Gradle Guides (https://guides.gradle.org)

Building JVM Libraries

build passing

(https://travis-ci.org/gradle-guides/building-jvm-libraries)

Table of Contents

What you'll build
What you'll need
Create a library project
Assemble the library JAR
Customize the library JAR
Consume the library JAR
Summary
Next steps

This guide walks you through the process of using Gradle to produce a JVM library suitable for consumption by other JVM libraries and applications.

What you'll build

You'll start by creating a Java library project consisting of a single class. You'll then use Gradle to compile and package the library into a *Java Archive* (JAR (https://docs.oracle.com/javase/8/docs/technotes/guides/jar/index.html)) that contains the compiled class and a default manifest file. Next, you'll configure your Gradle build script to customize the name of the JAR and the contents of its manifest. Finally, you'll verify that the library JAR is well-formed by consuming it from a separate command-line application.

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 1.7 or better
- A Gradle distribution (https://docs.gradle.org/3.3/userguide//installation.html#sec:download), version 3.3 or better

Create a library project

Create a new project directory named `mylib` and initialize the directory structure that will contain your library's sources, for example by running the following commands on a Unix-like system:

SHELL

```
$ mkdir mylib
$ cd mylib
$ mkdir -p src/main/java/org/example/mylib
```

Now create a class named `Greeter` as follows:

src/main/java/org/example/mylib/Greeter.java

JAVA

```
package org.example.mylib;

public class Greeter {
    private String name;

    public Greeter(String name) { this.name = name; }

    public String getGreeting() { return "Hello, " + this.name + "!"; }
}
```

You now have the necessary components for a simple Java library project. The next step is to create a Gradle build script capable of compiling and packaging the library into a JAR. To do so, add a file named `build.gradle` in the root of the project and apply Gradle's Java plugin (https://docs.gradle.org/3.3/userguide/java_plugin.html) as follows:

build.gradle

GROOVY

```
apply plugin: 'java'
```

Once you've saved the file, you're ready to assemble your library JAR.

Assemble the library JAR

The Java plugin adds a number of tasks (https://docs.gradle.org/3.3/userguide/java_plugin.html#sec:java_tasks) to your project, including a `jar` task. To assemble your library JAR, run `gradle jar`, like so:

```
$ gradle jar
:compileJava
:processResources UP-TO-DATE
:classes
:jar

BUILD SUCCESSFUL
```

The output above tells you that running the `jar` task causes Gradle to:

1. compile the Java source code
2. process any resource files
3. package everything up into a JAR file

> All of this happens without any additional configuration in `build.gradle` because Gradle's Java plugin assumes your project sources are arranged in a conventional project layout (https://docs.gradle.org/3.3/userguide/java_plugin.html#sec:java_project_layout). You can customize the project layout if you wish as described in the user manual (https://docs.gradle.org/3.3/userguide/java_plugin.html#sec:changing_java_project_layout).

You can find your newly packaged JAR file in the `build/libs` directory with the name `mylib.jar`. Verify that the archive is valid by running the following command:

```
$ jar tf build/libs/mylib.jar
META-INF/
META-INF/MANIFEST.MF
org/
org/example/
org/example/Greeter.class
```

You should see the required manifest file— `MANIFEST.MF` —and the compiled `Greeter` class.

At this point, your library is ready to be consumed by other JVM-based projects, but there are a couple of customizations you may want to make first. We'll look at those next.

Customize the library JAR

You will often want the name of the JAR file to include the library *version*. This is easily achieved by setting a top-level `version` property in the build script, like so:

build.gradle

GROOVY
```
apply plugin: 'java'

version = '0.1.0'
```

Now run the `jar` task once more:

```
$ gradle jar
```

and notice that the resulting JAR file at `build/libs/mylib-0.1.0.jar` contains the version as expected.

Another common requirement is customizing the manifest file, typically by adding one or more attributes. Let's include the library name and version in the manifest file by configuring the `jar` task (https://docs.gradle.org/3.3/userguide/more_about_tasks.html#sec:configuring_tasks). Add the following to the end of your build script:

build.gradle

GROOVY
```
jar {
    manifest {
        attributes('Implementation-Title': project.name,
                   'Implementation-Version': project.version)
    }
}
```

To confirm that these changes work as expected, run the `jar` task again, and this time also unpack the manifest file from the JAR:

```
$ gradle jar
$ jar xf build/libs/mylib-0.1.0.jar META-INF/MANIFEST.MF
```

Now view the contents of the `META-INF/MANIFEST.MF` file and you should see the following:

META-INF/MANIFEST.MF

MF

```
Manifest-Version: 1.0
Implementation-Title: mylib
Implementation-Version: 0.1.0
```

Learn more about configuring JARs

The `manifest` is just one of many properties that can be configured on the `jar` task. For a complete list, see the Jar section (https://docs.gradle.org/3.3/dsl/org.gradle.api.tasks.bundling.Jar.html) of the Gradle Language Reference (https://docs.gradle.org/3.3/dsl/) as well as the Jar (https://docs.gradle.org/3.3/userguide/java_plugin.html#sec:jar) and Creating Archives (https://docs.gradle.org/3.3/userguide/working_with_files.html#sec:archives) sections of the Gradle User Manual (https://docs.gradle.org/3.3/userguide/).

Now you can complete this exercise by trying to compile some Java code that uses the library you just built.

Consume the library JAR

Create a new Java file in the root of the project called `Main.java` and put the following code in it:

Main.java

JAVA

```java
import org.example.mylib.Greeter;

public class Main {
    public static void main(String... args) {
        System.out.println(new Greeter("Gradle").getGreeting());
    }
}
```

If you now try to compile this file, you'll get the following error:

```
$ javac Main.java
Main.java:1: error: package org.example.mylib does not exist
import org.example.mylib.Greeter;
                        ^
Main.java:5: error: cannot find symbol
        System.out.println(new Greeter("Gradle").getGreeting());
                               ^
  symbol:   class Greeter
  location: class Main
2 errors
```

Let's fix that quickly by including our JAR file on the compilation classpath:

```
$ javac -cp .:build/libs/mylib-0.1.0.jar Main.java
```

Finally, run the application to test everything is working:

```
$ java -cp .:build/libs/mylib-0.1.0.jar Main
Hello, Gradle!
```

Summary

That's it! You've now successfully built a Java library project, packaged it as a JAR and consumed it within a separate application. Along the way, you've learned how to:

- Apply Gradle's Java plugin
- Run the Java plugin's `jar` task and examine its output
- Customize the name of a JAR file and the content of its manifest

Next steps

Building a library is just one aspect of reusing code across project boundaries. From here, you may be interested in:

- Consuming JVM libraries (https://docs.gradle.org/3.3/userguide/artifact_dependencies_tutorial.html)
- Publishing JVM libraries (https://docs.gradle.org/3.3/userguide/artifact_management.html)
- Working with multi-project builds (https://docs.gradle.org/3.3/userguide/intro_multi_project_builds.html)

Last updated 2017-02-13 21:51:00 UTC