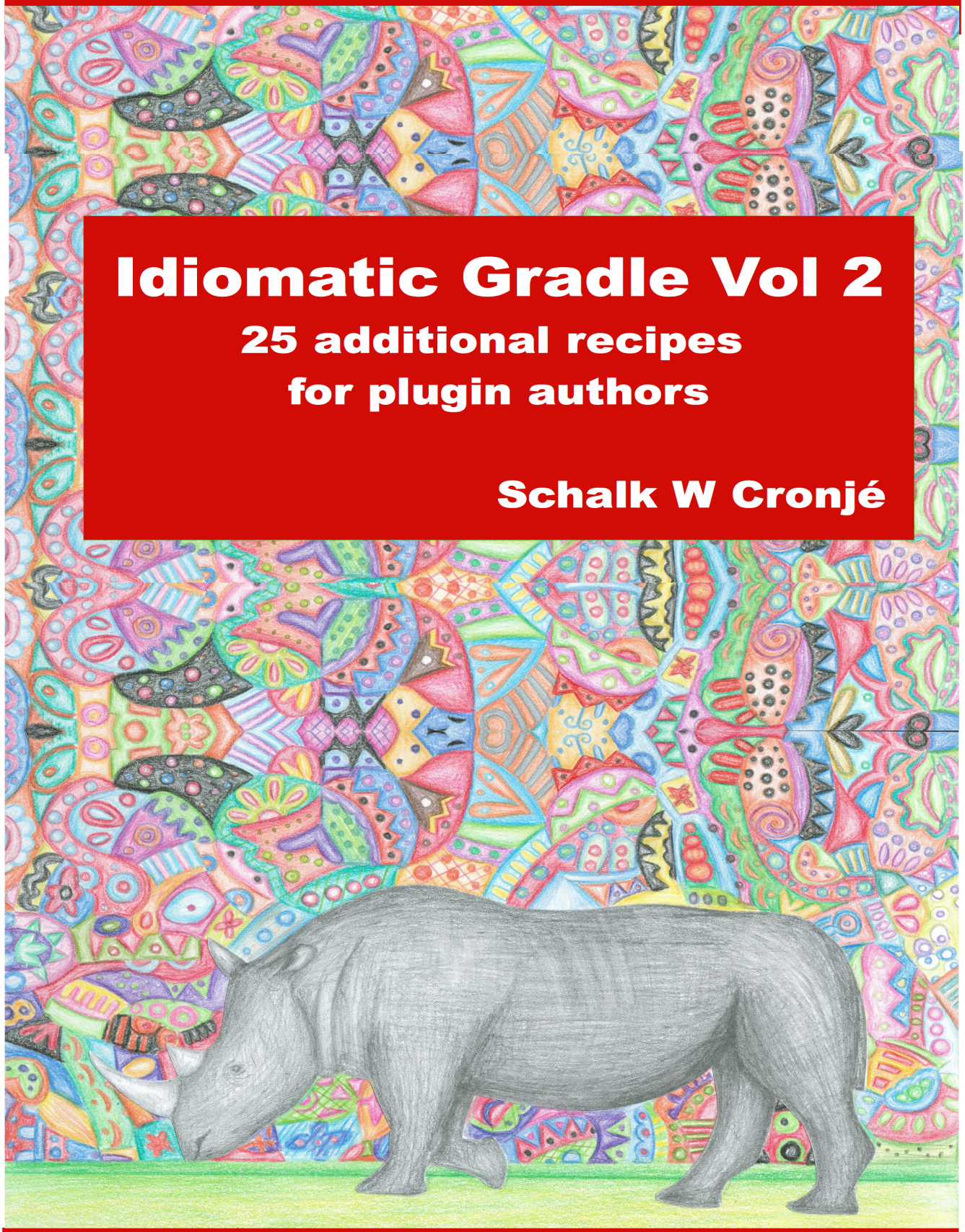
Idiomatic Gradle Vol 2

25 additional recipes for plugin authors

Schalk W Cronjé



ysb33r tech series

Idiomatic Gradle Plugins Vol 2

25 additional recipes for plugin authors

Schalk Cronjé

This book is for sale at http://leanpub.com/idiomaticgradlepluginsvol2

This version was published on 2017-11-18

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

Configure Global, Customise per Task

Summary

Use of extensions is a common pattern for global configuration. In cases where multiple tasks rely on such global configuration, it is sometimes required to customise specific tasks not to use the global configuration. This leads to additional properties on a task with additional testing and complexity.

Solution

Create an extension that can be applied to both the project and tasks. Make the tasks in your plugin that would previously have relied only on a project extension to be task-extension aware. Build the extension in such a way that task extensions are aware of the global project extension. If the properties of the task extension is not explicitly configured by the build script author, the task extension will retrieve the value from the project extension.

Examples

Consider for a moment that you are writing an extension for the GNU Make tool. In this extension you want to set the number of concurrent jobs and as well as the flags that should always be passed to an invocation of the tool. Conceptually you may model this in your extension as follows.

GnuMakeExtension.groovy

```
@CompileStatic
class GnuMakeExtension {
    void setNumJobs(int numJobs) {
        this.numJobs = numJobs
    }

    Integer getNumJobs() {
        this.numJobs
    }

    void flags(final Map<String,String> extraFlags) {
        flags.putAll(extraFlags)
    }
```

```
14
15      Map<String,String> getFlags() {
16          this.flags
17      }
18
19      private Integer numJobs
20      private Map<String,String> flags = [:]
21  }
```

The first step is to add the constructors for both task and project.

GnuMakeExtension.groovy

```
1   final static String NAME = 'gnumake'
2
3   GnuMakeExtension(Project project) {
4       this.project = project
5   }
6
7   GnuMakeExtension(Task task) {
8       this.task = task
9   }
10
11  private final Project project
12  private final Task task
```

Line #1: By convention add the name that the extensions will be known by.

Line #11: Both a `Project` and a `Task` instance is kept in the extension. One of them will always be null. Also ntoe the usage of `final` as these will be one-shot assignments during instantiation of the extension.

The second step is to modify in the internal fields so that they can be recognised as being uninitialised. The most common case is to use null, but depending on your context this might be different. For containers you might want to simply leave them as empty. Now modify your project constructor to set some default values and leave them uninitialised in the task form of the extension.

GnuMakeExtension.groovy

```
1  GnuMakeExtension(Project project) {
2      this.project = project
3      this.numJobs = 4
4  }
5
6  GnuMakeExtension(Task task) {
7      this.task = task
8  }
9
10 private Integer numJobs
11 private final Map<String,Object> flags = [:]
```

Line #3: Assume for the purpose of this example, that in your plugin you want four jobs to always be run concurrently.

No defaults are set in the task constructor.

The final step is to retrieve the values in an intelligent manner. The logic is always to look in the task extension first and then in the project extension. However, this latter logic should only occur when an extension is attached to a task. All other entities in Gradle should not be aware of this behaviour. Start by adding a helper function that will always return the project extension

GnuMakeExtension.groovy

```
1  private GnuMakeExtension getProjectExtension() {
2      project ? this : (GnuMakeExtension)(task.project.extensions.getByName(NAME))
3  }
```

Now continue to modify the getters in the extension to check first whether the latter is attached to a project or a task and modify behaviour accordingly:

GnuMakeExtension.groovy

```
1  Integer getNumJobs() {
2      if(project) {
3          return this.numJobs
4      } else {
5          this.numJobs ?: getProjectExtension().getNumJobs()
6      }
7  }
8
9  Map<String,Object> getFlags() {
10     if(project) {
11         this.flags
12     } else {
13         this.flags.isEmpty() ? getProjectExtension().getFlags() : this.flags
14     }
15 }
```

Line #2: If this is a project extension, return the current value of `numJobs`.

Line #5: If this is a task extension, check whether it has been set. If so return the configured value, otherwise defer to the project extension.

Line #13: In a similar fashion an empty collection can be used to defer to a project extension. The context of your own plugin will determine whether an empty collection is a feasible approach, but this is shown as one possible option.

Having coded your extension it is now time to add it your plugin code and to your plugin's task types. Create the global extension as per usual when the plugin is applied.

GnuMakePlugin.groovy

```
1  void apply(final Project project) {
2
3      project.extensions.create(
4          GnuMakeExtension.NAME,
5          GnuMakeExtension,
6          project
7      )
8  }
```

The task extension is added when the task is instantiated. Assuming you have created a task type called `GnuMakeTask`, just add one line to the constructor.

GnuMakeTask.groovy

```
1  GnuMakeTask() {
2      gnumake = extensions.create(GnuMakeExtension.NAME,GnuMakeExtension,this)
3  }
4
5  private final GnuMakeExtension gnumake
```

Line #2: It is also good practice to store the task extension reference locally so that your code does not have to do a lookup via the extension container everytime.

Accessing the values in the extension simply becomes a case of referring to the local reference. For instance, you may want to add a method that returns the correct command-line parameter for concurrent jobs, This can be done as follows.

GnuMakeTask.groovy

```
1  String getNumJobsParameter() {
2      "-j${this.gnumake.getNumJobs()}"
3  }
```

Line #2: The logic previously coded will ensure that task extension will perform a fallback to the project extension if nothing has been configured.

At this point all that remains is for a build script author to apply your plugin and make use of. Assuming that the build script creates two `GnuMakeTask` instances called `makeProjectA` and `makeProjectB`, configuration becomes readable and comprehensible.

build.gradle

```
1  gnumake {
2      numJobs = 10
3  }
4
5  makeProjectA {
6      gnumake {
7          defaultFlags BUILDNUM : '1234'
8      }
9  }
10
11 makeProjectB {
12     gnumake {
13         numJobs = 1
14     }
15 }
```

Line #2: Configure setting globally

Line #7: `makeProjectA` uses its own set of flags, but use the `numJobs` from the project extension.

Line #13: `makeProjectB` will use its own setting for `numJobs` but will use the global set of flags.

A number of plugins in the field already make use of this recipe. If you want to study their usage and implementations have a look at Node.js + NPM plugins and well as the Packer plugin.

Gradle API Updates

The Gradle team has made advances in moving conventions to a public API. This approach are in many cases still more flexible and useful than conventions.

Caveats

- The above example simplifies the construction of task extensions and makes the assumption that it will always be added with the same name. If you are concerned about potential misuse, then you should add a second parameter to the task constructor which takes the name of the project extension.
- When values in the task extension is modified, the task will not be out of date. If this behaviour is important to you, implement a task input property which can monitor changes in the task extension.

Grolifant

AbstractCombinedProjectTaskExtension[1] is a base class inf Grolifant that can be used to simplify building of this recipe.

---

[1] http://ysb33rorg.gitlab.io/grolifant/0.5.1/api/org/ysb33r/grolifant/api/AbstractCombinedProjectTaskExtension.html

Migrate Extension to Unmanaged Model

Summary

Although Link Extension to Model is a good for a first migration, a plugin author might just decide to completely remove compatibility with older version of a plugin and embrace a new model approach. On the other hand, performing a Migrate Extension to Managed Model might be a step too far as keeping the syntax as close as possible to previous versions of the plugin might make migration for script authors less painful.

A plugin author might also be new to the new software model and might not be *au fait* with some of the intricacies of managed model elements and would prefer to settle for a more familiar unmanaged element.

Solution

Keep the existing extension class the same and enhance it with some declarative DSL methods. The latter will be necessary as unmanaged DSL object are not decorated as would have been the case with project extension objects. Finish it off by adding a model creation rule to create the unmanaged object.

Examples

Continuing on from Link Extension to Model, we can stay with the external tool metaphor. In this case out legacy extension might look something like below for GNU Make.

Legacy extension for external tool

```
class ExternalToolExtension {
    String executable = 'make'
    List<String> execArgs = []

    void execArgs(String... args) {
        this.execArgs.addAll(args as List)
    }
}
```

We can keep our extension code mostly intact and enhance it with some declarative methods. (If we have anything that depends on the legacy `Project` object we need to rework those sections as we won't have access to it anymore).

Legacy extension ehanced and ready to be used ans unmanaged model element

```
class ExternalToolExtension {
    String executable = 'make'
    List<String> execArgs = []

    void execArgs(String... args) {
        this.execArgs.addAll(args as List)
    }

    void executable(String exe) {
        this.executable = exe
    }
}
```

Line #9: This allows for `executable '/usr/bin/make'` in addition to `executable = '/usr/bin/make'`, which is important if we want to keep the recommended declarative style.

All that remains is to add a model creation rule. As this is an unmanged model element our rule has to return an instance of the model element instead of `void` as would be the case for managed model elements.

Creating the ruleset

```
1  class ExtensionContainerRules extends RuleSource {
2      @Model
3      ExternalToolExtension externalTool() {
4          new ExternalToolExtension()
5      }
6  }
```

Line #3: Model creation rules for unmanaged elements always return an instance of the element type.

Line #4: Return an instance of the modified extension type. If the original extension's constructor took a `Project` object as a parameter, it will need to be modified to operrate without access to the `Project` instance.

Once the rules are applied we have all of the greatness of the original extension object available within the model element.

Configuring as a model element

```
1  model {
2      externalTool {
3          executable = 'gmake'
4          execArgs = ['-i']
5          execArgs '-s','-B'
6      }
7
8      externalTool {
9          executable 'amake'
10     }
11 }
```

Line #2: As the model rule was called `externalTool` it is available as a top-level model element by the same name within the model DSL. Anything that was configurable in the original extension is still configurable in the model configuration block as the rest of the callouts demonstrate.

Line #3: Configuration by assignment.

Line #4: List configuration assignment.

Line #5: Append items to the list.

Line #9: Due to the additional method that was added in the extension, we can still use a declarative form.

Caveats

This is a suggested approach if DSL-compatibility with an older plugin version needs to be maintained. However, the following needs to be kept in mind:

- As the extension is unmanaged, Gradle can never guarantee the configuration to be immutable.
- Gradle will not decorate the extension with any other methods, and it is up to the plugin author to add the appropriate enhancements.

References & Credits

- Mark Viera clarified a number of caveats with this approach. [MViera3]

Appendix: Understanding the legacy native software model

Managed Data Annotations

@Managed

Managed model interface

```
1  @Managed
2  interface ManagedDocker {
3      String getDockerHost()
4      void setDockerHost(String host)
5
6      File getCertPath()
7      void setCertPath(File path)
8  }
```

Line #2: Definition of a managed model is either an `interface` or an `abstract class`.

At this point the managed class can already be used in a build script as long as it is available on the classpath.

Model declaration in build script

```
1  model {
2      mysqlContainer(ManagedDocker) {
3          dockerHost 'https://192.168.99.100:2376'
4      }
5  }
```

Line #2: Register an instance of `ManagedDocker` model and call it `mysqlContainer`.

Line #3: Configure the settings on the new model. Any property or property on the managed class can be configured here.

This is effectively the equivalent of what a `@Model` annotation will create. By keeping a mental picture of this equivalence in mind, it will be easier for many of us plugins authors to transition from the classic Gradle way of thinking to this new style.

RuleSource Annotations

These are special annotations that are applied to methods in a class that extends `RuleSource`.

@Model

When applied to a method it indicates that a new top level element is created in the model space. This element takes the name of the method unlike the annotation is given a value. In the latter case the provided name must start with a lowercase character and only consist of ASCII word characters (Regex `\w`).

The element also has to provide a type, but the way of providing the type depends on whether this is a [Managed] type or not.

- ***Managed***: The return type is always void. The first parameter is a type defined somewhere else and annotated with `@Managed`. The first parameter is not allowed to be unmanaged or annotated with `@Path`. (See [Managed] for more details. Any additional parameters are considered inputs.
- ***Unmanaged***: The return type is the model type. All parameters are considered inputs.

A confusing subject

When reading the Gradle documentation on the new model, the term *subject* will be encountered and can cause confusion. All we need to remember that if we see this term in the documentation it refers to a managed or unmanaged type. In all other `RuleSource` annotations beside `@Model` it is always the first parameter of the method.

Declaring a new managed model with an interface

```
1  class ExampleRules extends RuleSource {
2
3      @Model
4      void docker(ManagedDocker md) {}
5
6  }
```

Line #1: All model rule definitions must start by extending `RuleSource`.

Line #4: Minimum declaration required to create a new managed model. Provided typed must be annotated with `@Managed`. Initialisation can be performed in the code block.

When building the model rules for a managed type, the first parameter must always be the type and the return type is void. (This pattern is common across `RuleSource` rule annotation, but it is worth repeating here). The name of the method becomes the name of the model in the model registry.

Using the new model

```
1  apply plugin : ExampleRules
```

Line #1: The model rules class must be applied as a plugin. In many cases this will happen within the plugin class, but there are other ways in which the rules can be automatically applied.

Reinforcing this equivalence of code and script, we should visualise this code as simply being.

Script equivalent

```
1  model {
2      docker(ManagedDocker)
3  }
```

It is also to create a model without using a managed type. Consider that we rather wanted to implement the data class for out Docker descriptor ourselves. A simplified version of it might have looked something like this:

Unmanaged extension

```
class NonManagedDocker {
    String dockerHost = 'http://192.168.1.1:1234'
    File certPath
}
```

In order to use this as a model, the method in our rules needs to return an instance of this class (as opposed to `void` in the managed case).

Unmanaged class as a model

```
@Model
NonManagedDocker dockerNonManaged() {
    new NonManagedDocker()
}
```

Line #2: The name of the method will still become the name of the toplevel model in the build script.

Line #3: Initialise & return the object as appropriate.

The `@Model` annotation also allows for setting a custom name that might be more suitable for use within a DSL than the method name. Simply use `value=NewName` as parameter to the annotation.

Changing the name of the model

```
@Model(value='DrDocker')
void docker(ManagedDocker md) {}
```

Line #1: The toplevel model will be known as `DrDocker` instead of `docker`.

@Defaults

Use the `@Defaults` annotation to set default property values for a subject. The name of the method is irrelevant as far as the build script author is concerned and should rather be named to describe the intent of setting the default value. The first parameter of the method is always the subject, which will be mutable for the duration of the method call. The return type is always `void`.

Rules class with default values.

```
class DefaultExampleRules extends RuleSource {
    @Model
    void docker(ManagedDocker md) {}

    @Defaults
    void defaultServer(ManagedDocker md) {
        md.dockerHost = 'https://192.168.99.100:2376'
    }
}
```

For a given `RuleSource` deriviative, methods annotated with `@Defaults` will execute before methods annotated with `@Model` or `@Mutate` and before any user configuration in the build script. Default are executed in alphabetical order, not order in code, therefore `ruleAbc` will be executed before `ruleDef`. It is possible to have duplicate default rules under different method names. In such cases the last executed one wins.

@Mutate

These are some of the most powerful kind of rules as they are not only used to set values on model subjects, but also to create tasks and other entities. The first parameter of a mutate method is always the subject, which will be mutable for the duration of the method call. Additional parameters can be supplied which can be used as inputs. The return type is always `void`.

For a given `RuleSource` deriviate, methods annotated with `@Mutate` will execute after methods annotated with `@Defaults` and any user configuration in the build script, but before methods annotated with `@Finalize`. Rules are executed in alphabetical order, not order in code, therefore `ruleAbc` will be executed before `ruleDef`. (This might just be an implementation result, and should not be relied upon).

@Finalize

According to gradle core developer, Mark Viera, there is very little difference between `@Finalize` and `@Mutate` rules. They are effectively just two groups of the same things, with all `@Mutate` rules being executed, before any `@Finalize` rules[MViera1]. Anything else that has been mentioned previously for `@Mutate` will apply equally here. From a idiomatic point of view, it is recommended that `@Finalize` rules are only used for final configuration in a similar fashion that `project.afterEvaluate` has been used in the classic model.

For a given `RuleSource` deriviative, methods annotated with `@Finalize` will execute after methods annotated with `@Mutate`, but before methods annotated with `@Validate`. Rules are executed in alphabetical order, not order in code, therefore `ruleAbc` will be executed before `ruleDef`. (This might just be an implementation result, and should not be relied upon).

@Validate

Validation rules are used to check the integrity of properties before execution can begin. The first parameter of the method is always the subject, which will be immutable for the duration of the method call. It is the responsibility of the plugin author to terminate execution upon validation failure by throwing an appropriate exception. Groovy power asserts can also be used as shown below [MViera2]:

Example of using Groovy Power Assert

```
1  @Validate
2  void alwaysUseHttp(ManagedDocker md) {
3      assert md.dockerHost?.startsWith('http')
4  }
```

Line #3: Check that the condition are met and raise exception if not

In the 2.10 and prior versions the exception is propagates as is. From 2.11 onwards the exception is wrapped up in a `ModelRuleExecutionException` which is from the internal `org.gradle.model.internal.core` package.

Inspecting the Model

Script authors should not need to do this, but as plugin author at some stage will probably have th need to look under the hood. This is especially the case when attempting to understand how the new model work the first time or even for some low-level unit tests. For this purpose use a method on the internal `Project` class namely `modelRegistry`. This is best described by means of a little Spock Framework test.

Revisiting our `DefaultExampleRules` class from earlier

Example rules

```
class DefaultExampleRules extends RuleSource {
    @Model
    void docker(ManagedDocker md) {}

    @Defaults
    void defaultServer(ManagedDocker md) {
        md.dockerHost = 'https://192.168.99.100:2376'
    }
}
```

we can author a test to check that the `dockerHost` property has a default value set.

Access via the model registry

```
class DefaultExampleRulesSpec extends Specification {
    def project = ProjectBuilder.builder().build()

    def "Default rule must set dockerHost"() {
        given: "A simple model"
        project.allprojects {
            apply plugin : DefaultExampleRules
        }

        def node = project.modelRegistry.find('docker',ManagedDocker)

        expect: "dockerHost to have default value"
        node?.dockerHost == 'https://192.168.99.100:2376'
    }

    def "Configuration overrides Default rules"() {
        given: "A simple model"
        project.allprojects {
            apply plugin : DefaultExampleRules

            model {
                docker {
                    dockerHost 'https://192.168.99.100:1234'
                }
            }
        }

```

```
28          when:
29          def node = project.modelRegistry.find('docker',ManagedDocker)
30
31          then:
32          node?.dockerHost == 'https://192.168.99.100:1234'
33      }
34
35  }
```

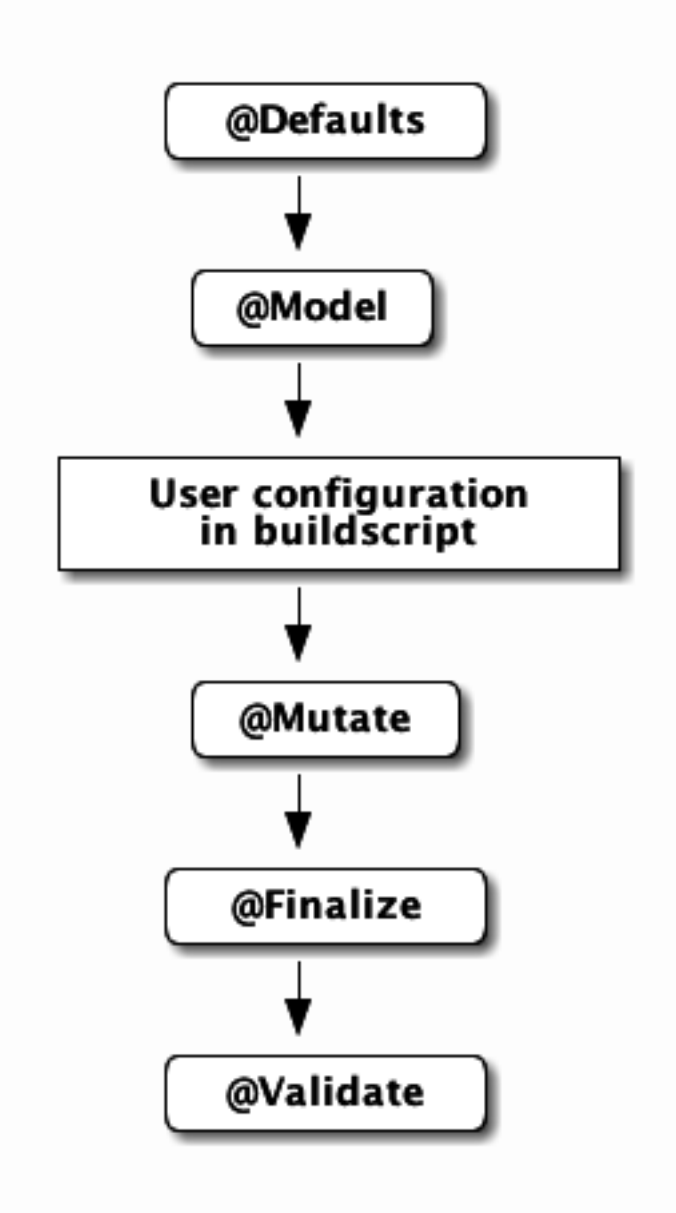
Line #7: Apply the rules class using `apply plugin:` syntax. This might seem strange as one would not expect a rules class to be a plugin class, however since a rules class has to extend the `RuleSource` which implements the appropriate interface.

Line #10: Sneakily using `modelRegistry`, we pass the model name as a string, plus the subject type to registry's find method. If it is exists it will return the instantiated component, otherwise it will be a null value

Line #13: Now it is purely a case of querying the appropriate property from our class.

This technique is merely meant for inspecting the model. Do not use this to attempt to mutate a model. Attempting to do so may lead to undefined behaviour and may bring the wrath of the people who use your plugin upon you. You have been warned!

Appendix: Legacy native software model configuration order



Bibliography

Discussion Forums

[**AMurdoch1**] Adam Murdoch. Obtaining the name of a component in model rules[2].

[**LPelletier**] Luc Pelletier. Limitation of generatedBy in native software model[3].

[**MViera1**] Mark Viera. @Finalize is not well described anywhere in the docs[4].

[**MViera2**] Mark Viera. @Validate needs a a little more clarification[5].

[**MViera3**] Mark Viera. Extension objects in new model[6].

Software

[**GNUMake**] Schalk W. Cronjé. GNU Make Gradle Plugin[7]

[**NodePlugin**] Schalk W. Cronjé. Node.js Gradle Plugin[8]

[**PackerPlugin**] Schalk W. Cronjé. Packer Gradle Plugin[9]

[**Terraform**]

Books

[**SCronje**] Schalk Cronjé. 'Idiomatic Gradle: 25 recipes for plugin authors'. Leanpub[10].

[2] https://discuss.gradle.org/t/obtaining-the-name-of-a-component-in-model-rules/17790/5
[3] https://discuss.gradle.org/t/limitation-of-generatedby-in-native-software-model/17321
[4] http://discuss.gradle.org/t/finalize-is-not-well-described-anywhere-in-the-docs/17553
[5] https://discuss.gradle.org/t/validate-needs-a-little-more-clarification/17554
[6] https://discuss.gradle.org/t/extension-objects-in-new-model/17587
[7] https://github.com/ysb33r/gnumake-gradle-plugin
[8] https://gitlab.com/ysb33rOrg/nodejs-gradle-plugin
[9] https://gitlab.com/ysb33rOrg/packer-gradle-plugin
[10] http://leanpub.com/idiomaticgradle