Clojuresque explained!

apply plugin: "clojure"

Meikel Brandmeyer

Clojuresque explained!

apply plugin: ”clojure”

Meikel Brandmeyer

Tweet This Book!

Please help Meikel Brandmeyer by spreading the word about this book on Twitter!

The suggested hashtag for this book is #clojuresque.

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

https://twitter.com/search?q=#clojuresque

Contents

Chapter 1: Getting Started

In this first short chapter, we will go through an example build script, which makes the build ready to use the plugin. However, this will be strictly limited to the Clojure plugin part! The presented script is not fully usable for a full-fledged project build.

But before we start...

Note

This book is an introduction to get up and running with the Clojure plugin for the Gradle build system. Although we will go slowly through the topics, this book is not intended as a general introduction to Gradle. There are other books, in particular from the official Gradle crew, which are more thorough than this book in this respect[1].

So there is a certain level of familiarity with Gradle itself assumed. However we will point out and explain key concepts. For more details we will point the astute reader to corresponding references in the official documentation of Gradle.

A minimal build script

Let's dive right in with a minimal build script example. It contains only the necessary parts to make the clojuresque plugin work. We will go through each step with a short explanation.

A minal build script

```
1  buildscript {
2      repositories {
3          mavenCentral()
4          maven { url "http://clojars.org/repo" }
5      }
6      dependencies {
7          classpath "clojuresque:clojuresque:1.6.0"
8      }
9  }
10
11 apply plugin: "clojure"
```

[1] http://www.gradle.org/books

```
12
13 repositories {
14     mavenCentral()
15 }
16
17 dependencies {
18     compile "org.clojure:clojure:1.5.1"
19 }
```

This build script consists of two main parts. The first part actually tells Gradle how to get the clojuresque plugin. Since it is a third-party plugin, it is not distributed with gradle proper. So you have to tell Gradle where to look for what to actually make the plugin available for the build.

The buildscript block

```
1 buildscript {
2     repositories {
3         mavenCentral()
4         maven { url "http://clojars.org/repo" }
5     }
6     dependencies {
7         classpath "clojuresque:clojuresque:1.6.0"
8     }
9 }
```

Doing so is done by using a so-called `buildscript` block[2]. In this block we specify meta information which relate not to the project but the build system itself.

Here we specify in the `repositories` section[3] that Gradle can find some of the dependencies of clojuresque in the central maven repository. The plugin itself and other dependencies are hosted in the Clojars repository[4], which is in wide use in the Clojure world. It also uses the maven repository structure. Hence it blends in quite smoothly into the Gradle infrastructure.

Finally, we add the actual plugin to the build script's `dependencies` block. For this we use the special `classpath` configuration[5]. This configuration handles all dependencies for the actual build script and not the project itself. They will not interfere with the project's dependencies.

[2] http://www.gradle.org/docs/current/userguide/organizing_build_logic.html#sec:external_dependencies

[3] http://www.gradle.org/docs/current/userguide/dependency_management.html#sec:repositories

[4] http://clojars.org

[5] http://www.gradle.org/docs/current/userguide/dependency_management.html#sec:how_to_declare_your_dependencies

Note

The version number of the plugin might change. You can enquire the latest version number on the Clojars page of the plugin[6]. There you'll also get some cut'n'paste snippet to easily transfer the dependency to your build script.

With the build script prepared we can dive into the actual project build and tell Gradle to actually use the Clojure plugin.

Applying the plugin

```
11 apply plugin: "clojure"
```

This simple command sets everything up as you are used to it from the Java or Groovy plugins which come with Gradle itself. In fact it also applies the Java plugin, since it provides certain parts of the required infrastructure. Everything is set up expecting the standard Gradle project layout. However, the source goes into the `clojure` subdirectory instead of, for example, `java`. `main` defines the main source code for the projects as usual. The tests go to the `test` source set.

Example project layout

```
.
├── build.gradle
└── src
    ├── main
    │   └── clojure
    │       └── example
    │           └── namespace.clj
    └── test
        └── clojure
            └── example
                └── test_namespace.clj
```

Finally, we specify the dependencies for the actual project. A Clojure project without a dependency on it won't make much sense. So we add it to the compile configuration.

Again we have to add the central maven repository since Clojure is hosted there.

[6] http://clojars.org/clojuresque

The Clojure dependency

```
13 repositories {
14     mavenCentral()
15 }
16
17 dependencies {
18     compile "org.clojure:clojure:1.5.1"
19 }
```

And that's it. That's enough to compile your Clojure code into a jar file which you can distribute further. Of course, there is more to it as will we see in the following chapters. And most likely you will have to set up other things like additional dependencies, the project's version and description, &c &c. But the above is a walk-through through the minimal steps to make it work.

Chapter 2: Source sets and options

After getting the build script ready in the previous chapter, let's now have a deeper look at the various features added by the plugin. These are are extensions to Gradle's source set concept to handle Clojure source code.

Gradle's source set concept is quite powerful[7]. A source set comprises several parts of the project which belong together. Be it source code or additional resources. In the simple case this is just the project source code in the `main` source set and its testing related parts in the `test` source set. But the concept can be driven further as we will shortly see.

Source location

Usually the source code for the different languages go into correspondingly named subdirectories of the source set directory structure. And the Clojure plugin also adheres to this convention. So Clojure code is usually found under `src/main/clojure`.

However, this is only the default convention. You can change the location of the source at any time. Say you want to change the source code layout to something more akin to a leiningen project layout, you can simply override the preconfigured default.

Alternative source location

```
sourceSets {
    main {
        clojure {
            srcDirs = [ "src" ]
        }
    }
    test {
        clojure {
            srcDirs = [ "test" ]
        }
    }
}
```

[7] http://www.gradle.org/docs/current/userguide/java_plugin.html#N11E2A

Note

You specify the root of the Clojure code tree! Not the source files themselves.

Source filtering

A similar standard functionality which works as expected on Clojure source code is filtering in the source sets. Let's say you want to exclude every `bar_baz.clj` under the `foo` directory in the source code, you could do so as follows:

Excluding files

```
sourceSets {
    main {
        clojure {
            exclude "foo/**/bar_baz.clj"
        }
    }
}
```

Additionally the plugin provides some convenience functions to actually work with Clojure namespaces. The same filter as in the previous listing can also be achieved by such a namespace filter.

Excluding Namespaces

```
sourceSets {
    main {
        clojure {
            excludeNamespace "foo.**.bar-baz"
        }
    }
}
```

Clojure-specific options

There are some Clojure specific options, which are added by the plugin. These mainly relate to the compiler behaviour. The project wide configuration options are put into the so-called Clojure extension of the project.

Normally, each source set inherits these settings from the extension. However the can be overriden on source set basis (and ultimatively even on a per-task basis if desired).

AOT compilation

The `aotCompile` option determines whether the Clojure code should actually be compiled in ahead-of-time mode. The default for this option is `false`.

Project-wide setting

```
clojure {
    aotCompile = true
}
```

As described above this setting is inherited by all source sets. To set only a certain source set to be AOT compiled simply set the option in the corresponding Clojure extension of the source set.

Source-set local setting

```
sourceSets {
    main {
        clojure {
            aotCompile = true
        }
    }
}
```

Reflection warning

The `warnOnReflection` option determines whether the Clojure code should emit warnings during compilation in case of non-resolved calls which lead to reflection at runtime. The default for this option is `false`.

Project-wide setting

```
clojure {
    warnOnReflection = true
}
```

As described above this setting is inherited by all source sets. To set only a certain source set to emit reflection warnings simply set the option in the corresponding Clojure extension of the source set.

Source-set local setting

```
sourceSets {
    main {
        clojure {
            warnOnReflection = true
        }
    }
}
```

Chapter 3: Tasks

The Clojure plugin provides several Clojure related tasks, which range from actual compilation to documentation generation and testing. Some of the tasks are pure utilities and not really Clojure specific.

Delayed options

Many options of the tasks are "delayed." That means their value is only retrieved in case it is really needed. Normally you don't notice any difference from normal options. However if needed setting the value can be delayed using a closure.

Delayed options work like this:

```
someTask {
    someOption = project.someSetting
}
```

In this case the option is set immediately to the value of `someSetting`. Later changes to the setting will not be honored!

```
someTask {
    delayedSomeOption = { project.someSetting }
}
```

Now the computation of the value for the option is delayed until it is accessed the first time. Only then the closure is executed to retrieve the actual value. Changes to the setting in between will be honored!

Delayble options will be marked as such.

ClojureExec

The `ClojureExec` task is similar to `JavaExec` but executes a Clojure function instead of a class with a main method. It accepts all options the `JavaExec` task[8] accepts as well.

[8] http://www.gradle.org/docs/current/dsl/org.gradle.api.tasks.JavaExec.html

Options

main

The `main` option describes the fully qualified name of the Clojure function to invoke. The function will be called with the command line arguments.

Execute a clojure function

```
task callClojure(type: ClojureExec) {
    main = "my.name.space/call-me"
}
```

ClojureCompile

The `ClojureCompile` task is in charge of actually compiling the Clojure code. This might seem a bit strange, since Clojure is usually compiled when loaded in the running program.

However, there is on the one hand the so called “ahead-of-time” compilation mode, which speeds up loading time at the expense locking down the used Clojure version. On the other hand the compile task can simply `require` all namespaces to trigger any compilation errors as well as reflection warnings.

For each source set the plugin pre-configures a compile task with this source set as source. The task also inherits any compile related option from the source set.

Inputs

This task is source directory based. That means you specify the source directories, not the source files themselves. A single source directory can be added via the `srcDir` method. Multiple directories can be added via the `srcDirs` method. A whole source set can be added via the `from` method. Setting `srcDirs` overrides any previously added source directories.

Adding sources to the compile task

```
task compileClojure(type: ClojureCompile) {
    srcDir  "a-src-dir"
    srcDirs "b-src-dir", "c-src-dir"
    from    project.sourceSets.main.clojure
    srcDirs = [ "nuke-any-of-the-above" ]
}
```

Note

Source directories are subject to treatment through `project.files`.

Filtering

This task is filterable. That means you can filter source files with the usual Gradle `include`/`exclude` machinery as well as via Clojure namespaces.

Filtering source for the compile task

```
compileClojure {
    exclude "please/**/exclude_me.clj"
    excludeNamespace "please.**.exclude-me"
}
```

Note

The filtering of a source set, which was added as a source, is honored.

Outputs

The compiled class files – if `aotCompile` is `true` – are written to the destination directory specified by the `destinationDir` option. In the case of non-AOT compilation the source files are copied to the destination directory.

This is a delayable option.

Set compile task output destination

```
compileClojure {
    destinationDir = "out"
}
```

Note

For the pre-configured compile tasks the destination directory is set to the classes directory of the source set. Changes to the classes directory setting will be honored.

Options

aotCompile

The `aotCompile` option switches on the AOT compilation mode for this task. The default for this option is `false`. This is a delayable option.

Set AOT compile mode

```
compileClojure {
    aotCompile = true
}
```

Note

For the pre-configured compile tasks the `aotCompile` option is inherited from the source set. Changes to the source set setting will be honored.

warnOnReflection

The `warnOnReflection` option switches on the reflection warnings for this task which aid to track down perfomance impacts be unresolved Java interop calls. The default for this option is `false`. This is a delayable option.

Set reflection warnings

```
compileClojure {
    warnOnReflection = true
}
```

Note

For the pre-configured compile tasks the `warnOnReflection` option is inherited from the source set. Changes to the source set setting will be honored.

classpath

The `classpath` option controls the compilation classpath. It is subject to all the usual classpath handling facilities provided by Gradle.

This is a delayable option.

Setting the classpath

```
compileClojure {
    classpath = project.files(project.configurations.compile)
}
```

Note

For the pre-configured compile tasks the `classpath` refers to the `compileClasspath` of the corresponding source set.

fileMode

The `fileMode` option is used to set the file mode when copying files in non-AOT compilation mode. For some SCMs, which set files to read-only, setting this explicitly might be necessary to allow recompilation of files. The default for this option is `null`. The option is identical to the `fileMode` option of the official `Copy` task[9].

[9] http://www.gradle.org/docs/current/dsl/org.gradle.api.tasks.Copy.html#org.gradle.api.tasks.Copy:dirMode

Set file mode in non-AOT mode

```
compileClojure {
    fileMode = 0644
}
```

dirMode

The `dirMode` option is used to set the directory mode when copying files in non-AOT compilation mode. For some SCMs, which set files to read-only, setting this explicitly might be necessary to allow recompilation of files. The default for this option is `null`. The option is identical to the `dirMode` option of the official `Copy` task[^copy].

Set dir mode in non-AOT mode

```
compileClojure {
    dirMode = 0755
}
```

jvmOptions

The `jvmOptions` option takes a closure, which is executed against the underlying `ClojureExec` task. This way things like heap size etc. can be set.

This is a delayable option.

Set JVM exec options

```
compileClojure {
    jvmOptions = {
        maxHeapSize = "200m"
    }
}
```

ClojureTest

The `ClojureTest` task is in charge of actually running the Clojure tests. Currently only `clojure.test` is supported for testing.

Currently testing doesn't blend in well with the rest of the Gradle ecosystem. There are first steps done to integrate test runs with junit reporting. However this is far from being complete.

There is only one `ClojureTest` task pre-configured. Namely for the `test` source set of the project.

Inputs

This task is source directory based. That means you specify the source directories, not the source files themselves. A single source directory can be added via the `srcDir` method. Multiple directories can be added via the `srcDirs` method. A whole source set can be added via the `from` method. Setting `srcDirs` overrides any previously added source directories.

Adding sources to the test task

```
task clojureTest(type: ClojureTest) {
    srcDir  "a-src-dir"
    srcDirs "b-src-dir", "c-src-dir"
    from    project.sourceSets.test.clojure
    srcDirs = [ "nuke-any-of-the-above" ]
}
```

Note

Source directories are subject to treatment through `project.files`.

Filtering

This task is filterable. That means you can filter source files with the usual Gradle `include`/`exclude` machinery as well as via Clojure namespaces.

Filtering source for the test task

```
clojureTest {
    exclude "please/**/exclude_me.clj"
    excludeNamespace "please.**.exclude-me"
}
```

Note

The filtering of a source set, which was added as a source, is honored.

Outputs

FIXME: junit output

Options

junit

junitOutputDir

classpath

The `classpath` option controls the test classpath. It is subject to all the usual classpath handling facilities provided by Gradle.

Setting the classpath

```
clojureTest {
    classpath = project.files(project.configurations.testRuntime)
}
```

This is a delayable option.

Note

For the pre-configured `clojureTest` task the `classpath` refers to the `testRuntime` configuration of the project, the `main` source set's output and the `test` source set's source directories.

jvmOptions

The `jvmOptions` option takes a closure, which is executed against the underlying `ClojureExec` task. This way things like heap size etc. can be set.

Set JVM exec options

```
clojureTest {
    jvmOptions = {
        maxHeapSize = "200m"
    }
}
```

This is a delayable option.

Note

For the pre-configured `clojureTest` task inherits the `jvmOptions` from the `main` source set's `compileClojure` task.

ClojureDoc

The `ClojureDoc` task is in charge of generating API documentation based on Vars and their docstrings. It uses codox[10] for generating the documentation.

There is only one `ClojureDoc` task pre-configured. Namely for the `main` source set of the project.

Note

If you want to generate also the documentation for another source set, you'll most likely want to add it to the pre-configured task as an additional source. You cannot merge the output of two `ClojureDoc` tasks.

Inputs

This task is source directory based. That means you specify the source directories, not the source files themselves. A single source directory can be added via the `srcDir` method. Multiple directories can be added via the `srcDirs` method. A whole source set can be added via the `from` method. Setting `srcDirs` overrides any previously added source directories.

[10] https://github.com/weavejester/codox

Adding sources to the doc task

```
task clojuredoc(type: ClojureDoc) {
    srcDir  "a-src-dir"
    srcDirs "b-src-dir", "c-src-dir"
    from    project.sourceSets.test.clojure
    srcDirs = [ "nuke-any-of-the-above" ]
}
```

Note

Source directories are subject to treatment through `project.files`.

Filtering

This task is filterable. That means you can filter source files with the usual Gradle `include`/`exclude` machinery as well as via Clojure namespaces.

Filtering source for the doc task

```
clojuredoc {
    exclude "please/**/exclude_me.clj"
    excludeNamespace "please.**.exclude-me"
}
```

Note

The filtering of a source set, which was added as a source, is honored.

Outputs

The compiled documentation is written to the destination directory specificed by the `destinationDir` option.

This is a delayable option.

Set doc task output destination

```
clojuredoc {
    destinationDir = "docs"
}
```

Note

For the pre-configured doc task the destination directory is set to the `clojuredoc` subdirectory of the docs directory of the project. Changes to the docs directory setting will be honored.

Options

codox

The `codox` option takes a simple map with options for codox. At the moment these are the following:

- `writer`
- `srcDirUri`
- `srcLinenumAnchorPrefix`

Set codox options

```
clojuredoc {
    codox = [
        writer: "my.custom/doc-writer",
        srcDirUri: "http://github.com/clojure/clojure/blob/master/",
        srcLinenumAnchorPrefix: "L"
    ]
}
```

Note

The option names are automatically snake-cased for you.

classpath

The `classpath` option controls the classpath. It is subject to all the usual classpath handling facilities provided by Gradle.

Setting the classpath

```
clojuredoc {
    classpath = project.files(project.configurations.runtime)
}
```

This is a delayable option.

Note

The pre-configured doc task inherits the `classpath` from the `main` source set's `compile` task.

jvmOptions

The `jvmOptions` option takes a closure, which is executed against the underlying `ClojureExec` task. This way things like heap size etc. can be set.

Set JVM exec options

```
clojuredoc {
    jvmOptions = {
        maxHeapSize = "200m"
    }
}
```

This is a delayable option.

Note

The pre-configured doc task inherits the `jvmOptions` from the `main` source set's `compile` task.

ClojureRepl

The `ClojureRepl` task is in charge of starting a nrepl server for the project. This is particularly important for an effective Clojure development cycle, since you usually work with tight integration of the editor and the repl server.

This task is supposed to be started and keeps running. You can parallel builds on the same project as long as you don't change the `build.gradle` file. In that case you will get cache issues. You then have to restart the repl task to be able to run parallel builds again.

Note

Currently there is no way to get an interactive repl. This due to some limitations on the groovy side of Gradle. You'll have to connect to the repl server by a client as provided by most of the development environments.

Options

port

The `port` option controls the port where the repl server will listen for connections from clients. This may be a string or an integer.

Setting the repl server port

```
clojureRepl {
    port = 7888
}
```

Note

The pre-configured repl task uses `7888` as default for the port.

handler

The `handler` option controls the repl handler which the server will pass control to after receiving a client connection. The option is the fully qualified name of the handler. If not set, a default handler as provided by `tools.nrepl` will be used.

Setting the repl handler

```
clojureRepl {
    handler = "my.repl/handler"
}
```

middleware

The `middleware` option consists of a list of middleware to apply to the default handler. The middlewares are given by their fully qualified names.

This is intended as a short-cut for the common case, that you only need additional middlewares but not a truely custom handler. If a custom handler is set, this option is ignored.

Setting the repl middleware

```
clojureRepl {
    middleware << "my.repl/middleware"
}
```

classpath

The `classpath` option controls the classpath. It is subject to all the usual classpath handling facilities provided by Gradle.

Setting the classpath

```
clojureRepl {
    classpath = project.files(project.configurations.testRuntime)
}
```

This is a delayable option.

Note

The pre-configured repl task uses as the classpath:

1. all source sets' source directories
2. all source sets' output
3. the `testRuntime` configuration
4. the `development` configuraiton

So changes to the source files will be visible in the repl.

jvmOptions

The `jvmOptions` option takes a closure, which is executed against the underlying `ClojureExec` task. This way things like heap size etc. can be set.

Set JVM exec options

```
clojureRepl {
    jvmOptions = {
        maxHeapSize = "200m"
    }
}
```

This is a delayable option.

Note

The pre-configured repl task inherits the `jvmOptions` from the `main` source set's `compile` task.

Upload

TaskWatcher

Inputs

Uberjar

Inputs

Outputs

Deps

Chapter 4: Tips'n'Tricks

FIXME: Some real project, which is actually non-trivial which shows a full setup with some tips around the build script.

Chapter 5: The nitty gritty

FIXME: Some details on the actual plugin structure should someone ever want more fine-grained control.

Plugin: common

Plugin: base

Plugin: nrepl

Plugin: clojars

Plugin: extras

Chapter 6: Cheatsheet

FIXME: Some short tabular overview for each task with its options.