Gradle Plugins

Gradle at its core intentionally provides very little for real world automation. All of the useful features, like the ability to compile Java code, are added by plugins. Plugins add new tasks (e.g. JavaCompile), domain objects (e.g. SourceSet), conventions (e.g. Java source is located at src/main/java) as well as extending core objects and objects from other plugins.

A **Gradle plugin** is an extension to Gradle which configures your project in some way, typically by adding some pre-configured tasks which together do something useful.

Applying a plugin to a project allows the plugin to extend the project's capabilities. It can do things such as:

- Extend the Gradle model (e.g. add new DSL elements that can be configured)
- Configure the project according to conventions (e.g. add new tasks or configure sensible defaults)
- Apply specific configuration (e.g. add organizational repositories or enforce standards)

By applying plugins, rather than adding logic to the project build script, we can reap a number of benefits. Applying plugins:

- Promotes reuse and reduces the overhead of maintaining similar logic across multiple projects
- Allows a higher degree of modularization, enhancing comprehensibility and organization
- Encapsulates imperative logic and allows build scripts to be as declarative as possible

Using plugins

To use the build logic encapsulated in a plugin, Gradle needs to perform two steps. First, it needs to resolve the plugin, and then it needs to apply the plugin to the target, usually a Project.

Resolving a plugin means finding the correct version of the jar which contains a given plugin and adding it the script classpath. Once a plugin is resolved, its API can be used in a build script. Script plugins are self-resolving in that they are resolved from the specific file path or URL provided when applying them. Core binary plugins provided as part of the Gradle distribution are automatically resolved.

Applying a plugin is achieved using `apply plugin: 'PLUGIN_NAME'` in the Project you want to enhance with the plugin. Applying plugins is idempotent.

That is, you can safely apply any plugin multiple times without side effects. Find information here for using the plugins DSL for applying plugins.

Java Plugin

The simplest build script for a Java project applies the :books Java plugin and optionally sets the project version and Java compatibility versions:

```
apply plugin: 'java'

sourceCompatibility = '1.8'
targetCompatibility = '1.8'
version = '1.2.1'
```

By applying the Java Plugin, you get a whole host of features:

- A `compileJava` task that compiles all the Java source files under src/main/java
- A `compileTestJava` task for source files under src/test/java
- A `test` task that runs the tests from src/test/java
- A `jar` task that packages the main compiled classes and resources from `src/main/resources` into a single JAR named `<project>-<version>.jar`

A `javadoc` task that generates Javadoc for the main classes.

Source sets

Gradle's Java support was the first to introduce a new concept for building source-based projects: **source sets**. The main idea is that source files and resources are often logically grouped by type, such as application code, unit tests and integration tests. Each logical group typically has its own sets of file dependencies, classpaths, and more. Significantly, the files that form a source set don't have to be located in the same directory!

Source sets are a powerful concept that tie together several aspects of compilation:

- the source files and where they're located
- the compilation classpath, including any required dependencies (via Gradle configurations)
- where the compiled class files are placed

The Java plugin assumes the project layout shown below. None of these directories need to exist or have anything in them. The Java plugin will compile whatever it finds, and handles anything which is missing.

- `src/main/java`: Production Java source, which is compiled and assembled into a JAR.

- `src/main/resources`: Production resources, such as XML and properties files.
- `src/test/java`: Test Java source, which is compiled and executed using JUnit or TestNG. These are typically unit tests, but you can include any test in this source set as long as they all share the same compilation and runtime classpaths.
- `src/test/resources`: Test resources.

Tasks

Check the tasks provided by the Java plugin.

Groovy Plugin

The :books: Groovy plugin extends the Java plugin to add support for Groovy projects. It can deal with Groovy code, mixed Groovy and Java code, and even pure Java code (although we don't necessarily recommend to use it for the latter). The plugin supports joint compilation, which allows you to freely mix and match Groovy and Java code, with dependencies in both directions. For example, a Groovy class can extend a Java class that in turn extends a Groovy class. This makes it possible to use the best language for the job, and to rewrite any class in the other language if needed.

To apply it use: `apply plugin: 'groovy'`

Eclipse Plugin

With the :books: Eclipse plugin you can configure your project so that it can be imported in Eclipse using Gradle. For example, we are going to use this plugin for configuring some the projects that we are going to use so that you can import them and start working straightaway. Keep in mind that you also need to learn how to develop them from scratch.

Usage

```
apply plugin: 'eclipse'
```

Tasks

:books: The most relevant tasks are: * `eclipse`: Generates all Eclipse configuration files * `cleanEclipse`: Removes all Eclipse configuration files

:books: Configuration of Eclipse plugin. We are going to focus on configuring Eclipse project properties using the properties of the `EclipseProject` domain object shown below:

```
apply plugin: 'eclipse'

eclipse {
  project {
    //if you don't like the name Gradle has chosen
    name = 'someBetterName'

    //if you want to specify the Eclipse project's comment
    comment = 'Very interesting top secret project'

    //if you want to append some extra referenced projects in a declarative fashion:
    referencedProjects 'someProject', 'someOtherProject'
    //if you want to assign referenced projects
    referencedProjects = ['someProject'] as Set

    //if you want to append some extra natures in a declarative fashion:
    natures 'some.extra.eclipse.nature', 'some.another.interesting.nature'
    //if you want to assign natures in a groovy fashion:
    natures = ['some.extra.eclipse.nature', 'some.another.interesting.nature']

    //if you want to append some extra build command:
    buildCommand 'buildThisLovelyProject'
    //if you want to append a build command with parameters:
    buildCommand 'buildItWithTheArguments', argumentOne: "I'm first", argumentTwo: "I'm seco

    //if you don't want any node_modules folder to appear in Eclipse, you can filter it out
    resourceFilter {
      appliesTo = 'FOLDERS'
      type = 'EXCLUDE_ALL'
      matcher {
        id = 'org.eclipse.ui.ide.multiFilter'
        arguments = '1.0-name-matches-false-false-node_modules'
      }
    }
  }
}
```

For information about the different project natures that can be used explore this tutorial or simply us the ones that we will propose in the module.

Application Plugin

The :books: application plugin facilitates creating an executable JVM application. It makes it easy to start the application locally during development, and to package the application as a TAR and/or ZIP including operating system specific start scripts.

Applying the Application plugin also implicitly applies the Java plugin. The main source set is effectively the "application".

Applying the Application plugin also implicitly applies the Distribution plugin. A main distribution is created that packages up the application, including code dependencies and generated start scripts.

Usage

To use the application plugin, include the following in your build script:

```
apply plugin: 'application'
```

The only mandatory configuration for the plugin is the specification of the main class (i.e. entry point) of the application.

```
mainClassName = "org.gradle.sample.Main"
```

You can run the application by executing the `run` task (type: JavaExec). This will compile the main source set, and launch a new JVM with its classes (along with all runtime dependencies) as the classpath and using the specified main class. You can launch the application in debug mode with gradle run `--debug-jvm` (see JavaExec.setDebug(boolean)).

Since Gradle 4.9, the command line arguments can be passed with `--args`. For example, if you want to launch the application with command line arguments foo `--bar`, you can use gradle run `--args="foo --bar"` (see `JavaExec.setArgsString(java.lang.String)`.

If your application requires a specific set of JVM settings or system properties, you can configure the applicationDefaultJvmArgs property. These JVM arguments are applied to the run task and also considered in the generated start scripts of your distribution.

Example: Configure default JVM settings

```
applicationDefaultJvmArgs = ["-Dgreeting.language=en"]
```

Documentation

- Create a Java Application (Gradle guide)
- Building Java projects

- Groovy quickstart
- Gradle plugins
- Java quickstart
- Application plugin: to run Java apps on the console
- Groovy plugin
- Eclipse plugins

References

References