Gradle as a Build tool

**What is a Gradle?**

Gradle is an open source, advanced general-purpose build management system. It supports Groovy based Domain Specific Language (DSL) over XML. Gradle is used as a build automation tool for Java as well as Groovy projects.

Gradle supports the automatic download and configuration of dependencies or other libraries. Gradle is built on ANT, Maven and Ivy repositories and uses these repositories for retrieving these dependencies. This allows reusing the artifacts of existing build systems. Gradle supports multi-project and multi-artifact builds.

**Why, Gradle?**

Gradle build scripts are declarative, readable, and clearly express their intention. Writing code in Groovy instead of XML, sprinkled with Gradle's build-by-convention philosophy, significantly cuts down the size of a build script and is far more readable.

Gradle uses its own DSL(Domain Specific Language) Groovy-based scripts to write build scripts. Unlike Ant and Maven, it does not use complex XML build scripts. Gradle is developed using Java and configuration elements are developed using Groovy. It uses Groovy to write build scripts.

**How to use Gradle plugin in Eclipse?**

**For Eclipse 4.5 Mars 2 IDE**

1. Open Eclipse Marketplace and install “Buildship Gradle Integration for Eclipse”.
2. Create New Project by choosing Gradle Project from pop-up list.
3. Select “Java Quickstart” as Sample Project.
4. Open build.gradle file from directory structure created by Eclipse Gradle plugin.
5. Default, it gives Person.java and PersonTest.java to start with.
6. Right click on Project >> Run As >> Gradle Build…
7. It shows Edit Configuration Wizard, pressing Ctrl + space will list of tasks can be executed.
8. Type build, to build your application with.
9. Refresh the project to see the build file, which can be deployed.

**For Eclipse 4.7 Oxygen IDE**

1. Open Eclipse Marketplace and install “Buildship Gradle Integration for Eclipse”.
2. Create New Project by choosing Gradle Project from pop-up list.
3. For plain java application, add classes and run as Java Application.
4. For web application, add plugins as discussed below.
5. Build the application, it will perform all the required tasks.
6. Add run configuration based server plugin added to build.gradle file.

**The Gradle build file - build.gradle**

A project using Gradle describes its build via a build.gradle file. This file is located in the root folder of the project. The build file for Gradle builds is based on a Domain Specific Language(DSL). In this file, you can use a combination of declarative and imperative statements. You can also write Groovy or Kotlin code, whenever you need it. Tasks can also be created and extended dynamically at runtime.

**Core Components of build.gradle**

1. **Creating task in build.gradle**
2. **Declaring your Dependencies**
3. **Repositories**
4. **Plugins**
5. **buildscript**
6. **sourceSets**
7. **Specifying a Java Version**
8. **Creating a Task in build.gradle**

Gradle builds a script file for handling two things; one is projects and another one is tasks. A task means a piece of work which a build performs. A task might be compiling some classes, creating a JAR, generating Javadoc, or publishing some archives to a repository. Gradle also offers tasks for introspection of Gradle itself, so a Gradle project can be analyzed by using Gradle’s default tasks.

**Creating custom task in gradle**

To create custom task or script, use following example. We can run these tasks using gradle with task name as : Right click on Project Root Directory >> Run As >> Run Configurations… >> Under the Gradle Project option in left pane create new configuration >> Enter task name >> Apply >> Run.

![A screenshot of a cell phone

Description generated with very high confidence]()

build.gradle

task hello {

group 'customtask'

description 'The hello task greets Gradle by saying "Hello Gradle"'

doFirst {

println 'Hello Gradle'

}

doLast {

println 'Bye bye Gradle'

}

}

To execute : gradle –q hello

Output :

:hello

Hello Gradle

Bye bye Gradle

BUILD SUCCESSFUL

Total time: 0.082 secs

**Task Structure**

Gradle has different phases, when working with tasks. First of all, there is a configuration phase, where the code, which is specified directly in a task’s closure, is executed. After the configuration phase, the execution phase then runs the code inside the doFirst or doLast closures of those tasks, which are executed.

1. **Declaring Your Dependencies**

Gradle follows some special syntax to define dependencies. Dependency configuration is nothing but defines a set of dependencies. You can use this feature to declare external dependencies, which you want to download from the web. This defines the following different standard configurations.

Compile − The dependencies required to compile the production source of the project.

Runtime − The dependencies required by the production classes at runtime. By default, also includes the compile time dependencies.

Test Compile − The dependencies required to compile the test source of the project. By default, it includes compiled production classes and the compile time dependencies.

Test Runtime − The dependencies required to run the tests. By default, it includes runtime and test compile dependencies.

dependencies {

compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'

}

1. **Repositories**

While adding external dependencies. Gradle looks for them in a repository. A repository is just a collection of files, organized by group, name and version. We have to define at least one repository explicitly.  Use this code in build.gradle file.

repositories {

mavenCentral()

}

We can use our own repository in build.gradle. i.e., Instead of using local repository, Remote repository can be used (organizational repository). Following code is to define remote maven. Use this code in build.gradle file.

repositories {

maven {

url ' <http://artifactory:8081/artifactory/libs-release>'

}

}

1. **Plugins**

A plugin is nothing but a set of tasks, almost all useful tasks such as compiling tasks, setting domain objects, setting up source files, etc. are handled by plugins.

Types of Plugins :

Script Plugins : Script plugins can be applied from a script on the local filesystem or at a remote location. Filesystem locations are relative to the project directory, while remote script locations specifies HTTP URL. Look at the following code snippet. It is used to apply the base.gradle plugin to the build script. Use this code in build.gradle file.

apply from: 'base.gradle'

Binary Plugins : Each plugin is identified by a plugin id. Some core plugins use short names to apply it and some community plugins use fully qualified name for plugin id. Sometimes it allows to specify a class of plugin. It shows how to apply Java plugin by using its type. Use this code in build.gradle file.

apply plugin: Java

Standard & Integration Gradle Plugins

These plugins add support for various languages which can be compiled and executed in the JVM.

apply plugin: java The Java plug-in provides tasks to compile Java source code, run unit tests, create Javadoc and create a JAR file.

apply plugin: groovy Adds support for building Groovy projects.

apply plugin: jetty Deploys your web application to a Jetty web container embedded in the build.

apply plugin: war Adds support for assembling web application WAR files.

apply plugin: eclipse Generates files that are used by Eclipse IDE, thus making it possible to import the project into Eclipse.

1. **buildscript**

buildscript specifies repositories to look for in case of applied plugin in build script (build.gradle is often called build script). Gradle fetches required jars from buildscript provided repositories. If your build script needs to use external libraries, you can add them to the script's classpath in the build script itself. You do this using the buildscript() method, passing in a closure which declares the build script classpath.

buildscript {

repositories {

jcenter()

}

dependencies {

classpath 'com.bmuschko:gradle-tomcat-plugin:2.0'

}

}

1. **sourceSets**

SourceSets can be used to specify a different project structure. For example, the sources are stored in a src folder rather than in src/main/java. Look at the following directory structure.

sourceSets {

main {

java {

srcDir 'src'

}

}

test {

java {

srcDir 'test'

}

}

}

We can achieve the above setting as below also. Here, groovy files are in src/main/java folder. To notify gradle to compile theses file as well, add the following script in build.gradle.

sourceSets.main.java.srcDirs = []

sourceSets.main.groovy.srcDirs += ["src/main/java"]

1. **Specifying a Java Version**

Usually, a Java project has a version and a target JRE on which it is compiled. The target compatibility, version & source compatibility property can be set in the build.gradle file.

version = 1.0.0

sourceCompatibility = 1.8

targetCompatibility = 1.8

If the artifact is an executable Java application, the MANIFEST.MF file must be aware of the class with the main method.

apply plugin: 'java'

jar {

manifest {

attributes 'Main-Class': 'com.zetainteractive.foundation.securityserver.SecurityApp'

}

}

**Executing Multiple Tasks**

You can execute multiple tasks from a single build file. Gradle can handle that build file using gradle command. This command will compile each task in the order that they are listed and execute each task along with the dependencies using different options.

Example − There are four tasks - task1, task2, task3, and task4. Task3 and task4 depends on task 1and task2. In the above four tasks are dependent on each other, represented with an arrow symbol. Take a look at the following code. Copy and paste it into build.gradle file.

task task1 << {

println 'compiling source'

}

task task2(dependsOn: task1) << {

println 'compiling unit tests'

}

task task3(dependsOn: [task1, task2]) << {

println 'running unit tests'

}

task task4(dependsOn: [task1, task3]) << {

println 'building the distribution'

}

**Build a JAVA Project**

First, we have to add Java plugin to the build script because it provides tasks to compile Java source code, run unit tests, create Javadoc and create a JAR file. Use the following line in build.gradle file. apply plugin: 'java'

Whenever you add a plugin to your build, it assume a certain setup of Java project (similar to Maven). Look at the following directory structure.

src/main/java :-- contains the Java source code

src/test/java :-- contains the Java tests

If you follow this setup, the following build file is sufficient to compile, test, and bundle a Java project.

Run this command on command line at root project or as gradle task : build

**Deploying a web Project**

Deploying a web application requires plugins as apply plugin: 'war'. But gradle provide support test the application by addling server plugins also. This makes gradle project deploy on single task execution.

Create a run configuration as tomcatRun as right click on project, Run As >> Run Configurations.. >> Under Gradle Project (left pane), create a new launch configuration >> give tomcatRun as Gradle Tasks: >> click Apply >> click Run. Check the console, application is deployed with configured port and context path.

build.gradle for adding tomcat plugin.

apply plugin: 'com.bmuschko.tomcat'

// dependencies to run on tomcat, are mandatory for tomcat plugin

dependencies {

def tomcatVersion = '7.0.57'

tomcat "org.apache.tomcat.embed:tomcat-embed-core:${tomcatVersion}",

"org.apache.tomcat.embed:tomcat-embed-logging-juli:${tomcatVersion}"

tomcat("org.apache.tomcat.embed:tomcat-embed-jasper:${tomcatVersion}") {

exclude group: 'org.eclipse.jdt.core.compiler', module: 'ecj'

}

}

tomcat {

httpPort = 8090

httpsPort = 8091

}

// context where tomcat is deployed, by defautl localhost:8080/

tomcatRun.contextPath = '/demo\_gradle'

tomcatRunWar.contextPath = '/demo\_gradle'

**Gradle Tasks :**

Gradle also offers tasks for introspection of Gradle itself, so a Gradle project can be analyzed by using Gradle’s default tasks. You can use gradle tasks to list the tasks of a project. This will let you see the tasks that the Java plugin has added to your project.

Build tasks

-----------

assemble - Assembles the outputs of this project.

build - Assembles and tests this project.

classes - Assembles main classes.

clean - Deletes the build directory.

jar - Assembles a jar archive containing the main classes.

war - Generates a war archive with all the compiled classes, the web-app content and the libraries.

testClasses- Assembles test classes.

IDE tasks

---------

cleanEclipse - Cleans all Eclipse files.

cleanEclipseWtp - Cleans Eclipse wtp configuration files.

eclipse - Generates all Eclipse files.

eclipseWtp - Generates Eclipse wtp configuration files.

Web application tasks (if tomcat plugin is provided in build.gradle)

---------------------

tomcatJasper - Runs the JSP compiler and turns JSP pages into Java source.

tomcatRun - Uses your files and deploys them to tomcat

tomcatRunWar - Assembles the webapp into a war and deploys it to Tomcat.

tomcatStop - Stops Tomcat.

**Gradle Project Build structure**

1. **Jar for Java console application [ task : gradle jar / gradle build ]**

If gradle project is simple java console application, then running jar task which in turn perform all the basic tasks described above results in creating jar file for that project. We can use separate task build which will do the same job irrespective of type of application we have (console & web).

Project Root Directory

---------------------

...

...

build

|-- classes [Contains .class files in respective packages]

|-- libs

|-- project-name.jar [Generated jar file for the project]

|-- reports

|-- tests

|-- index.html [Reports web page, if any]

|-- test-results [Test results in xml, if any]

|-- tmp

...

1. **war for Java web application [ task : gradle war / gradle build ]**

If gradle project is a java web application, then running war task which in turn perform all the basic tasks and create war file for that project under build directory. We can use separate task build which will do the same job and performs ide task also if plugins are added to build.gradle file.

Project Root Directory

---------------------

...

...

build

|-- classes [Contains .class files in respective packages]

|-- libs

|-- demo\_gradle.war [Generated war file, to be deployed to any server]

|-- resources

|-- main

|-- (resource files) [Contains project all resources file used]

|-- tmp

...

We can modify the war file name by using war.baseName script in build.gradle file. As

war.baseName = 'demo\_gradle'

***For Reference:***

<https://docs.gradle.org/current/userguide/pt03.html>

<http://www.vogella.com/tutorials/Gradle/article.html#running-a-build>