What is a Build Tool?

Before you start with **Gradle**, it will be helpful to get comfortable with build tools first.

Build tools are programs that automate the process of turning up source code into executable applications.

Example: Ant, Maven, Gradle, and NAnt

They are responsible for:

* **Compiling code resources**
* **Managing dependencies​**
* **Generating documentation**
* **Running test scenarios**
* **Publishing the application**

Gradle is a **declarative build tool** for Java projects.

It combines the best features of other build tools.

Gradle helps to automate:

* **Compiling**
* **Testing**
* **Packaging**
* **Deployment of software or projects**.

Android Studio, Eclipse IDE, ​IntelliJ IDEA, and NetBeans use Gradle build system.

##### Gradle - Features

Gradle is an open-source project, and it is licensed under the Apache Software License (ASL). Following are some of the features:

* Utilizes a **Domain Specific Language (DSL)** based on Groovy or kotlin to declare builds.
* With **Deep AP**, Gradle allows to monitor and configure the build script’s execution behavior.
* Manages **dependencies** and **Structures** the build.
* Supports Ant tasks and projects.
* Uses Maven and Ivy repositories to publish or fetch dependencies.
* Supports incremental builds by executing only the necessary tasks in a build.
* Increases productivity.
* Supports multi-project builds.
* Extends easy migration of builds.
* The Gradle Wrapper allows to execute Gradle builds on machines where Gradle is not installed.
* Gradle build files are less verbose as they are written in **Groovy**, while others are in XML .
* **Stronger dependency management** than others ,especially its ability to dynamically replace project dependencies with external ones and vise versa

The following features are exclusively for Gradle :

* Supports **incremental builds**
* Provides **compiler daemon** which compiling faster
* Provides **Incremental compilations for Java classes**
* Supports **Compile avoidance for Java**
* Provides **version conflict resolution**

Gradle's build scripts are written in Groovy, not XML. Do you want to know why?

Groovy:

* Provides **greater transparency** for Java people.
* The **base syntax, type system, and package structure** of Grovvy are the same as Java.
* Overcomes all **Java limitations**
* **Reduces the size of a build script**
* **Groovy** is far more readable
* The **reduced line of code** implies a reduction in time.

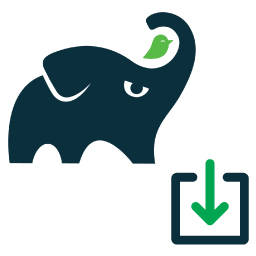
##### Gradle in Local Machine - Prerequisite

To setup Gradle on your local machine, ensure that Java JDK or JRE, version 7 or higher is installed on the computer. To check, run the subsequent command on the command line:

java -version

Gradle carries its Groovy library. Therefore, **Groovy does not require to be installed explicitly**. If Groovy is installed, Gradle ignores that.

Download and Install Gradle



* Download a Gradle distribution from the Gradle portal -

<https://gradle.org/install/>.

* Extract the zip file to the desired folder.
* Set Environment variables – JAVA\_HOME pointing to JDK, GRADLE\_HOME pointing to Gradle. Add GRADLE\_HOME/bin to PATH environment variable.
* To check if Gradle is correctly installed, type gradle -v. The output shows the Gradle version and also the local environment configuration (Groovy, JVM version, and OS).

##### Gradle Playground

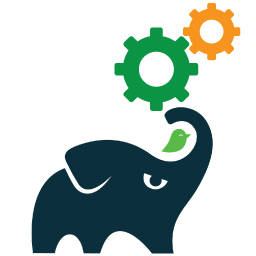
In case, you do not have Gradle installed on your system; you could make use of the playground [Katacoda](https://www.katacoda.com/courses/ubuntu/playground) and try by clicking Start Scenario. You may have to sign-up for accessing this playground.

After launching the Ubuntu playground, follow the commands mentioned below in the interactive terminal:

* sudo apt-get update
* sudo apt-get install Gradle

For checking the Gradle installation, use gradle -version command.

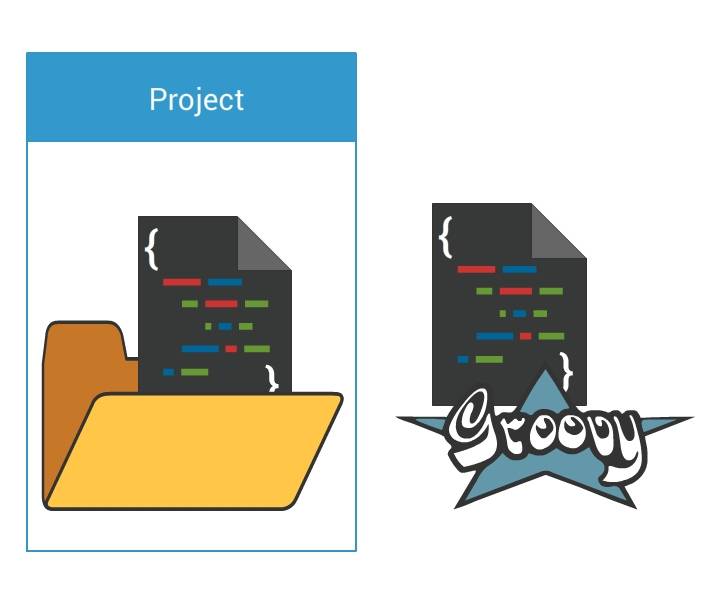
Gradle Configuration Files



The Gradle build has the following threeconfiguration files:

* The Gradle build script (build.gradle) specifies a project and its tasks.
* The Gradle properties file (gradle.properties) is used to configure the properties of the build.
* The Gradle Settings file (settings.gradle) is optional in a build which has only one project. If Gradle build has more than one project, it is necessary as it explains which projects engage to the build. Each multi-project build should include a settings file in the project hierarchy's root project.

build.gradle



* build.gradle is the file located in the root folder of the project.
* This file is the build script for the project.
* It defines a project and its tasks.
* This can be written in Groovy or Kotlin.

##### Custom Tasks

Let's see how to execute a simple task which prints 'Hello World'.

In the project folder, create a file **build.gradle** and add below code :

task hello {

doLast {

println 'Hello world!'

}

}

With this code, we define a hello task. The task prints the words "Hello world!" to the console.

In the command line, type – gradle -q hello for checking the output.

Output of **gradle -q hello**

> gradle -q hello

Hello world!

##### Custom Task in Action

You may use gradle installed locally in your machine or use [Katacoda plaground](https://www.katacoda.com/courses/ubuntu/playground) to carry out hands-on exercises from here on.

For checking the gradle installation use gradle -version command.

Now, create **build.gradle** file using cat command : cat > build.gradle

Enter the file contents and then press Ctrl + C to exit and return to the command prompt

Type gradle -q <name\_of\_task> for checking the output.

**build.gradle**

task hello {

doLast {

String text = 'Hello world!'

println "Original: " + text

println "Upper case: " + text.toUpperCase()

}

}

Output of **gradle -q hello**

> gradle -q hello

Original: Hello world!

Upper case: HELLO WORLD!

Skipping Tasks

Gradle provides several ways to skip the execution of a task.

* onlyIf() method is used to attach a predicate to a task.
* The task’s actions are executed only if the predicate evaluates to true.
* The predicate is implemented as a closure.

**build.gradle**

task hello {

doLast {

println 'hello world'

}

}

hello.onlyIf { !project.hasProperty('skip\_hello') }

Output of **gradle hello -Pskip\_hello**

>gradle hello -Pskip\_hello

:hello SKIPPED

Task Dependencies

* Declare tasks that rely on different tasks.
* The tasks could be on the same project or different projects.

**build.gradle**

task hello {

doLast {

String text = 'Hello world!'

println "Original: " + text

println "Upper case: " + text.toUpperCase()

}

}

task greeting(dependsOn: hello) {

doLast {

println 'This is executed after hello!'

}

}

* Here, **greeting** task depends on **hello**.
* **hello** task executes first and then **greeting**.

Output of **gradle -q greeting** :

> gradle -q greeting

Original: Hello world!

Upper case: HELLO WORLD!

This is executed after hello!

* Tasks also depend on tasks that could not be defined (yet).

##### Task Properties

New properties can be added to a task. These properties can be read and set like a predefined task property.

In the following sample, to add a property named myProperty, set ext.myProperty to a value.

**Example: Adding extra properties to a task**

task hello{

ext.myProperty = "Hello world!"

}

task greeting{

doLast {

println hello.myProperty

}

}

**Output of gradle -q greeting**

> gradle -q greeting

Hello world!

##### Default Tasks

Gradle allows defining one or more default tasks that are executed if no other tasks are specified.

To set the default tasks, use the method defaultTasks. In the following build script, the tasks **hello** and **greeting** are made default tasks:

**Defining a default task:**

defaultTasks 'hello', 'greeting'

task hello{

doLast {

println 'Default Hello!'

}

}

task greeting{

doLast {

println 'Default Greeting!'

}

}

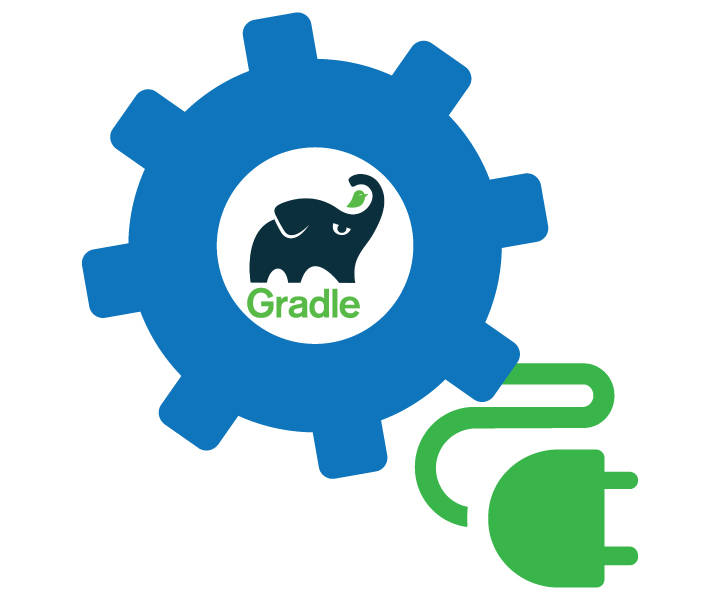
**Output of gradle -q**

> gradle -q

Default Hello!

Default Greeting!

Gradle Plugins



* A Gradle plugin works as an **extension to Gradle**. It extends the project's capabilities when applied to a project.
* Gradle comes with several plug-ins, and you can create custom plug-ins. *Example:* **Java plug-in** adds tasks to a project that permits to create a JAR file, run unit tests and compile Java source code.
* A plug-in is included in a build.gradle file with the apply plugin:'pluginname'statement.

**Example**: The entry apply plugin:'com.android.application' makes the Android plug-in available for a Gradle build.

* Gradle also provides a registry for plug-ins via <https://plugins.gradle.org/>

**Applying plugins**

* Plugins are applied using the Project.apply() method.
* The same plugin can be applied multiple times.

Types of Plugin

**Script plugins**

* They are additional build scripts.
* They follow a declarative approach to manipulate the build.
* They are applied from a script on the local filesystem or at a remote location.

**Example: Applying a script plugin**

apply from: 'other.gradle'

**Binary plugins**

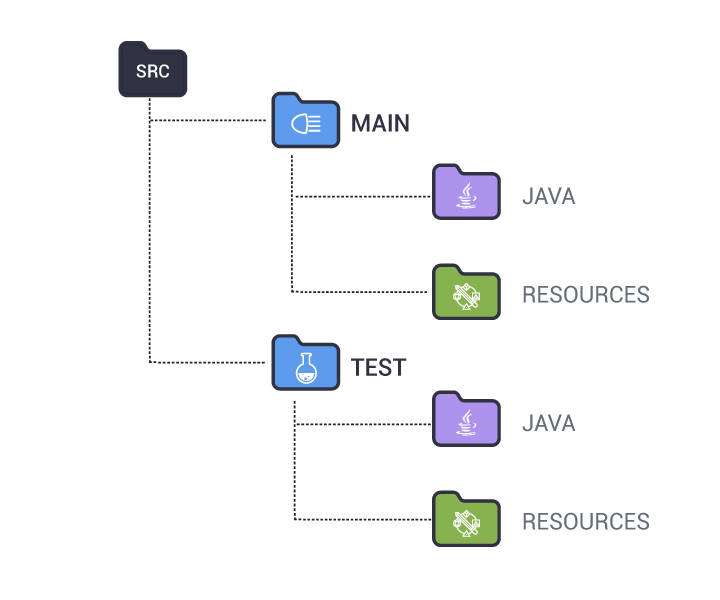
* They are classes that implement the plugin interface.
* They follow a programmatic approach to manipulate the build.
* They occupy within a build script/project hierarchy/externally in a plugin jar.

**Example: Applying a binary plugin**

apply plugin: 'java'

* Applied using a unique plugin id (in the above case - 'java')

##### Source Sets



A **source set** is considered as a group of source files (Java source files and resource files) that can be compiled and executed together.

The Java plugin defines two standard source sets - **main** and **test**.

* src/main/java contains the *Java production source code*.
* src/test/java contains the *Java test source code*.
* Non-Java source files that are included in the JAR file are placed in src/main/resources.
* Non-Java source files needed for testing are placed in src/test/resources.
* To start the build, type gradle build on the command line.
* The figure given here is the output while running the command gradle build.
* build is one of the tasks the Java plugin adds to the project. Here, some tasks are marked with the message UP-TO-DATE, which means that the task has been skipped.
* The build task compiles the code, runs tests, and assembles the JAR file, in the correct order.
* Once the build is successful, open the project directory, and you will see a build folder created with the following contents:

- \*\*classes\*\* - contains the compiled classes.

- \*\*libs\*\* - contains the assembled JAR file.

- \*\*tmp\*\* - contains temporary generated files, such as a manifest file.

##### Customizing the Project

**Changing properties and adding a JAR header**

The subsequent code snippet shows the way to include a header attribute to the JAR manifest and to configure default values in the build script.

**Customization of MANIFEST.MF**

version = '1.0'

sourceCompatibility = 1.7

jar {

manifest {

attributes 'Main-Class': 'HelloGradle'

}

}

* A version number for the project is specified, and the Java source compatibility is indicated.
* The jar task automatically adds a manifest file to the JAR file it creates.
* Jar task’s manifest property is used to add new entries to the manifest file. Here, a Main-Class header is added to jar file's manifest.

**Changing the project default layout**

sourceSets {

main {

java.srcDirs=['src']

}

}

* In the above code, we have told Gradle where the source code is.
* The Location of source code is at the root level, in the source directory (src -name of directory).

##### Configuring and Using External Dependencies

**Defining external dependencies for the build script**

dependencies {

compile 'com.google.code.gson:gson:2.8.0'

}

Here, we have told Gradle, there is a **dependency**.

* The dependency for the compile configuration is defined.
* There are three parts for the name of dependency:

name of the organization that created the dependency: name of the dependency itself:version number

**Defining the repository**

repositories {

mavenCentral()

}

Here, we have told Gradle, **the location of the repositories** for the dependencies.

##### Publishing the JAR file

In Gradle, JAR files are published to repositories.

In the example below, JAR file is published to a local directory.

It can also be published to a remote location or multiple locations.

**Example: Publishing the JAR file**

uploadArchives {

repositories {

flatDir {

dirs 'repos'

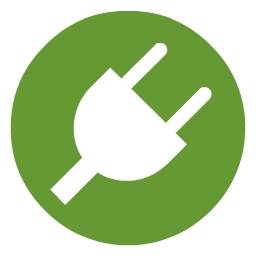
}

}

}

To publish the JAR file, run gradle uploadArchives.

Build Init Plugin



* The Gradle Build Init plugin is used to:
  + create brand new projects of different types
  + convert existing builds to Gradle builds.
* It is an automatically applied plugin.
* To use the plugin, execute the task named init, where the Gradle build is to be created. Without new parameters, this task creates a Gradle project that comprises settings.gradle file, gradle wrapper files, and build.gradle.

init with --type parameter

* The init supports different build setup types.
* The type is specified by supplying a --type argument value.

Example: To create a Java library project, execute:

gradle init --type java-library.

* If a --type parameter is not supplied, Gradle will attempt to infer the type from the environment.
* If the type cannot be inferred, the type “basic” will be used.

The **java-application** build init type must be explicitly specified.

**Features:**

* Uses the **application** plugin to produce a command-line application implemented using Java.
* Uses the **jcenter** dependency repository.
* Uses **JUnit** for testing.
* Features **directories** in the general locations for source code.
* Contains a **sample class and unit test**, if there are no existing source or test files.

To use a different test framework, execute:

gradle init --type java-application --test-framework testing`

##### java-library

The **java library** must be explicitly specified.

**Features:**

* Uses the **java** plugin to produce a library Jar.
* Uses the **jcenter** dependency repository.
* Uses **JUnit** for testing.
* Has **directories** in the conventional locations for source code.
* Contains a **sample class and unit test**, if there are no existing source or test files.

To use a different test framework, execute:

gradle init --type java-library --test-framework testing

##### Creating an Eclipse Project

Another plugin should be added to the build file to create Eclipse-specific descriptor files (.project).

**Example: Eclipse plugin**

apply plugin: 'eclipse'

Here, Gradle eclipse command is executed to generate Eclipse project files.

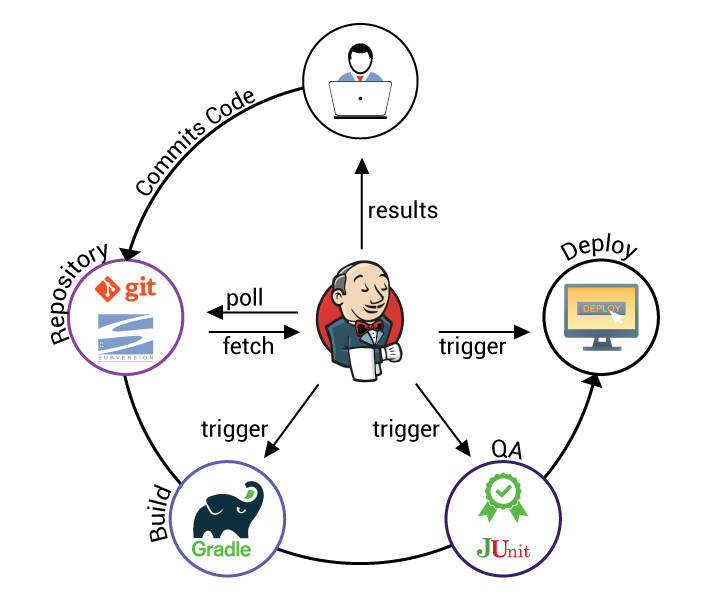
Jenkins



* One of the favorite open-source **continuous integration** tool.
* Helps to automate software build.
* Performs software build using build tools such as Maven, Gradle, and Ant.
* Offers a variety of plugins supporting build, deployment, and automation of projects.
* Supports different repositories like SVN, Git, and Subversion.

You can visit the course [Continuous Integration with Jenkins](https://play.fresco.me/course/162) for a better understanding.

Workflow in Jenkins



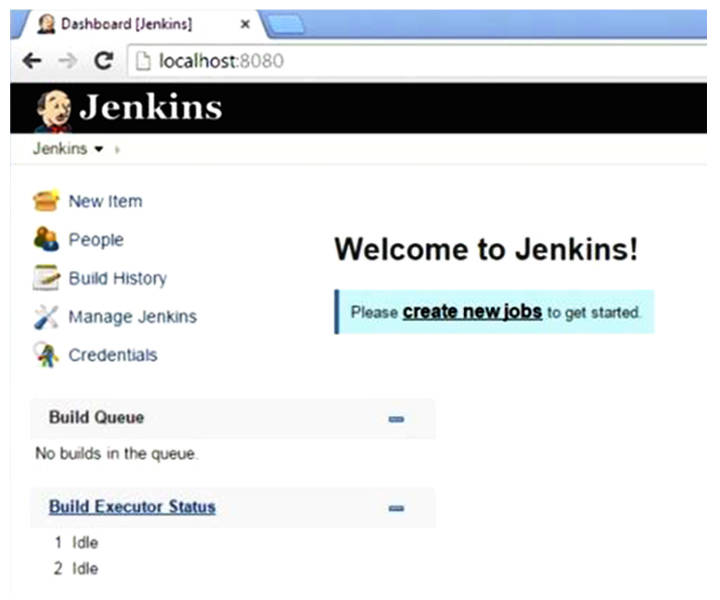
* Developer commits the code into the repository.
* Jenkins detects the changes.
* Triggers to build the source via Gradle.
* Initiates quality checks.
* Prepares to deploy.

Jenkins Installation



* Download jenkins.war from <http://jenkins-ci.org/>.
* After the download, it can be deployed in a container such as Tomcat, or via the command line with java -jar jenkins\*.war.
* If it is started locally, open a browser and type the following URL: http://localhost:8080/ and Jenkins welcome page will be displayed.

Jenkins Configuration



Configure the following plugins for Jenkins:

* Gradle Plugin
* Git Plugin (required if Git is used as a repository)

The Jenkins welcome page is shown above.

* Click **Manage Jenkins** on the left-hand side vertical menu.
* A list of different categories will be displayed.
* Click **Manage Plugins**. Four tabs will be displayed.
* Go to the **Available** tab and filter (top right) for **Gradle Plugin**.
* Select the plugin and click **Download now** and install it.

This adds Gradle build execution capability to the Jenkins server.

* Next, configure JDK, Gradle, and Git with Jenkins.

To configure these settings:

- Open Jenkins URL.

- Click \*\*Manage Jenkins\*\* and click Configure System.

- Enter the path to JDK and save the settings.

##### Create the First Gradle Build Job

Jobs are the unit of execution in Jenkins. A build job can perform **compilation**, **run automated tests**, **package and deployment related tasks**.

**Steps**:

* Go to Jenkins home page and click **Create new jobs**.
* Choose a category of jobs (Freestyle project) that can be created in Jenkins.
* Type a name for the project and then click OK.

In the next page, the following details have to be configured:

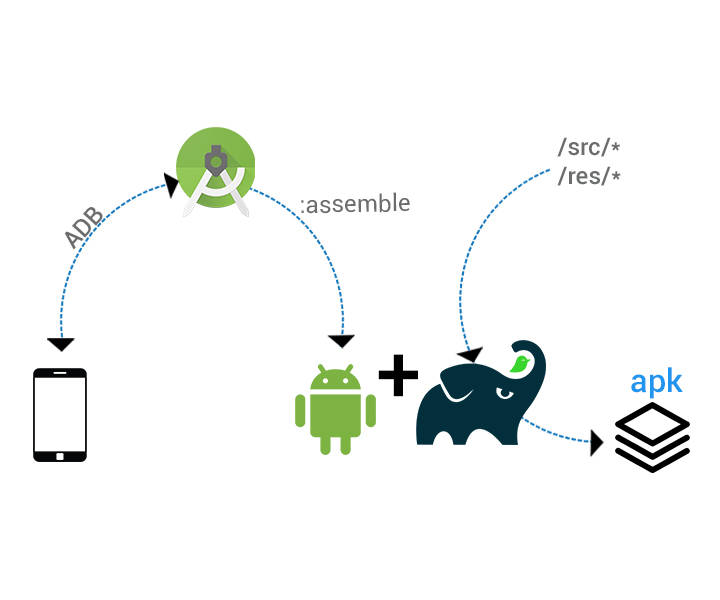
* **Source code management location** to download the project.
* **Build step** for the project.
* **Schedule** the Build task (daily, hourly, after every commit, etc.).
* Put in any **post build action** to perform.

After saving the configurations, the project will be displayed on the dashboard.

Execute Job

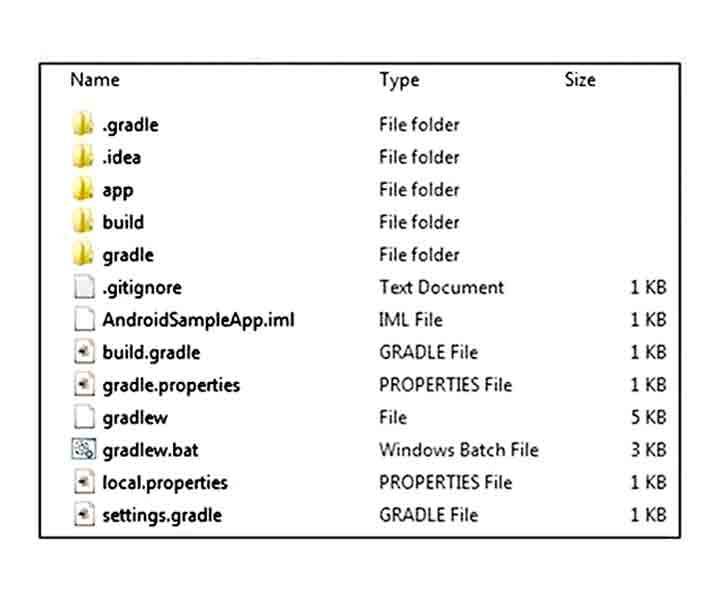
* Click the <name\_of\_project> job on the Jenkins home page. It gets navigated to the job console at http://localhost:8080/job/name\_of\_project/.
* On the job console, click the **Build Now** option (left-hand side) to execute the job manually.

Linking with Android Studio



* Gradle is regarded as the build tool for **Android Studio**.
* Android Studio assigns the entire process of building android apps to Gradle.
* Gradle doesn't know anything about Android. Thus **Android Gradle plugin** is leveraged to bridge the gap.
* Gradle offers to create variants of apps easily like,
  + debug and release builds
  + paid versus free versions

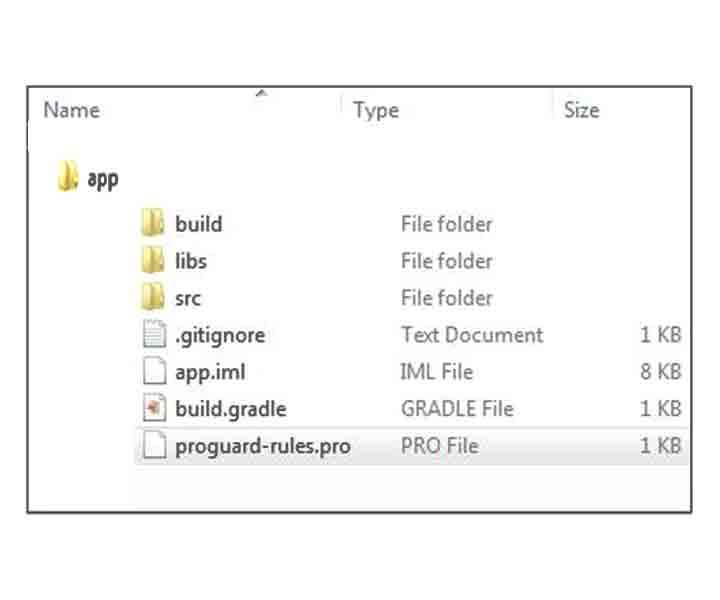
Creating Android Project



When the Android Studio creates a project, the **directory structure** should look like the figure given.

* In the project home directory, there are build.gradleand settings.gradle files. i.e., Android Studio has created a multi-project build structure.
* Android Studio creates :
  + one build.gradle for the parent project
  + individual build.gradle files for the subprojects.
  + creates settings.gradle file that includes all the subprojects.
* Android Studio also adds **Gradle Wrapper**. Thus, Android project can be built on a machine where Gradle is not installed.

##### Creating Android Project



The actual Android application is in the app directory, and its content is shown in the figure.

**Source directory**: src/main/java

**Test directory**: src/androidTest/java

##### Building Android Project with Gradle

Android Studio has automatically generated two build files for the project:

* one in the **root folder** of the project
* other in the **app directory**.

The build.gradle file of the app folder is used to build the Android application.

This build.gradle file has the following content:

apply plugin: 'com.android.application'

android {

compileSdkVersion 22

buildToolsVersion "22.0.1"

defaultConfig {

applicationId "ch10.androidsampleapp"

minSdkVersion 15

targetSdkVersion 22

versionCode 1

versionName "1.0"

}

buildTypes {

release {

minifyEnabled false

proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

}

}

}

dependencies {

compile fileTree(dir: 'libs', include: ['\*.jar'])

compile 'com.android.support:appcompat-v7:22.1.1'

}

With these configurations, the application is ready to get build with Gradle.

buildTypes

* The **buildTypes** configuration is used to define types or environments of build, such as debug, release, QA.
* By default, both the debug and release versions of the Android project are in the build/outputs/apk directory.

You can customize both build and release build types and also extend the build types by adding your own build types, as follows:

buildTypes {

release {

minifyEnabled false

proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

}

staging.initWith(buildTypes.release)

staging {

debuggable true

}

}

Here, one more build type **staging** is added and configured it to be a copy of the release build type and added **debuggable true**.

##### Sample Build File

Let's have a glimpse of build file **build.gradle** :

apply plugin: 'java'

apply plugin: 'eclipse'

sourceCompatibility = 1.7

version = '1.0'

jar {

manifest {

attributes 'Implementation-Title': 'Gradle Quickstart',

'Implementation-Version': version

}

}

repositories {

mavenCentral()

}

dependencies {

compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'

testCompile group: 'junit', name: 'junit', version: '4.+'

}

test {

systemProperties 'property': 'value'

}

uploadArchives {

repositories {

flatDir {

dirs 'repos'

}

}

}

##### Gradle Course Summary

**Congrats! You have completed your learning on the Gradle basics.**

In this course, you read about one of the most popular Build Automation System - **Gradle**and covered in-depth about:

* How to install Gradle
* Create Gradle script
* Gradle plugins especially Java plugins
* Build init plugins
* Integrating Jenkins with Gradle
* Building Android apps with Gradle