Grails Plugin Best Practices

Burt Beckwith

SpringSource

@burtbeckwith

https://burtbeckwith.com/blog/

Also see:

http://gr8conf.eu/Presentations/Grails-Plugin-Best-Practices

My Plugins

- Acegi
- App Info
- App Info Hibernate
- AspectJ (in-progress)
- Atomikos
- Binary Artifacts
- BlazeDS (reworked)
- Cache
- Cache-Ehcache
- Cache-Redis
- Cache-Gemfire
- Cloud Foundry
- Cloud Foundry UI
- Cloud Support
- CodeNarc
- Console (rework, current owner)
- Database Reverse Engineering
- Database Sessions
- Database Migration
- Database Migration JAXB
- Datasources
- Dumbster
- Dynamic Controller
- Dynamic Domain Class (core code)
- EJB (in-progress)
- EJB Glassfish (in-progress)
- FamFamFam
- Flex (reworked, current owner)
- Flex Scaffold (major rework, not yet released)
- HDIV (in-progress)

My Plugins

- Heroku
- Hibernate Filter (rework, current owner)
- Jbossas
- jdbc-pool (rework, current owner)
- JMX
- LazyLob
- Logback
- Memcached
- P6Spy UI
- Ratpack
- Remoting (reworked, current owner)
- SpringMVC
- Spring Security Core
- Spring Security ACL
- Spring Security App Info
- Spring Security CAS
- Spring Security Kerberos
- Spring Security LDAP
- Spring Security OAuth Consumer (in-progress)
- Spring Security OAuth Provider (in-progress)
- Spring Security Open ID
- Spring Security Shiro
- Spring Security UI
- Standalone
- standalone-tomcat-memcached
- standalone-tomcat-redis
- tcpmon
- Twitter
- UI Performance
- Webxml (rework, current owner)

What Is A Plugin?

What Is A Plugin?

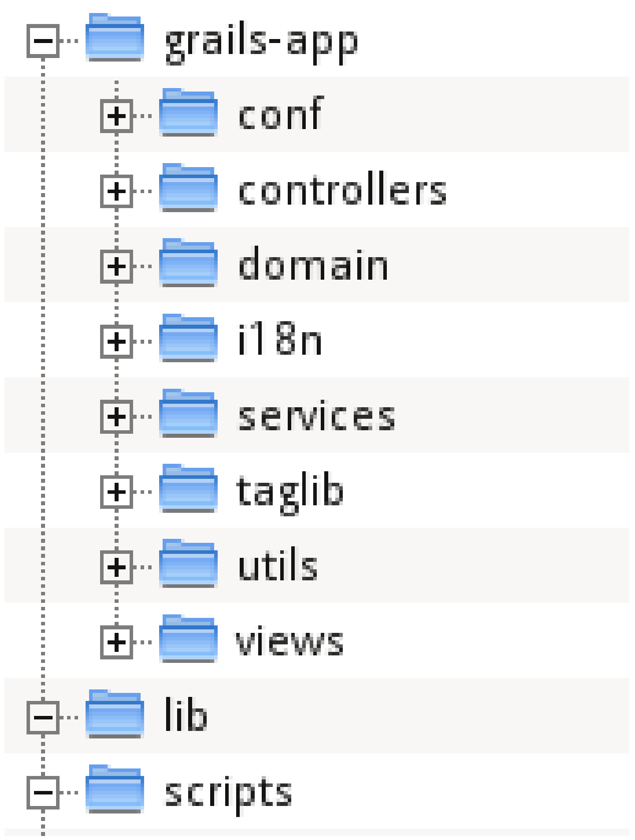
“A set of software components that adds specific abilities to a larger software application

https://secure.wikimedia.org/wikipedia/en/wiki/Plug-in_(computing)

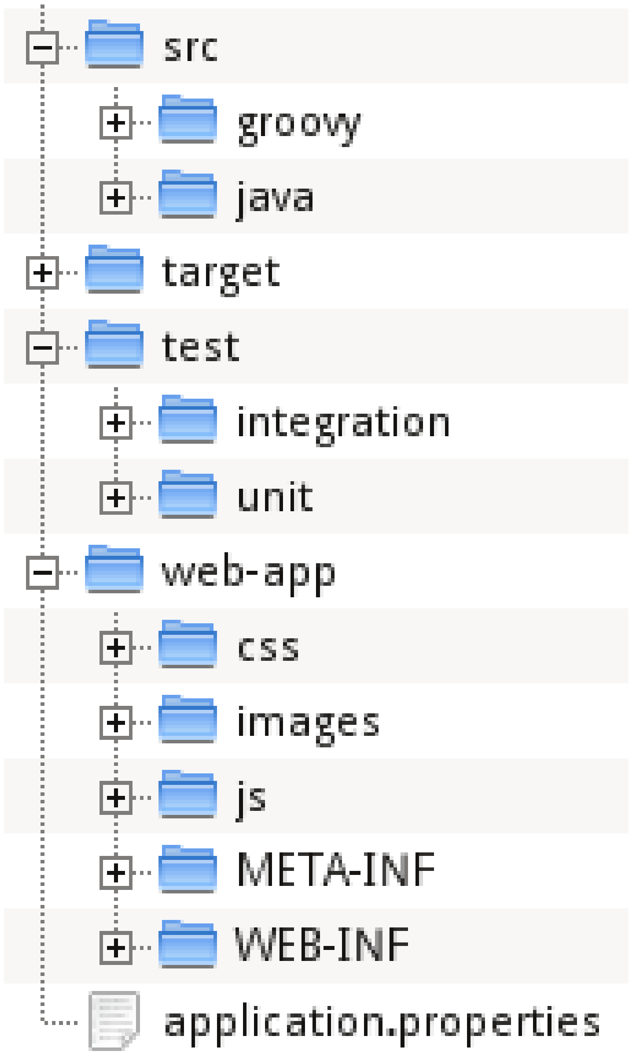
What Is A Plugin?

- Very similar to a Grails application project
- Plus a *GrailsPlugin.groovy plugin descriptor file
- Use `grails create-plugin` to create one
- Often adds artifacts, resources, scripts
- Good for code reuse
- Modular development

Application Directory Structure

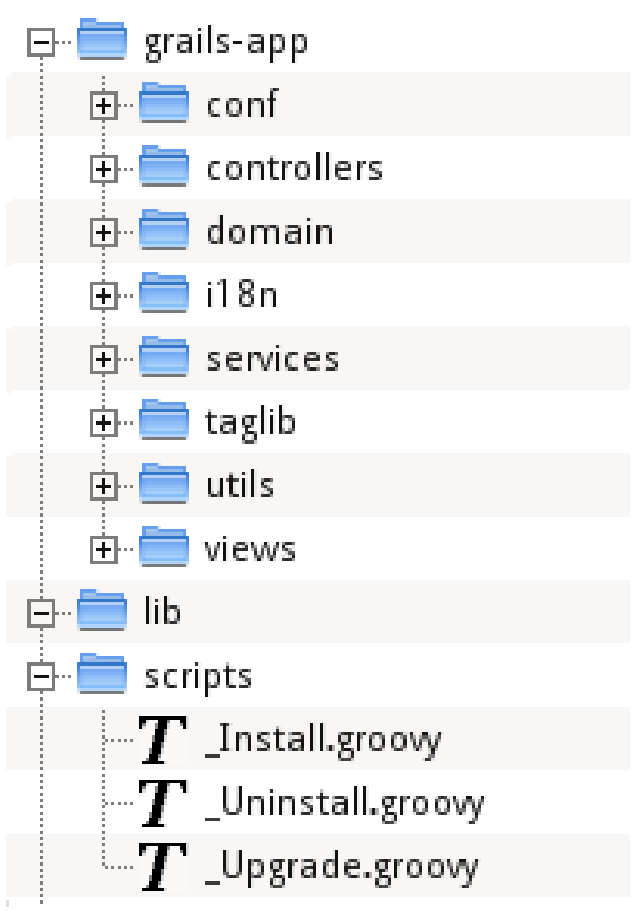


```
grails-app
    conf
    controllers
    domain
    i18n
    services
    taglib
    utils
    views
lib
scripts
```

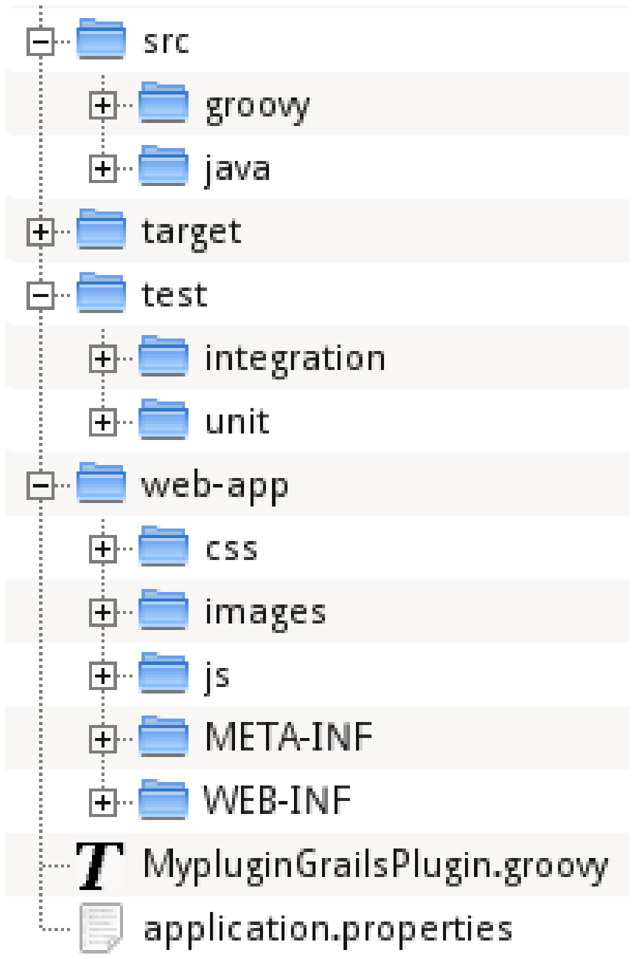


```
src
    groovy
    java
target
test
    integration
    unit
web-app
    css
    images
    js
    META-INF
    WEB-INF
application.properties
```

Plugin Directory Structure



```
grails-app
    conf
    controllers
    domain
    i18n
    services
    taglib
    utils
    views
lib
scripts
    _Install.groovy
    _Uninstall.groovy
    _Upgrade.groovy
```



```
src
    groovy
    java
target
test
    integration
    unit
web-app
    css
    images
    js
    META-INF
    WEB-INF
MypluginGrailsPlugin.groovy
application.properties
```

Best Practices and Tips

Best Practices and Tips

- Delete anything auto-generated that you don't use
  - `grails-app/views/error.gsp`
  - Empty `grails-app/i18n/messages.properties`
  - `_Uninstall.groovy` and other generated scripts
  - `web-app` files
  - `grails-app/conf/UrlMappings.groovy`
  - Empty descriptor callbacks

Best Practices and Tips

- Exclude files used in development or testing
  - Use `def pluginExcludes = [...]`
  - src/docs
  - Test artifacts
- Create **`.gitignore`** (manually or with **`integrate-with --git`**)
- Open ***every*** file, decide that you own it or not

Best Practices and Tips

- Use dependency management in **`BuildConfig.groovy`**, not lib dir
- Change the minimum Grails version to 2.0 or the lowest version you're comfortable supporting:
  - **`def grailsVersion = '2.0 > *'`**

Best Practices and Tips

- Write tests
- Run them often
- Run tests before commit and especially before release (use Gradle or build.xml)
- Create test apps (ideally programmatically)
- Programmatically build inline plugin location:

```
def customerName = System.getProperty("customer.name")
grails.plugin.location."$customerName" = "customer-plugins/$customerName"
```

Best Practices and Tips

- Prefer Java to Groovy (or `@CompileStatic`) for performance if applicable
- Instead of using inline plugins, install into a test app and use Meld or another diff tool to keep in sync
- Before you release, run `package-plugin` and open the ZIP, look at what's there

Best Practices and Tips

- **Use a sensible naming convention, e.g. grails.plugin.<pluginname>**
- **Most plugin metadata is not optional:**
  - Fix `grailsVersion`
  - Update `description` and `documentation`
  - Set value for `license`, `issueManagement`, `scm` (and `organization`, `developers` if applicable)

Best Practices and Tips

- Get to 1.0
  - Initial version 0.1 is just a suggestion
  - People trust 1.0+ more
- Start using source control early
  - Easy to run `git init` locally and later → GitHub
- Support your plugin!

Best Practices and Tips

- http://grails.org/Creating+Plugins
- http://grails.org/The+Plug-in+Developers+Guide
- http://blog.springsource.com/2010/05/18/managing-plugins-with-grails-1-3/
- http://grails.org/doc/latest/guide/plugins.html
- http://burtbeckwith.com/blog/?p=1973 (converting apps ↔ plugins)

Planning for Customization

Planning for Customization

- Plugins are compiled first, then the app; this means that everything can be overridden
- Prefer Spring beans to using **`new`**
- Prefer dynamic instantiation (to allow class name override) to using **`new`**
- Use **`protected`**, not **`private`** and avoid **`static`**
- Move logic from plugin descriptor to classes

Extension Points

- The Build System
- Spring Application Context
- Dynamic method registration
- Auto Reloading
- Container Config (web.xml)
- Adding new Artefact Types

Types of Plugin

- **New functionality**
  - Datasources, UI Performance, etc.
- **Wrapper**
  - Spring Security, Searchable, Quartz, etc.
  - Often have many scripts, e.g. Cloud Foundry, Database Migration
- **UI**
- **Bug fix**
  - http://grails.org/plugin/error-pages-fix, http://grails.org/plugin/aop-reloading-fix
- **Resource-only**
  - famfamfam

Plugin Goals

- **Convention-based approaches**
- **DRY (Don't Repeat Yourself)**
- **Required extension points satisfied**
- **Easy to distribute & install**
  - Without additional configuration

What Can A Plugin Do?

- **Enhance classes at runtime**
  - Add methods, constructors, properties, etc.
- **Perform runtime Spring configuration**
- **Modify web.xml programmatically**
- **Add new controllers, taglibs, services, etc.**
- **Add new artifact types**
- **Support development-mode file and config reloading**
- **Contribute scripts**

A Basic Plugin

```
class LoggingGrailsPlugin {

    def version = "0.1”
    def grailsVersion = "2.0 > *"

    def doWithDynamicMethods = {
        for (c in application.allClasses) {
            c.metaClass.getLog = {->
                LogFactory.getLog(c)
            }
        }
    }
}
```

Application Object

- **'GrailsApplication' object**
  - available with the `application` variable in the plugin descriptor, and the `grailsApplication` Spring bean
  - holds convention information
  - Convenient access to the config: `grailsApplication.config` (since the holders are deprecated)

Application Object

- **Useful methods like:**
  - get*Classes
    - Retrieve all GrailsClass instances for particular artifact, e.g. getControllerClasses()
  - add*Class(Class clazz)
    - Adds new GrailsClass for specified class, e.g. addControllerClass(myClass)
  - get*Class(String name)
    - Retrieves GrailsClass for given name, e.g. getControllerClass(name)

Grails Artifacts

- **Some resource that fulfills a convention**
  - controllers, services, etc.
- **Represented by the GrailsClass interface**
  - http://grails.org/doc/latest/api/org/codehaus/groovy/grails/commons/package-summary.html for API of all artifacts
- **Add new artifacts via the Artifact API**
  - http://grails.org/Developer+-+Artefact+API

Grails Artifacts

- **Design decision: include or generate artifacts?**
  - Include is more automatic, no explicit step
  - Including requires workflow to override
  - Generate requires extra step but is more customizable
  - Generation is static – old files can get out of date
- **Overriding domain classes?**
  - Really only an issue with create-drop and update, not with migrations
- **Overriding controllers?**
  - “un-map” with UrlMappings

```
"/admin/console/$action?/$id?"(controller: "console")
"/console/$action?/$id?"(controller: "errors", action: "urlMapping")
```

Plugin Descriptor

- **Metadata**
  - version, grailsVersion range, pluginExcludes, dependsOn, etc.
- **Lifecycle Closures**

`doWithWebDescriptor`	← Modify XML generated for web.xml at runtime
`doWithSpring`	← Participate in Spring configuration
`doWithApplicationContext`	← Post ApplicationContext initialization activities
`doWithDynamicMethods`	← Add methods to MetaClasses
`onChange`	← Participate in reload events

Configuring Spring

```
class JcrGrailsPlugin {
    def version = 0.1
    def dependsOn = [core:0.4]

    def doWithSpring = {
        jcrRepository(RepositoryFactoryBean) {
            configuration = "classpath:repository.xml"
            homeDir = "/repo"
        }
    }
}
```

Bean name is method name, first argument is bean class

Set properties on the bean

Overriding Spring Beans

- Use **`loadAfter`** to define plugin load order; your beans override other plugins' and Grails'
- resources.groovy loads last, so applications can redefine beans there

```
import com.mycompany.myapp.MyUserDetailsService
beans = {
   userDetailsService(MyUserDetailsService) {
      grailsApplication = ref('grailsApplication')
      foo = 'bar'
   }
}
```

Overriding Spring Beans

- For more advanced configuration, use a **`BeanPostProcessor`**, **`BeanFactoryPostProcessor`**, or a **`BeanDefinitionRegistryPostProcessor`**

Plugging In Dynamic Methods

```
def doWithDynamicMethods = { ctx ->
    application
        .allClasses
        .findAll { it.name.endsWith("Codec") }
        .each {clz ->
            Object
                .metaClass
                ."encodeAs${clz.name-'Codec'}" = {
                    clz.newInstance().encode(delegate)
                }
            }
        }
}
```

Taken from Grails core, this plugin finds all classes ending with “Codec” and dynamically creates “encodeAsFoo” methods!

The “delegate” variable is equivalent to “this” in regular methods

Example Reloading Plugin

```
class I18nGrailsPlugin {
    def version = "0.4.2"
    def watchedResources =
        "file:../grails-app/i18n/*.properties"

    def onChange = { event ->
        def messageSource =
            event.ctx.getBean("messageSource")

        messageSource?.clearCache()
    }
}
```

Defines set of files to watch using Spring Resource pattern

When one changes, event is fired and plugin responds by clearing message cache

The Event Object

- **event.source**
  - Source of the change – either a Spring Resource or a java.lang.Class if the class was reloaded
- **event.ctx**
  - the Spring ApplicationContext
- **event.application**
  - the GrailsApplication
- **event.manager**
  - the GrailsPluginManager

Adding Elements to web.xml

```
def doWithWebDescriptor = { xml →

   def contextParam = xml.'context-param'
   contextParam[contextParam.size() - 1] + {
      'filter' {
         'filter-name'('springSecurityFilterChain')
         'filter-class'(DelegatingFilterProxy.name)
      }
   }

   def filterMapping = xml.'filter-mapping'
   filterMapping[filterMapping.size() - 1] + {
      'listener' {
         'listener-class'(HttpSessionEventPublisher.name)
      }
   }
}
```

Dependency Management

- Jar files
  - Prefer BuildConfig.groovy → Maven repos
  - lib directory is ok if unavailable otherwise
- Other plugins
  - `def loadAfter = ['controllers']`
  - Declare dependencies in BuildConfig.groovy
- Grails
  - `def grailsVersion = '2.0 > *'`

BuildConfig.groovy

```
grails.project.dependency.resolution = {
   inherits 'global'
   log 'warn'

   repositories {
      grailsCentral()

      flatDir name:'myRepo', dirs:'/path/to/repo'

      mavenLocal()
      mavenCentral()

      mavenRepo 'http://download.java.net/maven/2/'
      mavenRepo 'http://my.cool.repo.net/maven/'
   }
```

BuildConfig.groovy

```
dependencies {
   runtime 'mysql:mysql-connector-java:5.1.22'
   compile group: 'commons-httpclient',
                 name: 'commons-httpclient',
                 version: '3.1'
   build('net.sf.ezmorph:ezmorph:1.0.4', 'net.sf.ehcache:ehcache:1.6.1') {
      transitive = false
   }
   test('com.h2database:h2:1.2.144') {
      export = false
   }
   runtime('org.liquibase:liquibase-core:2.0.1',
            'org.hibernate:hibernate-annotations:3.4.0.GA') {
      excludes 'xml-apis', 'commons-logging'
   }
}
```

BuildConfig.groovy

```
   plugins {
      runtime ":hibernate:$grailsVersion", {
         export = false
      }

      runtime ':jquery:1.8.3'
      compile ':spring-security-core:1.2.7.3'
      runtime ':console:1.2'

      build ':release:2.2.1', ':rest-client-builder:1.0.3', {
         export = false
      }
   }
}
```

Troubleshooting Dependency Management

- Run `grails dependency-report`
  - See http://burtbeckwith.com/blog/?p=624 for visualization tips
- Increase BuildConfig.groovy logging level
  - `log 'warn'` → `'info'`, `'verbose'`, or `'debug'`

Scripts

- Gant: http://gant.codehaus.org/
- Groovy + Ant - XML
- Can be used in apps but more common in plugins
- Put in the scripts directory
- _Install.groovy
  - runs when you run `grails install-plugin` or when it's transitively installed - don't overwrite! might be reinstall or plugin upgrade

Scripts

- _Uninstall.groovy
  - runs when you run `grails uninstall-plugin` so you can do cleanup, but don't delete user-created/edited stuff
- _Upgrade.groovy
  - runs when you run `grails upgrade` (not when you upgrade a plugin)

Scripts

- _Events.groovy
  - Respond to build events

```
eventCreateWarStart = { name, stagingDir →
…
}

eventGenerateWebXmlEnd = {
…
}
```

- Naming convention
  - Prefix with '_' → don't show in `grails help`
  - Suffix with '_' → 'global' script, i.e. doesn't need to run in a project dir (e.g. `create-app`)

Scripts

- Reuse existing scripts with includeTargets
  - `includeTargets << grailsScript("_GrailsWar")`
- Include common code in _*Common.groovy

  -
    ```
    includeTargets << new File(
       cloudFoundryPluginDir,
       "scripts/CfCommon.groovy")
    ```

Testing Scripts

- Put tests in test/cli
- Extend `grails.test.AbstractCliTestCase`
- stdout and stderr will be in target/cli-output
- http://www.cacoethes.co.uk/blog/groovyandgrails/testing-your-grails-scripts
- Run with the rest with `grails test-app` or alone with `grails test-app --other`

Scripts

Typical Structure

```
includeTargets << new File(cloudFoundryPluginDir,
          "scripts/_CfCommon.groovy")

USAGE = '''
grails cf-list-files [path] [--appname] [--instance]
'''


target(cfListFiles: 'Display a directory listing') {
   depends cfInit

   // implementation of script
}

def someHelperMethod() { … }

setDefaultTarget cfListFiles
```

Custom Artifacts

- ControllerMixinArtefactHandler extends ArtefactHandlerAdapter
- interface ControllerMixinGrailsClass extends InjectableGrailsClass
- DefaultControllerMixinGrailsClass extends AbstractInjectableGrailsClass implements ControllerMixinGrailsClass

Custom Artifacts

```
class MyGrailsPlugin {

   def watchedResources = [
      'file:./grails-app/controllerMixins/**/*ControllerMixin.groovy',
      'file:./plugins/*/grails-app/controllerMixins/**/*ControllerMixin.groovy'
   ]

   def artefacts = [ControllerMixinArtefactHandler]

   def onChange = { event →
      ...
   }
}
```

Testing

Testing

- Write regular unit and integration tests in test/unit and test/integration
- 'Install' using inline mechanism in BuildConfig.groovy
  - `grails.plugin.location.'my-plugin' = '../my-plugin'`
- Test scripts as described earlier
- Create test apps programmatically – all of the Spring Security plugins do this with bash scripts

Testing

- Use build.xml or Gradle to create single target that tests and builds plugin

```
<target name='package' description='Package the plugin'
        depends='test, doPackage, post-package-cleanup'/>
```

- Use inline plugin or install from zip

```
grails install-plugin /path/to/plugin/grails-myplugin-0.1.zip
```

- Zip install is deprecated, so better to use the maven-install script
  - Will resolve as a BuildConfig dependency (using **`mavenLocal()`**)

Testing

- Use CI, e.g. CloudBees BuildHive:
  - https://fbflex.wordpress.com/2012/07/12/using-cloudbees-buildhive-to-test-grails-plugins/
  - https://buildhive.cloudbees.com/job/burtbeckwith/job/grails-database-session/

Source Control and Release Management

Becoming A Plugin Developer

- **Create an account at grails.org**
- **Discuss your idea on the User mailing list**
  - http://grails.org/Mailing+lists
- Submit a request at http://grails.org/plugins/submitPlugin
  - Monitor the submission for questions and comments
- **Build (and test!) your plugin and run:**
  - `grails publish-plugin --stacktrace`
- **Profit!**

Source Control and Release Management

- **Internal server for jars & plugins is easy with Artifactory or Nexus**

```
repositories {
   grailsPlugins()
   grailsHome()
   mavenRepo "http://yourserver:8081/artifactory/libs-releases-local/"
   mavenRepo "http://yourserver:8081/artifactory/plugins-releases-local/"
   grailsCentral()
}
```

Source Control and Release Management

- **Release with** `grails publish-plugin --repository=repo_name`

```
grails.project.dependency.distribution = {
   remoteRepository(
         id: "repo_name",
         url: "http://yourserver:8081/artifactory/plugins-snapshots-local/") {
      authentication username: "admin", password: "password"
   }
}
```

Source Control and Release Management

- **Use “release” plugin http://grails.org/plugin/release**

```
plugins {
   build ':release:2.2.1', ':rest-client-builder:1.0.3', {
      export = false
   }
}
```

  - Sends tweet from @grailsplugins and email to plugin forum

    http://grails-plugins.847840.n3.nabble.com/

  - Will *not* check code into SVN for you

Want To Learn More?

Best Practices for Experienced Grails Developers

Programming Grails

O'REILLY®

Burt Beckwith

Thanks!