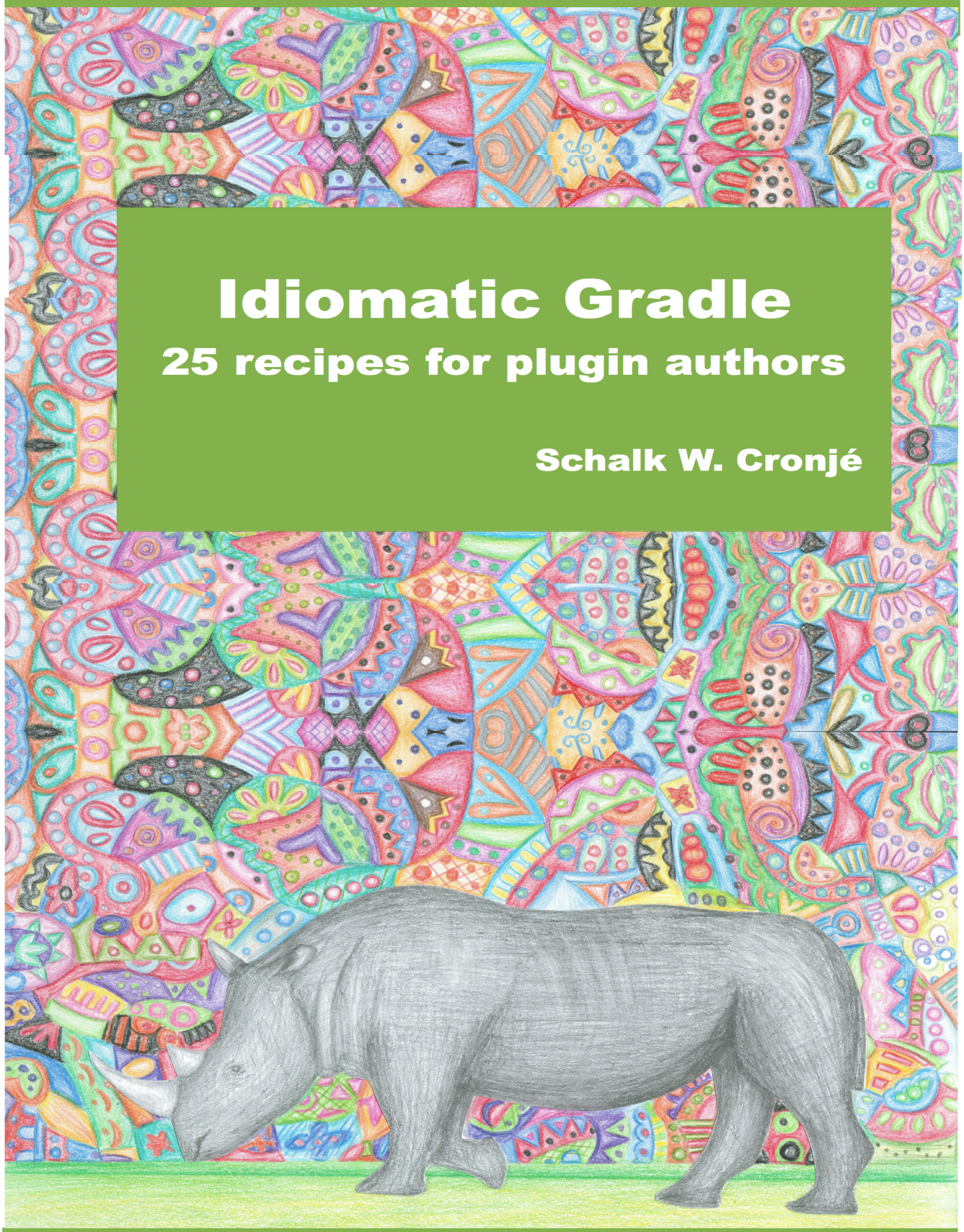
Idiomatic Gradle

25 recipes for plugin authors

Schalk W. Cronjé



Idiomatic Gradle Plugins

25 Recipes for Authors

Schalk Cronjé

This book is for sale at http://leanpub.com/idiomaticgradle

This version was published on 2017-11-15

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Schalk Cronjé by spreading the word about this book on Twitter!

The suggested hashtag for this book is #idiomaticGradle.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#idiomaticGradle

Contents

Avoiding Groovy Version Mismatch

Summary

As part of plugin development, it is highly probable that a plugin uses Groovy as the implementation or testing language. (As mentioned before, all recipes in this book assume Groovy as the implementation language). When relying on other dependencies that has Groovy as a transitive dependency, the build can fail due to a Groovy version mismatch.

Solution

Exclude `groovy-all` for the dependency that is adding it as a transitive dependency. If the plugin is part of a multi-project build, then exclude `groovy-all` for all configurations

Examples

When using Spock Framework[1] for unit testing, the version of Groovy that is resolved is usually different from that of `localGroovy()`. Exclude `groovy-all` for Spock.

Customising transitive dependencies

```
1 testCompile ('org.spockframework:spock-core:1.0-groovy-2.0') {
2     exclude module : 'groovy-all'
3 }
```

The Groovy VFS Gradle Plugin is built as part of a multi-project build. The main artifact it relies upon, is in another subproject called `groovy-vfs` and this is a pure Groovy implementation. As `groovy-vfs` is an independent distributed jar, it is compatible with a range of Groovy versions. When consumed by the plugin, resolving transitive dependencies will cause a version mismatch. In such a case exclude `groovy-all` for all configurations.

[1] http://docs.spockframework.org/en/latest

Gradle plugins in multi-project builds

```
1  configurations.all {
2    exclude module : 'groovy-all'
3  }
4
5  dependencies {
6    compile project (':otherProject')
7    compile localGroovy()
8  }
```

Line #2: Exclude `groovy-all` from all configurations. When run, the correct `groovy-all` jar will be made available on the classpath due to `localGroovy()`

Gradle API Updates

As from Gradle 2.8 the version of the bundled Groovy has moved to 2.4.4 (from 2.3.10 as opposed to Gradle 2.7). This results in the Spock Framework version being of the incorrect version. Although it is recommended to build plugins with Gradle 2.0, there are cases where a plugin might rely on APIs in later Gradle releases. A plugin developer might also choose to make it as easy to build and test a plugin with a later version without having to tweak the build script.

Covering different versions of Spock Framework

```
1  ext {
2      spockGroovyVer = GroovySystem.version.replaceAll(/\.\d+$/,'')
3  }
4
5  dependencies {
6    testCompile ("org.spockframework:spock-core:1.0-groovy-${spockGroovyVer}") {
7        exclude module : 'groovy-all'
8    }
9  }
```

References

- Peter Niederweser on excluding Groovy dependencies
- Cedric Champeau on obtaining Groovy Version

Collection of Files

Summary

Many tasks need one ore more properties that need to be a collection of files. It is important to keep the configuration readable and easy to use by script authors.

Solution

Store the configuration entity as a list of `Object` in the task class. Convert it to files only when accessed. This will allow for lazy evaluation when needed, replacement of list content and appending of more files to the list.

Examples

Assume for the moment that a task has a list of input sources files called `documents`.

Task class code snippet for file collections

```
@InputFiles
FileCollection getDocuments() {
    project.files(this.documents)
}

void setDocuments(Object... docs) {
    this.documents.clear()
    this.documents.addAll(docs as List)
}

void documents(Object... docs) {
    this.documents.addAll(docs as List)
}

private List<Object> documents = []
```

Line #1: Create a getter and annotate with `@InputFiles` or `@OutputFiles`. The purpose of the getter is to translate upon access to a `FileCollection` object.

Line #3: Translate from the list of `Object` using the built-in `project.files` method. This handles a large variety of types including files, strings and closures as well as lists and arrays thereof.

Line #6: Use a setter to allow for `setDocuments 'foo','bar'` replacement of current content with a new set of content. This becomes very useful should another plugin author decide to extend your task type.

Line #11: Use a method with the name of the property to allow for a expressive `documents 'foo','bar'` style.

Line #15: The property is left private as appropriate access is already provided.

Usage in build script

```
1  // Assuming we have a task called 'documentBinder'
2  documentBinder {
3
4      setDocuments '/path/to/doc', new File('/path/to/other/doc')
5
6      documents '/path/to/doc', new File(' / path / to / other / doc ')
7
8      documents project.file('add/other/doc')
9
10     documents { '/even/add/from/closure' }
11 }
```

Line #4: Clear any existing document list, and replace with the given list. List can contain anything that `project.files` can convert to `File` objects.

Line #6: Append more documents to existing list

Line #10: Even closures can be used to allow for late evaluation of documents

Collection of Strings

Summary

The use of string collections as task properties is quite common. It is important to keep the configuration readable and easy to use by script authors. The use of `toString()` by script authors will lead to less readable (and ugly) build scripts and as such plugin authors, should attempt to handle conversion from a variety of class types to `String` objects behind the scenes.

Solution

Store the configuration entity as a list of `Object`. Convert it to `String` objects only when accessed. Allow for replacement of list content or appending to the list. If required, also allow for lazy evaluation only at the point of task execution.

Examples

The `JRubyExec` task type in the JRuby Gradle plugin allows the script author to provide a list of arguments that can be passed to a Ruby script when run from Gradle. In order to make it easy for authors, these arguments can be provided as strings, objects convertible to strings or even closures.

Task class code snippet for string collections

```
1  @Input
2  List<String> getScriptArgs() {
3      CollectionUtils.stringize(
4          this.scriptArgs.collect {  it ->
5              it instanceof Closure ? (it as Closure).call() : it
6          } .flatten()
7      )
8  }
9
10 void setScriptArgs(Object... args) {
11     this.scriptArgs.clear()
12     this.scriptArgs.addAll(args as List)
13 }
14
15 void scriptArgs(Object... args) {
16     this.scriptArgs.addAll(args as List)
17 }
18
19 private List<Object> scriptArgs = []
```

Line #1: Create a getter and annotate with `@Input` or `@Output`. The purpose of the getter is to translate upon access to a collection of `String` objects.

Line #2: Collections usually are `Set` or `List`.

Line #3: Translate from the list of `Object` using the built-in `org.gradle.util.CollectionUtils.stringize` method. This handles a collection containing a large variety of types including files and strings, but not closures.

Line #5: Add special cases for handling closures.

Line #6: Flatten out embedded collections.

Line #10: Use a setter to allow for `setScriptArgs 'foo','bar'` replacement of current content with a new set of content. This becomes very useful should another plugin author decide to extend your task type or when a task is modified via an extension.

Line #15: Use a method with the name of the property to allow for a expressive `scriptArgs 'foo','bar'` style.

Line #19: The property is left private as appropriate access is already provided.

Configuration snippet

```
1 // Assuming we have a task called 'runMyScript'
2 runMyScript  {
3     setScriptArgs '--file', new File('/path/to/other/doc')
4
5     scriptArgs '--output', new File('/path/to/other/doc')
6     scriptArgs "${{->delayedString}}"
7     scriptArgs { 'string in closure' }
8     scriptArgs { ['list in closure','with multiple elements'] }
9 }
```

Line #3: Clear any existing arguments list, and replace with the given list. List can contain any number of items that can be converted to a string.

Line #5: Append more arguments to the existing list

Line #6: A `GString` containing a closure returning a single string

Line #7: A closure returning a single string

Line #8: A closure returning a list of strings

Property Maps

Summary

Another tool or system that is being wrapped by a Gradle plugin might need a list of free-form properties passed unto it. The underlying system might not perform any further validation, also ignoring anything that is not applicable.

Solution

The Groovy language already provides for an easy declaration of property maps and this feasture can be used as is within the task configuration DSL. Declare the property map as a private member and add getter, replacement (via setter) and append methods. Keeping the property map as private prevents accidental assignment from the configuration closure as this could be confusing to script authors or consumers.

Examples

The Asciidoctor Gradle plugin needs to pass a set of attributes to the underlying Asciidoctor engine. Attributes that are not required by the Asciidoctor engine will be silently ignored. This makes the application of a property map a very good fit, as superfluous attributes can be stored, but will not cause a runtime error.

Task class code snippet

```
1  @Input
2  Map getAttributes() {
3      this.attrs
4  }
5
6  void setAttributes(Map m) {
7      this.attrs=m
8  }
9
10 void attributes(Map m) {
11     this.attrs+=m
12 }
13
14 private Map attrs = [:]
```

Line #1: Annotate the getter as opposed to the property.

Line #6: Use the setter to replace one property map with another

Line #10: Use the basename to insert more properties into the existing map

Line #14: Keep the property private (or use `@PackageScope`) to keep it from accidental usage within the configuration closure.

Usage of property map in Asciidoctor plugin

```
1  ext {
2      predefined = [  doctype : 'book' ]
3  }
4
5  asciidoctor {
6      setAttributes toclevel : '3', revnumber : '1.0'
7
8      attributes 'source-highlighter': 'coderay'
9
10     attributes toc : 1, toclevel : '2'
11
12     attributes predefined
13
14 }
```

Line #6: All existing properties can be removed and replaced with a new set

Line #8: Properties can be past one per line. (aids readability and loops)

Line #10: Multiple properties can be passed per call

Line #12: Other property maps can be passed and will be merged.

Allow user to override specific version of underlying in-process library

Summary

In many cases a plugin consumer does want to be locked down to one version of a dependency. A plugin author might thus want to allow the user to override the default dependency with different version.

Solution

Create a project extension where the version can be set. Add an `afterEvaluate` closure, which in turn will add the dependency to the appropriate configuration group at the appropriate time. Optionally a classloader can be used to load the class when the task is executed.

Examples

The Asciidoctor Gradle plugin relies on a specific version of Asciidoctorj. When a new version of Asciidoctorj comes out, there might be a delay before the plugin is released. Alternatively the plugin author might want to experiment with development versions of Asciidoctorj before release.

The first step is to create an extension whereby the script author can set the version. Even though this could be handled directly by the script author using a `dependencies` block, the use of an extension provides clearer intent and reduces misunderstanding as to which dependency should be used.

Create extension

```
class AsciidoctorJExtension {
    String version = '1.5.0' //

    AsciidoctorJExtension(Project proj) {
        project=proj
    }

    @PackageScope
    Project project
}
```

Line #2: Set the default version as to the one that would be recommended to be used by the plugin author.

Once the extension is created, it can be added when the plugin is applied.

Add extension in plugin

```
1  void apply(Project project) {
2      project.extensions.create('asciidoctorj',AsciidoctorJExtension,project)
3      project.configurations.maybeCreate( 'internal_asciidoctorj' )
4
5      project.afterEvaluate {
6          project.dependencies {
7              internal_asciidoctorj "org.asciidoctor:asciidoctorj:${project.asciid\
8  octorj.version}"
9          }
10     }
11 }
```

Line #2: Create an extension called `asciidoctorj`.

Line #3: Create a configuration called `internal_asciidoctorj`.

Line #7: Add the dependency to the `internal_asciidoctorj` at the end of the configuration phase.

Noteworthy in the previous code snippet is the use of the interpolated `GString` to det the dependency. As it is within the closure added to `afterEvaluate` it be only be evaluated at the end of the configuration phase, by which time the correct version will already have been set.

When the plugin is published, the script author will simply be able to override the version of `asciidoctorj` by doing as below.

Setting the version in build.gradle

```
1  asciidoctorj {
2      version = '1.5.2'
3  }
```

Line #2: User can now override the version of the library.

In some cases it might be necessary to lazy load the object as well. In this case a custom classloader is utilised. The use of term classloader can induce fear in those not-so-Java developers, but there is no need for angst when loading it as part of a plugin. Loading can be accomplished by a little bit of code within the task action.

Use a custom classloader

```
1 def urls = project.configurations.internal_asciidoctorj.files.collect { it.toURI\
2 ().toURL() }
3 def classLoader = new URLClassLoader(urls as URL[], Thread.currentThread().conte\
4 xtClassLoader)
5 def asciidoctorInstance = classLoader.loadClass('org.asciidoctor.Asciidoctor$Fac\
6 tory')
```

Line #1: Get all of the files in the `internal_asciidoctorj` configuration that was created when the plugin was applied.

Line #2: Create the classloader for all those files. For simplicity the class loader that is currently in context is used (i.e. the context within which the task action is executed within).

Line #3: Load the class that is required. One or more classes can be loaded if needed.

Gradle API Updates

A new way of handling default dependencies was introduced in Gradle 2.5 in the form of defaultDependency[2]. This removes the need for an `afterEvaluate` closure and instead allows for a closure to be called when a configuration is first resolved and no specific dependency was provided elsewhere. This approach does not necessarily provide a shorter code form, but it does provide a standardised way of dealing with dependencies going forward.

[2] https://docs.gradle.org/2.5/release-notes#simpler-default-dependencies

Alternative plugin approach for Gradle 2.5+

```
1  void apply(Project project) {
2      project.with {
3          def asciidoctorj = extensions.create('asciidoctorj',AsciidoctorJExtensio\
4  n,project)
5          def conf = configurations.maybeCreate( 'internal_asciidoctorj' )
6          conf.defaultDependencies { deps ->
7              deps.add( project.dependencies.create(
8                  "org.asciidoctor:asciidoctorj:${asciidoctorj.version}")
9              )
10         }
11     }
12 }
```

Line #3: Create extension as before

Line #4: Create configuration as before

Line #5: Call `defaultDependencies on the configuration. The closure will be passed the` DependencySet‘ from the configuration.

Line #7: Use the `create` call on the project's `DependencyHandler` object to create a dependency that is not associated with a configuration and then add it to the provided `DependencySet`.

A plugin author wishing to advantage of the new functionality on offer, but still want to maintain one codebase with the largest possible version compatibility can resort to some minute Groovy metaprogramming.

Allow plugin to select functionality automatically

```
1  void apply(Project project) {
2      project.extensions.create('asciidoctorj',AsciidoctorJExtension,project)
3      def conf = project.configurations.maybeCreate( 'internal_asciidoctorj' )
4
5      if( conf.respondsTo('defaultDependencies')) {
6          conf.defaultDependencies { deps ->
7              deps.add( project.dependencies.create(
8                  "org.asciidoctor:asciidoctorj:${project.asciidoctorj.version}")
9              )
10         }
11     } else {
12         project.afterEvaluate {
```

```
13              project.dependencies {
14                  internal_asciidoctorj "org.asciidoctor:asciidoctorj:${project.as\
15  ciidoctorj.version}"
16              }
17          }
18      }
19  }
```

Line #2: Create extension and configuration as before

Line #5: Check whether configuration object support the `defaultsDependencies` method and select appropriate approach.

Caveats

The original implementation relies on the assumption that a script author will mostly not be aware of the internal configuration and therefore not try to add additional items into the configuration. Should the script author decide to explicitly set the Maven coordinates of the dependency against that of the internal configuration name, the probability is good that the newer version of the dependency will win out. It is only if the resolver strategy is explicitly modified by the script author, that outcome might be different.

In this regards, use of the new `defaultDependencies` will introduce a behavioural change. The version that the script author supplied, will be taken over any default version. Should a plugin author like to bring the same behaviour to older versions of Gradle it is will possible within the `afterEvaluate` closure as illustrated by the following code block.

Only add dependency if one does not exist already

```
1   project.afterEvaluate {
2       def hasDep = project.configurations.internal_asciidoctorj.dependencies.find {
3           it.group == 'org.asciidoctor' && it.name == 'asciidoctorj'
4       }
5       if (!hasDep) {
6           project.dependencies {
7               internal_asciidoctorj "org.asciidoctor:asciidoctorj:${project.asciid\
8   octorj.version}"
9           }
10      }
11  }
```

It has to be remembered that this kind of tinkering with internal configurations by script authors are rare and there is very little need to provide safety against it within the plugin. A script author that is in the need of performing such remediation should well be aware of the associated dangers.

A second problem may arise in the use `maybeCreate` when creating the configuration. In contract to the `create` which will emit an exception if the configuration already exists, `maybeCreate` will re-use the existing one. This has the advantage of more than one plugin using the same configuration, or to allow a script author to manipulate the configuration prior to the plugin being applied. Both these cases, however, can also lead to unexpected side-effects. It is possible to modify creation of the configuration as below:

Strict configuration creation

```
1 def conf = project.configurations.create( 'internal_asciidoctorj' )
2 conf.visible = false
```

Line #1: Fail the build should the configuration already exists

Line #2: Restrict the scope of the configuration only to the projects where the plugin is applied in.

This decision which approach to use is left to the plugin author. Most has advantages and disadvantages as explained before. It is suggested that where the second approach is taken, to use a longer configuration name that will have little change of being re-used by another plugin.

References

- A comment on defaultDependencies

Add SourceSet Support for JVM Language

Summary

Many new languages are being created that targets the JVM. Creating a plugin for a JVM language that follows the conventions of core languages such as Java, Groovy & Scala, may contribute to both the popularity of Gradle and the new JVM language.

Solution

Follow conventions for JVM languages such as Groovy & Scala by doing the following:

- Implement a loose-standing source set class for the new JVM language.
- Address joint-compilation if supported by the new JVM language
- Create a base plugin class and configure defaults for source sets to recognise the new language files.
- Follow this up by implementing Add Assemble Task Support for JVM Language.

Examples

Assume that support for a new fictitious JVM language Sprache (the German word for language) will be added. All source files for Sprache will be found in `.sprache` files. Sprache, like Groovy and Scala, will also support joint-compilation with Java files.

An independent source set class is the easiest way to start. Unfortunately the Gradle API does not support all of the necessary functionality in the public API and therefore some internal APIs will be required:

Source set imports

```
1 import org.gradle.api.file.SourceDirectorySet
2 import org.gradle.util.ConfigureUtil
3 import org.gradle.api.internal.file.DefaultSourceDirectorySet
4 import org.gradle.api.internal.file.FileResolver
```

Line #3: It is far easier to re-use `DefaultSourceDirectorySet`, than to perform a full custom implementation of the `SourceDirectorySet` interface. The API for `DefaultSourceDirectorySet` has remained stable throughout Gradle 2.0 - 2.11, but has been changed in Gradle 2.12.

Line #4: `FileResolver` is an internal API which helps Gradle find files and is required by `DefaultSourceDirectorySet`.

Understanding source set behaviour

Very important to notice is that this source set class does not implement the `SourceSet` interface. The latter is something that will be exposed automatically to the build script by Gradle, glue-ing in bits via convention mapping. Nor surprisingly, this is a huge point of confusion for many plugin writers.

`SourceSet` has four getter methods by default from which source is obtained - `getJava`, `getAllJava`, `getResources` and `getAllSource`.

- `getJava` will only return Java source files from the `java` source set.
- `getAllJava` will return Java source files found in the `java` source set as well as any other language source set. For instance if the Groovy plugin was applied, it will also return Java files from the `groovy` source set.
- `getResources` will only return files defined as resources.
- `getAllSource` will return all source files plus all resources.

Things can get more confusing for JVM language plugins. This is best illustrated via the Groovy plugin. When the mentioned plugin is applied, two more source accessors will appear on the `SourceSet` instance, being `getGroovy` and `getAllGroovy`. These two methods exhibit some potentially confusing behaviour:

- `getGroovy` will return both Java & Groovy source files from the `groovy` source set.
- `getAllGroovy` will only return Groovy source files from the `groovy` source set. (!)

> It can argued that the two methods should have been named the other way around. However, this is the state of Gradle today and plugin authors should just be aware of this naming convention.

Most of the work for a source set is done in the constructor. Two directory sets are defined - one that includes only the language-specific files and one that could contain both the former and some additional files.

Source set definition

```
class SpracheSourceSet {

    static final String LANG_NAME = 'sprache'

    SpracheSourceSet(String displayName, FileResolver fileResolver) {
        fullSourceSet = createSourceDirectorySet(
            "${displayName} ${LANG_NAME.capitalize()} source",
            fileResolver
        )
        fullSourceSet.filter.include("**/*.java", "**/*.sprache")

        spracheOnlySourceSet = createSourceDirectorySet(
            "${displayName} ${LANG_NAME.capitalize()} source",
            fileResolver
        )
        spracheOnlySourceSet.source(fullSourceSet)
        spracheOnlySourceSet.filter.include("**/*.sprache")
    }

    private final SourceDirectorySet fullSourceSet
    private final SourceDirectorySet spracheOnlySourceSet
}
```

Line #3: The language name will be used in a number of places as part of convention naming. It is useful to define it once-off.

Line #9: Define a directory set that will return all source files found under a given directory that is of interest to this source set.

Line #10: As Sprache supports joint-compilation, return both Java & Sprache files. If Sprache did not support joint-compilation, then the Java part of the filter could have been left out.

Line #15: Define a directory set that will return only Sprache source files found under a given file tree.

Line #16: Add in all files that were found by the joint-compilation file tree.

Line #17: Filter this file tree to only include Sprache files.

Wrapping internal APIs into seperate methods can be very useful to help with later maintenance should a class or interface change with a new release of Gradle. In the case of creating `DefaultSourceDirectorySet` this version if very simplistic for for Gradle 2.0 - 2.11, but see notes on Gradle API changes at the end of this recipe.

Wrapping DefaultSourceDirectorySet

```
1  private DefaultSourceDirectorySet createSourceDirectorySet(
2      String name,
3      FileResolver fileResolver
4  ) {
5      new DefaultSourceDirectorySet(name,fileResolver)
6  }
```

What remains for the source set definition is to add some getter methods and a single method to allow configuration via closure.

Source set getter and configuration methods

```
1  class SpracheSourceSet {
2      /* Constructor and fields defined previously */
3
4      SourceDirectorySet getSprache() {
5          fullSourceSet
6      }
7
8      SpracheSourceSet sprache(Closure configureClosure) {
9          ConfigureUtil.configure(configureClosure, getSprache())
10         return this
11     }
12
13     SourceDirectorySet getAllSprache() {
14         spracheOnlySourceSet
15     }
16 }
```

Line #4: Return the source set containing all of the Sprache and Java files.

Line #8: Allow for configuration of a source set via closure.

Line #9: Due to the way the directory sets have been set up in the constructor it is only necessary to configure the 'sprache' sourceset. (Calling `allSprache` will simply filter out anything it does not need).

Line #13: Return the source set containing only Sprache files.

Laying out the base plugin

```
1  import org.gradle.api.internal.plugins.DslObject
2  import org.gradle.api.internal.project.ProjectInternal
3  import org.gradle.api.tasks.compile.AbstractCompile
4  import org.gradle.util.GradleVersion
5
6
7  class SpracheBasePlugin implements Plugin<Project> {
8
9      static final String LANG_NAME = SpracheSourceSet.LANG_NAME
10
11     void apply(Project project) {
12         project.with {
13             apply plugin : JavaBasePlugin
14         }
15
16         createSourceSetDefaults(project)
17     }
18 }
```

Line #9: The language name will be used in number of places as part of convention naming. It is useful to define it once-off, but just re-using the one already defined in the source set class.

Line #13: The `JavaBasePlugin` is required for all JVM language implementations. It provides the core support for JVM source sets.

In order to instantiate this source set and attach it to a named global source set, some reliance on internal APIs are required once again.

Internal APIs required for source set creation

```
1  import org.gradle.api.internal.plugins.DslObject
2  import org.gradle.api.internal.project.ProjectInternal
3  import org.gradle.api.tasks.compile.AbstractCompile
4  import org.gradle.util.GradleVersion
```

Creating the base plugin for a JVM language.

```
class SpracheBasePlugin implements Plugin<Project> {
    /* For other code see previous code block */

    void createSourceSetDefaults(Project project) {
        def jpc = project.convention.getPlugin(JavaPluginConvention)
        jpc.sourceSets.all { SourceSet srcSet ->
            final SpracheSourceSet langSourceSet = new SpracheSourceSet(
                "${LANG_NAME.capitalize()} ${srcSet.name}",
                (project as ProjectInternal).fileResolver
            )
            final SourceDirectorySet dirSet = langSourceSet."${LANG_NAME}"

            new DslObject(srcSet).convention.plugins.put(LANG_NAME,langSourceSet)
            dirSet.srcDir("src/${srcSet.name}/${LANG_NAME}")
            srcSet.allJava.source(dirSet)
            srcSet.allSource.source(dirSet)
            srcSet.resources.filter.exclude {
                FileTreeElement fte -> srcSet."${LANG_NAME}".contains(fte.file)
            }
        }
    }
}
```

Line #7: Instantiate the Sprache source set given the provided source set name as well as the Gradle's internal `FileResolver` instance.

Line #13: Glue this source set to the global `SourceSet` container instance known as `sourceSets`.

Line #14: Set a default directory where to find sources, should no other directory be set.

Line #15: Since joint-compilation, tell Gradle to also Java files in this source set.

Line #16: Add all of the Sprache sources to the named global named source set.

Line #17: Ensure Sprache source files do not end up in resources.

The plugin is now ready to have the assemble tasks added. It is also a Gradle convention to create a base plugin for the language which provides all of the functionality to find source files and assemble them. The base plugin does not provide any directory layout conventions. See the Add Source Layout Conventions for JVM Language recipe on how to accomplish this.

Gradle API Updates

The way of adding source sets for a new language is also changing with the new incubating model ([GradleDocs2], [GradleDocs3], [GradleDocs4], [GradleDocs5]).

As a more direct consequence to this recipe, the internal API of the constructors for `DefaultSourceDirectorySet` have changed in Gradle 2.12 adding a third parameter in all cases. The latter is another internal interface being `DirectoryFileTreeFactory` from the `org.gradle.api.internal.file.collectior` package. Luckily it easy to construct an instance of this via `DefaultDirectoryFileTreeFactory` which is in the same internal package. This poses a dilemma for the plugin author, which has to make a decision on whether to release one version of the plugin for earlier versions of Gradle and one for 2.12 and beyond. Luckily there is a way to workaround this through the use of some metaprogramming, even if it results in more code.

Handling API change across multiple Gradle versions

```
@CompileDynamic
private DefaultSourceDirectorySet createSourceDirectorySet(
    String name,
    FileResolver fileResolver
) {
    DefaultSourceDirectorySet.constructors.findResult { ctor ->
        def params = ctor.parameterTypes
        if(params == [String,FileResolver] as Class[]) {
            return ctor.newInstance(name,fileResolver)
        } else if (params == [String,String,FileResolver] as Class[]) {
            return null
        }
        try {
            final String pkgName = 'org.gradle.api.internal.file.collections'
            Class<?> dftfInterface = Class.forName(
                "${pkgName}.DirectoryFileTreeFactory"
            )
            Class<?> fileTreeFactory = Class.forName(
                "${pkgName}.DefaultDirectoryFileTreeFactory"
            )
            if(params == [String,FileResolver,dftfInterface] as Class[] ) {
                return ctor.newInstance(name,fileResolver,fileTreeFactory.
                    newInstance())
            }
        } catch (ClassNotFoundException){}
        null
    }
}
```

Line #8: Look for the pre-Gradle 2.12 constructor

Line #9: Create a new instance of `DefaultSourceDirectorySet`.

Line #11: If the alternative three parameter constructor from pre-Gradle 2.12 is found, ignore it.

Line #17: Try to load the new `DirectoryFileTreeFactory` interface. If it is not found, then this is not a supported Gradle version. 2.0-2.11 is already covered by the previous two conditions, so this also acts as a safeguard to detect API changes beyond 2.12

Line #20: Try to load the `DefaultDirectoryFileTreeFactory` implementation class .

Line #21: Check whether there is a constructor which takes `DirectoryFileTreeFactory` as a parameter. It is important that the interface class is used and not the name of the implementation class.

Line #23: Instantiate `DefaultSourceDirectorySet` using the new three parameter constructor, passing an instance of `DefaultDirectoryFileTreeFactory`.

Line #25: The catch block is required to take care of the earlier `Class.forName` calls. The result is set to null as to indicate that no match was found.

Add the above code now ensure that the plugin can work across all of Gradle 2.0 - 4.3.1. In another API breaking change occurs this code will result in a NPE being emitted. A plugin author might want to wrap the code up into another exception that provides a more meaningful message.

There is also an alternative, potentially shorter and slightly more readable way that has been suggested by Lance Semmens.

Alternative way of handling API change across multiple Gradle versions

```
1  if(GradleVersion.current() < GradleVersion.version('2.12')) {
2      DefaultSourceDirectorySet.getConstructor(String,FileResolver).
3          newInstance(name,fileResolver)
4  } else {
5      final String pkgName = 'org.gradle.api.internal.file.collections'
6      DefaultSourceDirectorySet.getConstructor(
7          String,FileResolver,Class.forName(
8          "${pkgName}.DirectoryFileTreeFactory"
9      )).newInstance(
10         name,
11         fileResolver,
12         Class.forName(
13             "${pkgName}.DefaultDirectoryFileTreeFactory"
14         ).newInstance()
15     )
16 }
```

Line #1: Use `org.gradle.util.GradleVersion` to determine the current Gradle version

Line #3: If earlier than 2.12 just construct if via the old constructor interface, otherwise use a technique as described earlier.

References & Credits

- Dinko Srkoč for suggesting the use of metaprogramming.
- Benjamin Muschko - How `SourceSet` concept is added.
- Lance Semmens for suggesting an alternative way of solving the Gradle 2.12 problem.

Create Safe Filenames From Inputs

Summary

In certain conditions it might be necessary to create file or path names from input data over which the plugin author might not have control. Creating safe filenames in a portable manner can be erroneous and contain traps for the unwary or inexperienced cross-platform developer.

Solution

Use the internal API `org.gradle.internal.FileUtils.toSafeFileName` utility function.

As this is internal it might change in a future version, but at the time of writing it have been the same from Gradle 2.0 - 2.12.

Examples

The Asciidoctor plugin produces outputs in different folders for each of the backends that it supports. Should a new backend be added that may contain a character that is invalid on a specific operating system (for instance a `:` on Windows), that will cause the plugin to fail. By passing the name through `org.gradle.internal.FileUtils.toSafeFileName` an operating system-safe filename will be generated

Using toSafeFileName

```
File backendDirname(final File baseDir, final String backend) {
    new File(baseDir, org.gradle.internal.FileUtils.toSafeFileName(backend))
}
```

Self-referencing plugin

Summary

There are certain plugins that require themselve as part of the build. One common solution is to always use the previous version of the plugin for the development of the new. Unfortunately there are cases where the build process relies on code that exists only in the new (unreleased) plugin version.

Solution

Load the source code directly into the Gradle process via a `GroovyScriptEngine` instance. The tasks from the plugin will be loaded in the same way as if the compiled plugin is loaded in another build script.

Examples

The Unofficial Bintray plugin for Gradle might need the latest functionality in the plugin in order to publish a new version to Bintray. Apply this short code snippet into the `build.gradle` file to ensure that the plugin is loaded directly from the source code instead of a built jar.

Code snippet for build script

```
1 apply plugin: new GroovyScriptEngine(
2         ['src/main/groovy','src/main/resources'].
3             collect{ file(it).absolutePath }
4             .toArray(new String[2]),
5         project.class.classLoader
6 ).loadScriptByName('book/SelfReferencingPlugin.groovy')
```

Line #2: Add each toplevel source folder that will be required to build the plugin

Line #4: Set count to number of items in list defined above

Line #6: Set relative path below source folders where plugin class is to be found.

The previous example works in most cases, but when certain Gradle APIs are used, they are found in JARs that are in the `libs/plugins` folder of the Gradle distribution. In such a case a little bit more needs to be done.

Code snippet for extending the classpath

```
1  def pluginURLs = fileTree ("${gradle.gradleHomeDir}/lib/plugins") { include '*.j\
2  ar' } .files.collect {
3      it.toURI().toURL()
4  }
5  def selfReferencingClassLoader = new URLClassLoader(
6      pluginURLs.toArray(new URL[pluginURLs.size()]),
7      project.gradle.class.classLoader as URLClassLoader
8  )
9
10
11 apply plugin: new GroovyScriptEngine(
12     ['src/main/groovy','src/main/resources'].
13         collect{ file(it).absolutePath }.toArray(new String[2]),
14     selfReferencingClassLoader
15 ).loadScriptByName('book/SelfReferencingPlugin.groovy')
```

Line #3: Add all of the JARs as URLs

Line #7: The classloader used by the `project` instance is an extension of `java.net.URLClassLoader`, so it is just a matter of creating another `URLClassLoader` instance, using the classloader from `project` as the parent and providing URLs to all of the additional URLs

Line #13: Just pass the new classloader here instead of the one from `project`.

Caveats

This wonderful trick only works if the referenced plugin class actually compiles. If it does not, then compilation of the build script will fail and the plugin author will be no-man's land. The recommendation is to wrap the self-referencing code in a conditional block that can be turned off via a command-line property:

Build script safeguard for self-referencing plugins

```
1  if(!project.properties.DISABLE_GRADLETEST) {
2    /* Self-referencing code mentioned earlier go here */
3  }
```

Line #1: Fix a temporary problem by adding `-PDISABLE_GRADLETEST` to the command-line when building.

References

- Original idea from Knut Saua Mathiesen

Bibliography

Discussion Forums

[**GHale1**] Gary Hale. Dynamic dependency version for plugin[3]

[**MErdmann1**] Marcin Erdmann. Generate a Java file and include it in the Sourceset of compilation[4]

[**PNiederwieser1**] Peter Niederwieser. Controlling conflicting versions of Groovy[5]

[**SGreene1**] Stirling Greene. A comment on defaultDependencies[6]

[**CChampeau1**] Cedric Champeau. Groovy Version[7]

[**BMuschko2**] Benjamin Muschko. How to add a sourceSet without using any plugins[8]

[**AOberstar1**] Andrew Oberstar. Custom task with fields - assign directly or via conventionmapping[9]

[**LanceJava1**] Lance Semmens. DefaultSourceDirectorySet alternative[10]

[3] http://discuss.gradle.org/t/dynamic-dependency-version-for-plugin/6691

[4] http://discuss.gradle.org/t/generate-a-java-file-and-include-it-in-the-sourceset-for-compilation/6940

[5] http://discuss.gradle.org/t/controlling-conflicting-versions-of-groovy-for-a-plugin/5877

[6] https://discuss.gradle.org/t/a-comment-on-defaultdependences/12331

[7] https://discuss.gradle.org/t/getting-hold-of-version-of-localgroovy/12612/3

[8] https://discuss.gradle.org/t/how-to-add-a-sourceset-without-using-any-plugins/9510

[9] https://discuss.gradle.org/t/custom-task-with-fields-assign-directly-or-via-conventionmapping/4553

[10] https://discuss.gradle.org/t/defaultsourcedirectoryset-alternative/15193

Software

[**AsciidoctorProject**] Asciidoctor Github Organisation. Asciidoctor Project[11]

[**AsciidoctorJ**] Asciidoctor Github Organisation. Asciidoctorj Project[12]

[**AsciidoctorGradle**] Asciidoctor Github Organisation. Asciidoctor Gradle Plugin[13]

[**JRubyGradle**] JRuby-Gradle Github Organisation. JRuby Gradle Plugin[14]

[**MSBuild**] Microsoft, MSBuild on Github[15]

[**XBuild**] Mono Project. XBuild[16]

[**Spock**] Peter Niederwieser. Spock Framework 1.0[17]

[**DoxygenGradle**] Schalk Cronjé. Doxygen Gradle Plugin[18]

[**GradleTest**] Schalk Cronjé. GradleTest Gradle Plugin[19]

[**GroovyVfs**] Schalk Cronjé. Groovy VFS Gradle Plugin[20]

[**BintrayGradle**] Schalk Cronjé. Unofficial Bintray Gradle Plugin[21]

[**GnuMake**] Schalk Cronjé. GNU Make Gradle Plugin[22]

Other

[**Aalmiray1**] Griffon Github Organisation, 'Griffon Project'. Code snippet[23]

[**GradleDocs1**] Gradleware Inc. Gradle 2.0 User Guide. Writing Custom Plugins[24]

[**GradleDocs2**] Gradleware Inc. Gradle 2.9 Release Notes. Improvements to the incubating model infrastructure[25]

[**GradleDocs3**] Gradleware Inc. Gradle 2.10 Release Notes. DSL improvements for the Software

[11] https://github.com/asciidoctor/asciidoctor
[12] https://github.com/asciidoctor/asciidoctorj
[13] http://plugins.gradle.org/plugin/org.asciidoctor.gradle.asciidoctor
[14] http://plugins.gradle.org/plugin/com.github.jruby-gradle.base
[15] https://github.com/microsoft/msbuild
[16] http://www.mono-project.com/docs/tools+libraries/tools/xbuild/
[17] http://spockframework.github.io/spock/docs/1.0/index.html
[18] https://github.com/ysb33r/Gradle/tree/master/doxygen
[19] https://github.com/ysb33r/GradleTest/tree/master/
[20] https://github.com/ysb33r/groovy-vfs/tree/master/gradle-plugin
[21] https://github.com/ysb33r/bintray
[22] https://github.com/ysb33r/gnumake-gradle-plugin
[23] https://github.com/griffon/griffon/blob/griffon-2.4.0/build.gradle#L85
[24] https://docs.gradle.org/2.0/userguide/custom_plugins.html
[25] https://docs.gradle.org/2.9/release-notes#improvements-to-the-incubating-model-infrastructure

Model[26]

[**GradleDocs4**] Gradleware Inc. Gradle 2.11 Release Notes. Better support for developing plugins with the software model[27]

[**GradleDocs5**] Gradleware Inc. Gradle 2.12 Release Notes. Experimental software model improvements[28]

[**GradleDocsWrapper**] Gradleware Inc. Gradle 2.0 User Guide. Gradle Wrapper[29]

[**GradleTestKit**] Gradleware Inc. Gradle 2.7 User Guide. Gradle TestKit[30]

[**GradleWrapperBug**] Gradleware Inc. Gradle 2.7 Release Notes. Gradle 2.6 Wrapper Bug[31]

[**KHenney1**] Kevlin Henney, The Programmer[32]

[**KMathiesen1**] Knut Saua Mathiesen. Gist[33]

[**BMuschko1**] Benjamin Muschko. Gradle Plugin Best Practices by Example[34]

[**ELezmy**] Eyal Lezmy. Gradle Plugin. Take control of the build[35]. Presentation at Devoxx Belgium 12 November 2015.

[26] https://docs.gradle.org/2.10/release-notes#dsl-improvements-for-the-software-model
[27] https://docs.gradle.org/2.11/release-notes#better-support-for-developing-plugins-with-the-software-model
[28] https://docs.gradle.org/2.12/release-notes#experimental-software-model-improvements
[29] https://docs.gradle.org/2.0/userguide/gradle_wrapper.html
[30] https://docs.gradle.org/2.7/userguide/test_kit.html
[31] https://docs.gradle.org/2.7/release-notes#important:-performance-regression-with-any-wrapper-generated-by-gradle-2.6
[32] http://www.slideshare.net/Kevlin/the-programmer
[33] https://gist.github.com/ksaua/74d75901458235f48da1
[34] https://speakerdeck.com/bmuschko/gradle-plugin-best-practices-by-example
[35] http://bit.ly/gradle-plugin