A plugin that allows you to publish Grails plugins, either to a public or private repository. It also supports deploying Grails applications and plugins to Maven repositories without the need to use Maven directly.

Release Plugin - Reference Documentation

Authors: Peter Ledbrook

Version: 1.0.0.RC3

Table of Contents

1 Introduction

Many Grails projects are private applications and the only "publication" of those applications involves deploying the generated WAR file to a servlet container like Tomcat. But sometimes, particularly in the case of plugins, you want to share the project's WAR, zip or JAR. That's where the Release plugin comes in.

Once this plugin is installed, you can immediately start publishing your artifacts to Maven-compatible repositories. In the case of plugins, you can also publish them to Subversion-based repositories such as the Grails Central Plugin Repository. This user guide will show you exactly what you need to do.

1.1 Migrating from release-plugin

Starting with Grails 2.0, the Release plugin replaces the old `release-plugin` command. Fortunately, you should have little trouble switching to the new `publish-plugin` command. Let's take a look at the most important differences (apart from the name change!).

No --zipOnly option

The new `publish-plugin` command has a more flexible approach to source control management than `release-plugin` and so the `--zipOnly` option is no longer available. Instead, you specify whether you want the plugin to manage source control integration via the new `--scm` and `--noScm` options. The former is effectively the default, but you can disable SCM for the project via the `grails.release.scm.enabled` build configuration option.

The source control management is handled by plugins, such as the Subversion plugin that this plugin depends on. If you don't have the appropriate plugin for your SCM system, you won't be able to use the SCM integration.

Configuration changes

Although your old repository configurations will continue to work, i.e. ones like `grails.plugin.repos.distribution.<id>` used by `release-plugin`, there is far more flexibility if you migrate to the latest style of repository and portal configuration. In particular, by using the syntax described in the configuration chapter you'll be able to put repository credentials and URLs in your personal ~/.grails/settings.groovy file.

You can also continue to use the repository configuration options supported by the old Maven Publisher plugin, but again, the new syntax is preferable.

New metadata

You can put extra metadata into your plugin descriptor, which will then be reflected both in the plugin's POM and in the Grails plugin portal. For example, you can specify:

```
def license = "APACHE"
def organization = [ name: "SpringSource", url:
"http://www.springsource.org/" ]
def developers = [
        [ name: "Peter Ledbrook", email: "pledbrook@somewhere.net" ],
        [ name: "Graeme Rocher", email: "grocher@somewhere.net" ] ]
def issueManagement = [ system: "JIRA", url:
"http://jira.grails.org/browse/GRAILSPLUGINS" ]
def scm = [ url:
"http://svn.grails-plugins.codehaus.org/browse/grails-plugins/" ]
```

Note that the `developers` property should only include additional entries over and above the `author`. In other words, the primary author should still be declared in the `author` and `authorEmail` properties with only additional people going into the `developers` list.

2 Configuration

You can use the `publish-plugin` command to release plugins to the Grails Central Plugin Repository without any configuration whatsoever, but if you want to manage your own plugin repository and/or portal you will have to master the available configuration settings. The same goes if you want to deploy artifacts to a custom Maven-compatible repository. But don't worry: it's all very straightforward.

All the configuration options described in the next section can either go into your project's `BuildConfig.groovy` file or your personal `~/.grails/settings.groovy`. The latter is particular useful for storing credentials since the file is typically not stored in a shared source repository, thus making it easy to keep that information confidential.

Let's start by looking at how you can configure a Maven-compatible repository, since this is the most common scenario.

2.1 Repositories

With tools like Nexus and Artifactory, it's pretty easy to set up a Maven-compatible repository these days. Fortunately, it's equally easy to publish Grails plugins and applications to such repositories.

Let's say you have a repository running on your local machine and you want to deploy a plugin to it. Your first step should be to assign the repository a unique ID, such as 'myRepo'. Next, you tell Grails where to find the repository by adding an entry to `BuildConfig.groovy` specifying its URL:

```
grails.project.repos.myRepo.url = "http://localhost:8081/repos"
```

The general form of the above configuration option is `grails.project.repos.<repoId>.url` .

> ⚠ You can not use a repository ID of 'grailsCentral'. That is because it is reserved for the Grails Central Plugin Repository. The good news is that you can configure the username and password for 'grailsCentral' via the options described below.

You can now deploy artifacts to the repository by passing a `--repository=myRepo` argument to either the publish-plugin or maven-deploy commands. Since you often deploy a plugin or application to the same repository again and again, typing that argument gets a bit laborious, so you can also specify the name of a default repository to deploy to:

```
grails.project.repos.default = "myRepo"
```

The above configuration option means that a project will be deployed to "myRepo" unless an explicit `--repository` argument is provided on the command line. You can also pass a value of "grailsCentral" for the command line option or the 'default' config setting to indicate you want to publish to the Grails Central Plugin Repository. Note that the command line option always takes precedence over other settings.

Things become slightly more verbose if you want to provide extra details about the repository. What about user credentials for example? Here's a comprehensive configuration:

```
grails.project.repos.myRepo.url = "http://localhost:8081/repos"
grails.project.repos.myRepo.type = "maven"
grails.project.repos.myRepo.username = "admin"
grails.project.repos.myRepo.password = "password"
grails.project.repos.myRepo.portal = "grailsCentral"
```

As you can see, even a complete explicit configuration is pretty short. So what do the various entries mean?

- `url` - the URL to use when connecting to the remote repository. Typically this is HTTP-based, but "svn+ssh" is not uncommon for old-style Subversion plugin repositories.
- `type` - can be either "maven" or "svn", but the former is the default value so it's rare to explicitly declare a value of "maven" for this option.
- `username` - the username for connecting to the repository.
- `password` - the password for connecting to the repository.
- `portal` - the ID of the plugin portal to notify when publishing a plugin to this repository. Only affects the `publish-plugin` command.

These settings are common to both Maven-compatible and Subversion repositories - it's only the values that typically differ between them. It's also worth bearing in mind that Subversion repositories can only be used for plugins.

The last option, `portal`, raises the question of how to declare plugin portals. What exactly is meant by a portal ID? The answer lies with portal configuration.

2.2 Plugin portals

When a Grails plugin is published to a repository, a plugin portal can optionally be notified of the release. For example, when you publish a plugin to the Grails Central Plugin Repository, the command will automatically notify the main plugin portal on the grails.org website. That means people can see the details of the new release almost immediately.

Like repositories, portals have very few configuration options:

```
grails.project.portal.<portalId>.url = "http://beta.grails.org/plugin/"
grails.project.portal.<portalId>.username = "joe"
grails.project.portal.<portalId>.password = "ht56jU&B"
```

- `url` - the URL of the plugin portal.
- `username` - the username to use when notifying the portal.
- `password` - the password to use.

One thing to bear in mind: as with repositories, the ID 'grailsCentral' is reserved, this time for the main plugin portal on grails.org. Of course, you can still configure a username and password for this portal using the above settings.

2.3 Source control management

By default, source control management is enabled for the `publish-plugin` command. This means that the command ensures that the latest changes are committed and tagged before a plugin is published. If you don't want the command to do this, then you can disable it via the `--noScm` command line option, but that gets tedious if you use it every time you run the command.

An alternative approach is to use a configuration setting to disable source control management for the project (or all projects if you put it into `~/.grails/settings.groovy`):

```
grails.release.scm.enabled = false
```

Once the above setting is in place, `publish-plugin` will no longer attempt to commit and tag source changes. Of course, if you do this you lose the benefit of the plugin keeping the source and the published plugins reliably in sync.

3 Maven integration

One of the best things that sprung from Maven was a standard way to provide Java dependencies to projects via HTTP-based repositories. Not only do we now have the Maven Central repository, but it's almost trivial to set up your own company-wide Maven-compatible repositories using tools like Nexus or Artifactory. Following on from the Maven Publisher plugin, the Release plugin provides everything you need to easily and effectively deploy your project artifacts to such repositories.

Before we look at deployment to one of these remote repositories, let's look at another aspect of Maven: installing artifacts into the local Maven cache.

3.1 The local Maven cache

When Maven builds a project, it first looks for the project's dependencies in the local Maven cache (by default `$HOME/.m2/repository`). Only if a dependency is not in the cache does Maven pull it from the appropriate remote repository. This makes testing pretty easy: simply install your own version of the artifact into the Maven cache and that's the one that Maven will use. You don't have to deploy a development version of an artifact to a remote repository just to test it.

This approach doesn't only work for Maven. Grails can also pull artifacts from the Maven cache if you add the following entry to your project's `BuildConfig.groovy` file:

```
grails.project.dependency.resolution = {
    …
    repositories {
        mavenLocal()
        …
    }
    …
}
```

So how do you get your plugins or WAR files into the Maven cache? With the maven-install command:

```
grails maven-install
```

That's it! You can then test your new artifact locally. If you want, you can change where your artifacts are installed by adding the following configuration option to either `BuildConfig.groovy` or `settings.groovy`:

```
grails.project.mavenCache = "target/m2cache"
```

By adding this option to `settings.groovy` it will apply to every project that doesn't override it, so be careful!

Installing artifacts to the local Maven cache is trivial, so what about remote deployment?

3.2 Deployment to remote repositories

TBC See maven-deploy for the moment.

4 Publishing Grails plugins

If you want to make your Grails plugins available to other people, then the best approach is to publish them using this plugin. The publication process is straightforward:

1. Package the plugin as a zip or jar
2. Deploy it to a Maven or Subversion repository
3. Optionally notify a plugin portal of the release

The publish-plugin command does all of this for you. It can also integrate with a source control management (SCM) provider if a corresponding plugin is installed, ensuring that all your changes are committed and tagged before the plugin is published. You'll find more information about this later.

4.1 Publishing to Grails Central Plugin Repository

The most common use case for the `publish-plugin` command is to publish a public plugin so that it is available to all Grails users; well, to those that have an internet connection at least. Because of this, the command's default configuration is geared towards the Grails Central Plugin Repository. As long as you have a Xircles account, simply installing the Release plugin and executing

```
grails publish-plugin
```

will do everything necessary to publish the plugin and announce its release. You may be asked whether you want to import the plugin source into source control and for your Xircles and grails.org usernames and passwords, but otherwise that's it. Your plugin will be deployed to the Grails Central Plugin Repository, its plugin portal page on grails.org will be updated, and the release will be announced on various channels such as Twitter (user @grailsplugins).

This process can be streamlined in several ways. For example, if you are always asked whether you want to import the plugin source into SCM, you can disable the check with either the `--noScm` command line option or by adding this setting to either `BuildConfig.groovy` or `~/.grails/settings.groovy`:

```
grails.release.scm.enabled = false
```

It's also a pain to enter your username and password every time you publish a plugin, but you can set these for both the Grails Central Plugin Repository and the grails.org plugin portal in `~/.grails/settings.groovy`:

```
grails.project.repos.grailsCentral.username = "me"
grails.project.repos.grailsCentral.password = "s0longf!shthanks"
grails.project.portal.grailsCentral.username = "jane.doe"
grails.project.portal.grailsCentral.password = "gra!lsr0cks"
```

The above values will be used whenever you publish to the Grails Central Plugin Repository and grails.org.

4.2 Custom repositories and plugin portals

Not all plugins should be publicly available: some are too specific to certain projects while others are confidential. Still, teams can gain big advantages from deploying such plugins to their own private repositories. Because of that, the Release plugin can of course publish plugins to any target repository and even notify private plugin portals.

Repository types and configuration

Grails supports two types of plugin repository: the traditional Subversion-based one and Maven-compatible. The public plugin repository is currently an example of the former. All you need to do is set up a standard Subversion repository and the Release plugin will do the rest.

Maven-compatible ones can be easily set up with either Artifactory or Nexus. Subversion-based repositories are useful for older versions of Grails, but if you only use Grails 1.3 or above, we recommend you use a Maven-compatible repository for your plugins.

Whichever type of repository you go for, the configuration settings are mostly the same. The key options are the repository URL and the username and password for deploying to the repository. All of these can be configured in either `~/.grails/settings.groovy` or `BuildConfig.groovy`. Typically, a single repository hosts many plugins, so it's usually a good idea to put the configuration in `settings.groovy` and then specify in `grails-app/conf/BuildConfig.groovy` which repository a plugin should be published to by default:

```
grails.project.repos.default = "companyRepo"
```

or

```
grails.project.repos.default = "grailsCentral"
```

for example. This saves you from having to use the `--repository` command line argument every time you publish a plugin, although you can still use it to override the default. For example, you may have a local Maven-compatible repository you want to test deployment to, so you could configure it in `settings.groovy` and then run:

```
grails publish-plugin --repository=localReleases
```

One last thing: the Release plugin handles the differences between publishing to Subversion and Maven-compatible repositories, but you still need to let it know what type you're using. By default it assumes a Maven-compatible repository, but you can declare as repository as a Subversion one through its `type` configuration option. Just set it to "svn".

Your own plugin portal

How would you like your own plugin portal for your private plugins? You'll be able to see what plugins are available, tag them and do searches. The public plugin portal at grails.org is part of that web site, but it's also an open source project on GitHub. There are also plans to extract the plugin portal so that it can be used independently of the grails.org web site.

Currently, your best option if you want your own portal is to grab a copy of the grails.org source code and run your own version locally. It's a Grails application, so it's not hard to get it started. Once it's up and running, it provides a REST API that the Release plugin can use for plugin release notifications. The portal updates the information for the given plugin and then announces the release in various ways. You'll probably want to disable the announcements, which you can do by modifying the PluginUpdateService.announceRelease() method.

Configuring the Release plugin to notify your own portal is straightforward and described in the configuration section. Remember, you can specify a default portal for each project and you can also override that default via the `--portal` command line argument.