster)
Popularity	Older and widely used in enterprise projects	Gaining popularity, especially in **Android & Kotlin** projects
Default Lifecycle Phases	3 lifecycle phases (clean, build, site)	More flexible task execution model
Learning Curve	Easier for beginners due to structured XML	Slightly steeper due to script-based configuration
Best Suited For	**Stable Java projects**, enterprises, and legacy codebases	**Modern Java, Android, and Kotlin projects** needing high performance

🛠️ Which One Should You Choose?

✅ **Use Maven** if you need **stability, structured builds, and an easier learning curve**.
✅ **Use Gradle** if you want **faster builds, better performance, and flexibility** for complex projects.

🎯 **Final Verdict:** If you're working on a simple Java project, **Maven** is a good choice. However, for modern, large-scale, or Android projects, **Gradle** is the better option due to its **speed, flexibility, and scalability**. 🚀

Here is a KOTLIN DSL version for Gradle "This is Optional"

🎯 Gradle Kotlin Build Script (`build.gradle.kts`)

Here's a **simple Gradle build script** written in **Kotlin DSL (`build.gradle.kts`)** for a Java application. 🚀

📌 `build.gradle.kts` (Kotlin DSL for Gradle)

```
// Apply necessary plugins
plugins {
    kotlin("jvm") version "1.9.10"  // Kotlin plugin for JVM
    application  // Enables the `run` task
}

// Define project properties
group = "com.example"
version = "1.0.0"
application.mainClass.set("com.example.MainKt")  // Define the main class

// Repositories for dependencies
repositories {
    mavenCentral()  // Fetch dependencies from Maven Central
}

// Dependencies
dependencies {
    implementation(kotlin("stdlib"))  // Standard Kotlin library
    testImplementation("org.junit.jupiter:junit-jupiter:5.9.1")  // JUnit 5 for testing
}

// Enable JUnit 5 for tests
tasks.test {
    useJUnitPlatform()
}

// JAR packaging
tasks.jar {
    manifest {
        attributes["Main-Class"] = "com.example.MainKt"
    }
}

```

📌 `Main.kt` (Inside `src/main/kotlin/com/example/`)

```
package com.example

fun main() {
    println("Hello, Gradle with Kotlin DSL! 🚀")
```

```
}

```

📌 Running the Project

💻 Run the application

```
gradle run

```

📦 Build the project

```
gradle build

```

🧹 Clean the build files

```
gradle clean

```

📂 Generate a JAR file

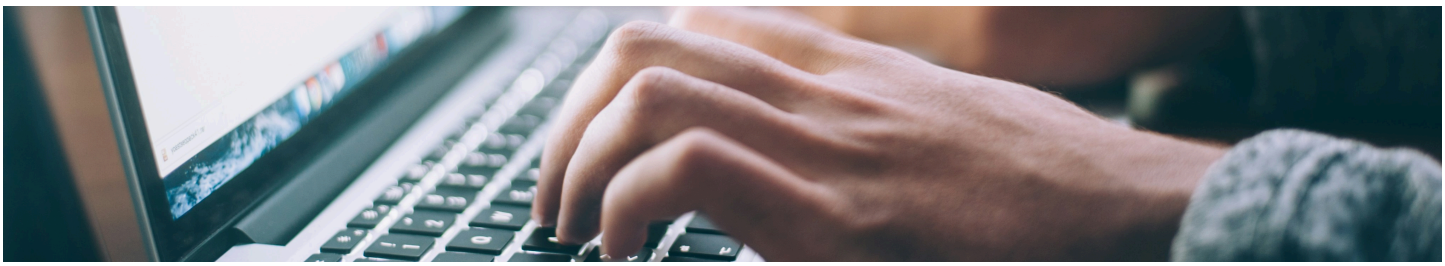
```
gradle jar

```

🌟 Why Use Kotlin DSL for Gradle?

✅ **Type-Safety** - Catch errors at compile-time.
✅ **Better IDE Support** - IntelliJ IDEA provides autocomplete and suggestions.
✅ **Interoperability** - Works seamlessly with Java and Kotlin projects.

This script sets up a **basic Kotlin/Java project with JUnit 5 support, dependency management, and a runnable JAR file**! 🚀



CA - Experiment 2 Part 2 : GRADLE KOTLIN DSL WORKFLOW || IntelliJ Idea

Here's a **detailed and well-formatted** documentation Gradle Kotlin DSL setup, including explanations, code snippets, and color-coded syntax (for reference when viewing in an IDE).

⚠️ **Important Note** : This experiment is only required if **students** or **VTU** asks you to show **GRADLE** with **KOTLIN DSL**, or else you can skip this.

Gradle Kotlin DSL: Setting Up & Building a Kotlin Project in IntelliJ IDEA

1️⃣ Setting Up the Gradle Project

Step 1: Create a New Project

1. Open **IntelliJ IDEA**.
2. Click on **File > New > Project**.
3. Select **Gradle** as the build system.
4. Choose **Kotlin** as the language.
5. Select **Gradle Kotlin DSL** (it will generate `build.gradle.kts`).
6. Name your project (e.g., `MVNGRDKOTLINDEMO`).
7. Set the **JDK** (use **JDK 17.0.4**, since that's your version).
8. Click **Finish**.

2️⃣ Understanding `build.gradle.kts`

After creating the project, the default `build.gradle.kts` file looks like this:

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
    kotlin("jvm") version "1.8.10"  // Use latest stable Kotlin version
    application
}

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

```

```
repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib"))  // Kotlin Standard Library
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.8.2")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.8.2")
}

tasks.test {
    useJUnitPlatform()
}

tasks.withType<KotlinCompile> {
    kotlinOptions.jvmTarget = "17"  // Match with your JDK version
}

application {
    mainClass.set("MainKt")  // Update this if using a package
}

```

3️⃣ Creating the Main Kotlin File

Now, create your `Main.kt` file inside `src/main/kotlin/`.
If you're using a package (e.g., `org.example`), it should look like:


```
package org.example

fun main() {
    println("Hello, Gradle with Kotlin DSL!")
}

```


If you're **not** using a package, remove the `package` line and ensure `mainClass.set("MainKt")` in `build.gradle.kts`.

4️⃣ Building and Running the Project

Build the Project

```
./gradlew build

```

Run the Project

```
./gradlew run

```

5️⃣ Packaging as a JAR

To run the project without IntelliJ, we need a **JAR file**.

Step 1: Create a Fat (Uber) JAR

Modify `build.gradle.kts`:

```
tasks.register<Jar>("fatJar") {
    archiveClassifier.set("all")
    duplicatesStrategy = DuplicatesStrategy.EXCLUDE
    manifest {
        attributes["Main-Class"] = "MainKt"
    }
    from(configurations.runtimeClasspath.get().map { if (it.isDirectory) it else zipTree(it) })
    with(tasks.jar.get() as CopySpec)
}

```

Step 2: Build the Fat JAR

```
./gradlew fatJar

```


Step 3: Run the Fat JAR

```
java -jar build/libs/MVNGRDKOTLINDEMO-1.0-SNAPSHOT-all.jar

```

6️⃣ Common Gradle Commands

Command	Description
`./gradlew build`	Builds the project
`./gradlew run`	Runs the application
`./gradlew test`	Runs the tests
`./gradlew clean`	Cleans the build directory
`./gradlew fatJar`	Creates a runnable JAR

7️⃣ Additional Features

Adding Custom Gradle Tasks

Example: A simple task that prints "Hello, Gradle!"

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

```

Run it with:

```
./gradlew hello
```