Writing Gradle Plugins

Table of Contents

This guide walks you through the process of creating reusable build logic in a Gradle plugin, which can then be applied to other Gradle builds. The core of Gradle provides an infrastructure for building anything, but plugins are what allow build users to get things done with a minimum of effort. They can apply conventions, add new task types, integrate with third-party tools and libraries, and more. This guide focuses on the most basic of plugins, but it represents the tip of the iceberg.

What you'll build

You'll write a simple plugin that adds a new task type and creates a task of that type in whatever project its applied to. You will also prove that the plugin works and see its effect by applying the plugin to a build.

What you'll need

- About 10 minutes
- A text editor or IDE
- A Java Development Kit (http://www.oracle.com/technetwork/java/javase/downloads/index.html) (JDK), version 1.8 or better

- A Gradle distribution (https://gradle.org/install), version 3.4.1 or better

Create a project

You'll need to create a directory for this plugin project and switch to it:

```
$ mkdir greeting-plugin
$ cd greeting-plugin
```

Next, you will create a special directory structure for the plugin code:

```
$ mkdir -p buildSrc/src/main/java/org/example/greeting
```

If you'd like to learn more about this `buildSrc` directory, you can read about how it allows you to organize your build logic in the User Manual (https://docs.gradle.org/3.4.1/userguide/organizing_build_logic.html#sec:build_sources).

You will often want to publish the plugin so that it can be reused across multiple builds. You can learn about the options to do just that in Next steps.

Create the plugin

Create the class `GreetingPlugin` in the directory you just created, setting its contents to the following:

buildSrc/src/main/java/org/example/greeting/GreetingPlugin.java

JAVA

```java
package org.example.greeting;

import org.gradle.api.Plugin;
import org.gradle.api.Project;

public class GreetingPlugin implements Plugin<Project> {
    public void apply(Project project) {
        project.getTasks().create("hello", Greeting.class, (task) -> {  1
            task.setMessage("Hello");
            task.setRecipient("World");                                  2
        });
    }
}
```

1 Creates a new task named `hello` of type `Greeting` (which you will define shortly)

2 Sets default values for the new task

This is the actual plugin and the Gradle `Project` object is your access point for the entire Gradle API, which allows you to do the same things as you can do in a build script. In this case, you are creating a simple task called `hello` in the target project.

Use the Gradle DSL Reference (https://docs.gradle.org/3.4.1/dsl/) and {javadocs}[Javadocs] to learn what you can do with the Gradle API. Start with the entry for `Project` (https://docs.gradle.org/3.4.1/dsl/org.gradle.api.Project.html). You can find out more about what you can achieve by also following the links in Next steps.

Next, you'll create the class for the task type that the plugin is using. Add a new `Greeting` class in the same package as the plugin:

buildSrc/src/main/java/org/example/greeting/Greeting.java

JAVA

```java
package org.example.greeting;

import org.gradle.api.DefaultTask;
import org.gradle.api.tasks.TaskAction;

public class Greeting extends DefaultTask {
    private String message;
    private String recipient;

    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }

    public String getRecipient() { return recipient; }
    public void setRecipient(String recipient) { this.recipient = recipient; }

    @TaskAction
    void sayGreeting() {
        System.out.printf("%s, %s!\n", getMessage(), getRecipient());  1
    }
}
```

1 Prints out the configured greeting when the task runs

You can learn more about creating your own task types in the User Manual (https://docs.gradle.org/3.4.1/userguide/custom_tasks.html).

You now have a plugin, but a plugin alone doesn't do anything. You need to apply it to a project using the `apply` keyword for it to be useful, so we will show you how to do that next.

Use the plugin in the main project

Create a `build.gradle` file in the root of the project with the following contents:

build.gradle

```
apply plugin: org.example.greeting.GreetingPlugin  1
```

GROOVY

1 This applies your plugin to the current `Project` instance, adding the `hello` task to the build.

The above syntax is all that's required because the plugin source resides in the `buildSrc` directory. Applying other, non-core Gradle plugins requires other syntax as described in the User Manual (https://docs.gradle.org/3.4.1/userguide/plugins.html#sec:binary_plugins).

You can now verify that your plugin is working by running its `hello` task in the main build:

```
$ gradle hello
:buildSrc:compileJava
:buildSrc:compileGroovy UP-TO-DATE
:buildSrc:processResources UP-TO-DATE
:buildSrc:classes
:buildSrc:jar
:buildSrc:assemble
:buildSrc:compileTestJava UP-TO-DATE
:buildSrc:compileTestGroovy UP-TO-DATE
:buildSrc:processTestResources UP-TO-DATE
:buildSrc:testClasses UP-TO-DATE
:buildSrc:test UP-TO-DATE
:buildSrc:check UP-TO-DATE
:buildSrc:build
:hello
Hello, World!
```

The bulk of the output reflects that the files in `buildSrc` are treated as a Java project, which needs to be built first. Once that happens, the classes inside that project become available in your main build and the main build can execute the task or tasks that you specified.

Your build is currently just using the default property values for the greeting, hence why it prints out "Hello, World!". This doesn't have to be the case as you can configure the task directly in the build script:

build.gradle

GROOVY

```
apply plugin: org.example.greeting.GreetingPlugin

hello {  1
    message = "Hi"
    recipient = "Gradle"
}
```

1 Configures multiple properties of the task named `hello`

You can learn more about the syntax for configuring tasks in the User Manual (https://docs.gradle.org/3.4.1/userguide/more_about_tasks.html#sec:configuring_tasks).

Now when you run the `hello` task — using `-q` to hide the `buildSrc` output this time — you'll see the following:

```
$ gradle -q hello
Hi, Gradle!
```

Your plugin is now functionally complete and you've seen it in action in the above build. There is just one more thing we want to show you, which helps make the build script a bit tidier and also helps when it comes to publishing your plugin: adding a plugin identifier.

Declare a plugin identifier

In most cases, you apply plugins using an ID because they are easier to remember than fully-qualified class names. They also result in tidier build files. So it makes sense to ensure that your own plugin can also be applied in the same way, which is why you will now declare an identifier for the plugin.

Create the following properties file:

buildSrc/src/main/resources/META-INF/gradle-plugins/org.example.greeting.properties

```
implementation-class=org.example.greeting.GreetingPlugin
```

Gradle uses this file to determine which class implements the `Plugin` interface. The name of this properties file excluding the `.properties` extension becomes the identifier of the plugin.

You must put the properties file in the directory `META-INF/gradle-plugins` as Gradle will try to resolve the file from that specific location in the plugin JAR.

That's all you need to do in your plugin, so now you can replace the following line of the build script:

```
apply plugin: org.example.greeting.GreetingPlugin
```
GROOVY

with one that uses the plugin ID:

```
apply plugin: "org.example.greeting"
```
GROOVY

Note how the name of the properties file — `org.example.greeting.properties` — maps to the ID above.

Always qualify the plugin name with a namespace that is unique to you instead of the "org.example" used in this guide. Doing so helps avoid name clashes between plugins. You can find more details about plugin IDs in the User Manual (https://docs.gradle.org/3.4.1/userguide/custom_plugins.html#sec:creating_a_plugin_id).

Summary

You're now done! You have successfully created a plugin and used it within a build. Along the way, you've learned how to:

- Put build logic into a plugin
- Use the `buildSrc` directory for a plugin's classes
- Give the plugin an ID and apply it in a build script

This guide focuses on the essence of what a plugin is, but most plugins are far more substantial in the features that they provide. The next section will guide you towards learning more about what plugins can do and how you should implement them.

Next steps

Now that you're familiar with the basics of building Gradle plugins, you may be interested in:

- Simplifying plugin development with the Java Gradle Plugin Development Plugin (https://docs.gradle.org/3.4.1/userguide/javaGradle_plugin.html)
- Publishing plugins to the Gradle Plugin Portal (https://plugins.gradle.org/docs/submit)
- Modeling your domain with extensions (https://docs.gradle.org/3.4.1/userguide/custom_plugins.html#sec:getting_input_from_the_build)
- Testing plugins (https://docs.gradle.org/3.4.1/userguide/test_kit.html)
- Adding incremental build support to new task types (https://docs.gradle.org/3.4.1/userguide/more_about_tasks.html#sec:up_to_date_checks)

Last updated 2017-03-22 08:23:30 UTC