Griffon 0.9

Error rendering macro 'toc' : null

Overview

Griffon 0.9 – "Aquila chrysaetos" - is the fifth major release of Griffon.

This release represents a huge leap in behavior and bug fixes, which warranted a bigger jump in version number. There's still room for improvement but the feature set for 1.0 is almost complete. Additional features may be delivered via plugins rather than having them in core.

New Features

Buildtime

Perhaps the biggest improvement at buildtime is the synchronization of the codebase with Grails 1.3.2, which brings the following benefits:

Project Infrastructure

Groovy 1.7.3 Support

Griffon 0.9 comes with the recently released 1.7.3 version of the Groovy language

JUnit 4

Griffon 0.9 now uses JUnit 4 to run tests. JUnit 4 features a richer assertion API and many features like timeouts, ignores, class level before and after methods, hamcrest matchers, assumptions, theories and more.

Pre Griffon 0.9 tests are fully backwards compatible.

Flexible Build Configuration

A new `griffon-app/conf/BuildConfig.groovy` file is available that allows you to configure different aspects of the Griffon build including output paths and servers used for resolution of plugins:

```
griffon.work.dir="/tmp/work"
griffon.plugins.dir="/usr/local/griffon/plugins"
griffon.project.test.reports.dir="/usr/local/griffon/test-reports"
```

This file replaces `griffon-app/conf/Config.groovy` as the source of buildtime configuration. `Config.groovy` has been promoted to runtime. Configuration set in `BuildConfig.groovy` (which is per application/plugin) can be merged with global settings if `$USER_HOME/.griffon/settings.groovy` is available.

Secured Plugin Repositories

You can now configure secure plugin repositories in `~/.griffon/settings.groovy` or `griffon-app/conf/BuildConfig.groovy` for example:

```
griffon.plugin.repos.discovery.myRepo="https://user01:password01@myserver.com"
```

Project Documentation Engine

The same documentation engine that powers the Griffon reference documentation is now available in your projects. Simply create your documentation inside the src/doc/ref and src/doc/guide directories of your project. See the Griffon documentation source for an example.

Build Event Order

The order of invocation of build event handlers now honors the dependency order of the plugin that provides them. For example the `CompileStart` event handler defined by the lang-bridge plugin will be called before the same event handler provided by the clojure plugin, because the later depends of the former. Previous behavior was to invoke event handlers in alphaetical order of discovery.

Plugins

Multiple PluginRepositories

Griffon now supports the ability to configure multiple plugin repositories by providing a `$USER_HOME/.griffon/settings.groovy` file or a `griffon-app/conf/BuildConfig.groovy` file that contains the configured repository details:

```
griffon.plugin.repos.discovery.myRepository="http://svn.codehaus.org/griffon/trunk/griffon-test-plugin-repo"
griffon.plugin.repos.distribution.myRepository="https://svn.codehaus.org/griffon/trunk/griffon-test-plugin-repo"
```

The Griffon plugin discovery commands like list-plugin and install-plugin will then automatically work against all configured repositories. To release a plugin to a specific repository you can use the repository argument:

```
griffon release-plugin -repository=myRepository
```

Automatic Transitive Plugin Resolution

Plugins no longer need to be checked into SVN and will automatically be installed via a plugins metadata when the application is first loaded.

In addition, plugin dependencies are now resolved transitively.

Modular Application Development with Plugins

An application can now load plugins from anywhere on the file system, even if they have not been installed. Simply add the location of the (unpacked) plugin to you `BuildConfig.groovy` file:

```
// Useful to test plugins you are developing.
griffon.plugin.location.coverflow = "/home/dilbert/dev/plugins/griffon-coverflow"

// Useful for modular applications where all plugins and
// applications are in the same directory.
griffon.plugin.location.'griffon-awesome' = "../griffon-awesome"
```

This is particularly useful in two cases:

- You are developing a plugin and want to test it in a real application without packaging and installing it first.
- You have split an application into a set of plugins and an application, all in the same "super-project" directory.

Licensing and Readmes

A LICENSE.txt file is now mandatory if you intend to release a plugin. Any LICENSE* and README* files located at the basedir of the plugin files will be automatically added to the generated zip file.

Source, Test and Javadoc jars

A source jar will generated for a plugin/addon if any of the following is true:

- additional mvc artifacts are provided
- additional sources (`src/main`) are present
    - you can add more source directories by setting a value for `griffon.plugin.pack.additional.sources` in `griffon-app/conf/BuildConfig.groovy`. The value must be a list of directories relative to the plugin's basedir.

- test sources (`src/test`) are available

A test jar will contain all compiled classes that may have been generated by test sources from `src/test`, including test resources from `test/resources`.

A javadoc jar will be generated from all sources (artifacts, regular, test and additional).

Zip-only plugin releases

If you prefer to use Git/Mercurial/etc. to version control your plugin you can now distribute only the zipped release of the plugin in the central repository and continue to manage the actual sources outside of SVN:

```
griffon release-plugin --zipOnly
```

Running Integration Tests on Addons

Previously to Griffon 0.9 running integration tests on a plugin/addon resulted in an error (missing a `GriffonApplication` instance). However with the availability of a default `MockGriffonApplication` integration testing is now possible.

Dependencies

Dependency Resolution DSL

Griffon 0.9 features a new DSL for configuring JAR dependencies that can be resolved against Maven repositories:

```
griffon.project.dependency.resolution = {
    inherits "global" // inherit Griffon' default dependencies
    repositories {
        griffonHome()
        mavenCentral()
    }
    dependencies {
       runtime 'com.mysql:mysql-connector-java:5.1.5'
    }
}
```

Built on Apache Ivy, users can now explicitly control how Griffon resolves all of its dependencies without needing to use Maven or Apache Ivy directly.

There is also a new command to easily install dependencies into your local cache for use with the DSL:

```
grails install-dependency mysql:mysql-connector-java:5.1.5
```

Maven Repository Support

Griffon now has full support for publishing plugins to (using the Maven Publisher plugin) and reading plugins from Maven compatible repositories.

You can easily configure additional plugin repositories in `BuildConfig.groovy` using the Ivy DSL:

```
repositories {
 mavenRepo "http://repository.codehaus.org"
}
```

The central Griffon repository can also now be easily enabled and disabled by including using the griffonCentral method:

```
repositories {
 griffonCentral()
}
```

Declarative Plugin Dependencies

Alongside the new Maven repository support you can now declare plugin dependencies using the Ivy DSL:

```
plugins {
 runtime ':coverflow:0.3'
}
```

Which allows you to easily control plugin exclusions:

```
plugins {
 runtime( ':transitions:0.8' ) {
  excludes "trident-builder"
 }
}
```

And the scope of a plugin:

```
plugins {
 build( ':clojure:0.8' )
}
```

Testing Infrastructure Improvements

Non JUnit providers

The test running mechanics have been completely overhauled, opening the door to all kinds of testing possibilities for Griffon applications. Previously it was very difficult for non JUnit based tests to be deeply integrated into Griffon (e.g griffon-easyb). Expect to see testing plugins taking advantage of this new infrastructure.

Test phase and type targeting

There is now a more sophisticated mechanism for targeting the exact test you wish to run. Previously, it was only possible to target test phases but it is now also possible to target test types .

You target particular test phases and/or types by using the following syntax:

```
griffon test-app «phase»:«type»
griffon test-app unit:spock    // run ‘spock’ test type in ‘unit’ test phase
```

Either side is optional, and it's absence implies all …

```
griffon test-app unit:    // run all types in ‘unit’ phase
griffon test-app :spock    // run ‘spock’ type in all phases
```

It can be used in conjunction with test targeting…

```
griffon test-app unit: SomeController   // run all test for ‘SomeController’ in the
‘unit’ phase
```

And is additive…

```
griffon test-app unit: integration:    // run all test types in the ‘unit’ and
‘integration’ phases
```

Legacy phase targeting syntax is supported for backwards compatibility

```
griffon test-app --unit    // equivalent to griffon test-app unit:
```

Clean testing

You can can force a clean before testing by passing -clean to test-app:

```
griffon test-app -clean
```

Echoing System.out and System.err

By default, griffon does not show the output from your tests. You can make it do so by passing -echoOut and/or -echoErr to test-app:

```
griffon test-app -echoOut -echoErr
```

Mocking

`griffon.test.mock.MockGriffonApplication` is a fully functional GriffonApplication with the advantage that it lets you override the location of all configuration classes: Application, Builder, Config and Events.

> ⚠ If you choose to change the default UIThreadHandler then you must do it so right after the application has been instantiated and no other operation that requires multi-thread access has been called, otherwise you won't be able to change it's value.

By default, a MockGriffonApplication defines the following:

- MockApplication - setups a 'mock' MVC group with 3 elements: MockModel, MockView and MockController
- MockBuilderConfig - defines a single builder entry: griffon.test.mock.MockBuilder
- MockConfig - defines a single config entry: mocked = true
- MockEvents - defines an event handler for 'Mock'

The remaining classes have these settings:

- MockBuilder - a single node named mock that returns a map with any properties that were defined on the node.
- MockModel - a lone observable property value of type String.
- MockView - simple script that calls the mock node defined by the builder.
- MockController - a controller with no actions.

Packaging

META-INF Resources

Both applications and plugins/addons now support defining resources that should be packaged inside their respective jar's META_INF directory. the directory `griffon-app/conf/metainf` is where all these resources should be placed.

Installer Plugin Integration

Targets coming from the Installer plugin can be used with the `package` command if the plugin is installed, for example

```
griffon package rpm
```

Results in the same output as running

```
griffon prepare-rpm
griffon create-rpm
```

Additional targets are: rpm, deb, mac, windows, jsmooth

GDSL support

Intellij IDEA 9 comes with Groovy DSL support (Mr. Haki explains it well here). Griffon's builtime jar (griffon-cli) includes a GDSL file for all MVC dynamic methods, threading methods and SwingBuilder nodes. Expect additional GDSL files to become available in plugins as new versions are released.

Gradle Inspired Features

Command target expansion

The gradle command supports camel case expansion of a target, this means you can type

```
gradle creP
```

and it will be expanded to 'createPackage' for instance. The Griffon command now sports a similar feature. Creating a new application is as simple as

```
griffon cAp Myapp
```

You'll get a list of options when the expanded script is ambiguous, for example `cA` resolves to `create-app`, `create-addon` and `create-archetype`, so be sure to write enough characters to let the expansion be resolved unambiguously.

Griffon command wrapper

Another interesting feature of gradle is that it ships with a command wrapper (gradlew) that enables a developer to build a project that requires gradle but without having a preinstalled version of gradle. This means the project is self contained in terms of its build. Griffon 0.9 provides a `griffonw` command that works in the same way.

This command will download Griffon from a predefined location and run it, thus enabling developers to ship an application in source form to their

friends and let their friends build the application without needing to installing Griffon in the typical way.

Application Archetypes

While it's true that artifact templates can be provided by plugins it simply was not possible to configure how an application is created. Application Archetypes fill this gap by providing a hook into the application creation process. Archetypes can do the following:

- provide new versions of existing templates, like Model, Controller and so forth
- create new directories and files
- most importantly perhaps, install a preset of plugins

So, if your company requires all applications to be built following the same template and basic behavior then you can create an archetype that enforces those constraints. Archetypes are simple zip files with an application descriptor and templates. Despite this, Griffon provides a few scripts that let you manage archetypes

- **create-archetype**
- **package-archetype**
- **install-archetype**
- **uninstall-archetype**

Archetypes are installed per Griffon location under `$USER_HOME/.griffon/<version>/archetypes`. Archetypes are registered with an application's metadata when creating an application. You can either manually modify the value of 'app.archetype' to a known archetype name or specify an `-archetype=<archetypeName>` flag when creating a new artifact.

```
griffon create-service math -archetype=foo
```

If no valid archetype is found then the `default` archetype will be used.

Runtime

Addons

Node and Method Contributions Expanded

It is now possible to contribute nodes, methods and properties to artifacts other than views using the '*' notation, for example:

```
root.'MyCustomAddon' {
   controller = '*:methods'
   actions = '*'
}
```

results in all explicit methods from MyCustomAddon being added to controllers; all methods, props and nodes being added to actions (if actions is configured as an MVC member).

The following additional qualifiers are available too: '*:methods', '*:factories', '*:props'.

Events

Global Event Handlers

The global event handlers defined in `griffon-app/conf/Events.groovy` are now loaded before any event are fired, including addon events.

Threading

UIThreadHelper Methods on Life-cycle Scripts

You can call all of UIThreadHelper's methods on a life-cycle script using the short notation, the same one available to the application instance and MVC members, i.e. `isUIThread()`, `execSync()`, `execAsync()`, `execOutside()`, `execFuture()`. Refer to the Griffon Guide to know more about the threading options provided by UIThreadHelper.

Application Infrastructure

Application Phase

All applications have the same life-cycle phases. You can inspect in which phase the application is currently on by calling the `getPhase()` metho d on an application instance. Valid values are defined by the `griffon.core.ApplicationPhase` enum : INITIALIZE, STARTUP, READY, MAIN and SHUTDOWN.

Application Locale

All applications sport a bound `java.util.Locale` property whose value is initially the default Locale. You can change this property to let other components be aware of Locale changes as long as they are registered as PropertyChangeListeners on the application instance.

Swing

Shorthand for PropertyChangeListeners

There's a new AST transformation (`@griffon.beans.Listener`) that enables you to write PropertyChangeListeners without all the boilerplate code. The @Listener annotation can be applied to both properties and classes, and it accepts single closures or a List of closures as value. The following example registers two PropertyChangeListeners, the first using a direct closure definition, the second using a property reference found in the same class

```
import griffon.beans.Listener
import groovy.beans.Bindable

class MyModel {
    @Bindable
    @Listener({controller.someAction(it)})
    String name

    @Bindable
    @Listener(myListener)
    String value

    def myListener = { evt -> ... }
}
```

Control Window Display

The `WindowManager` class is responsible for keeping track of all the windows managed by the application. It also controls how these windows are displayed (via a pair of methods: show, hide). WindowManager relies on an instance of `WindowDisplayHandler` to actually show or hide a window. The default implementation simple shows and hide windows directly, however you can change this behavior by setting a different implementation of `WindowDisplayHandler` on the application instance.

The following example shows how you can animate windows using a dropIn effect for show() and a dropOut effect for hide(). This code assumes you have installed the Effects plugin.

In `src/main/Dropper.groovy`

```
import java.awt.Window
import griffon.swing.SwingUtils
import griffon.swing.WindowDisplayHandler
import griffon.core.GriffonApplication
import griffon.effects.Effects

class Dropper implements WindowDisplayHandler {
    void show(Window window, GriffonApplication app) {
        SwingUtils.centerOnScreen(window)
        app.execOutside {
            Effects.dropIn(window, wait: true)
        }
    }

    void hide(Window window, GriffonApplication app) {
        app.execOutside {
            Effects.dropOut(window, wait: true)
        }
    }
}
```

Notice that the effects are executed outside of the UI thread because we need to wait for them to finish before continuing, otherwise we'll hog the UI thread.

The second step to get this example to work is to inform the application it should use Dropper to display/hide windows. This a task that can be easily achieved by adding an application event handler, for example in `griffon-app/conf/Events.groovy`

```
// No windows have been created before this step
onBootstrapEnd = { app ->
    app.windowDisplayHandler = new Dropper()
}
```

Breaking changes

Removed deprecated classes and methods

- class: griffon.util.IGriffonApplication
- class: griffon.applet.GriffonApplet
- class: griffon.application.SingleFrameApplication
- method: GriffonApplication.getApplicationProperties
- method: BaseGriffonApplication.getApplicationProperties
- method: BaseGriffonApplication.setApplicationProperties
- method: BaseGriffonApplication.loadApplicationProperties
- method: GriffonApplicationHelper.runScriptInsideEDT

Moved classes

`Buildtime`

- from `org.codehaus.griffon.util` to `griffon.util`
    - BuildSettings
    - BuildeSettingsHolder
    - GriffonUtil
      Rationale: Classes under packages `griffon.*` form part of the supported API and are visible to application developers. Classes under packages `org.codehaus.griffon.*` are considered internal details, they may change in the future without

notice.

Runtime

- from `griffon.applet` to `griffon.swing`
  - SwingApplet
- from `griffon.application` to `griffon.swing`
  - SwingApplication
- from `griffon.util` to `org.codehaus.griffon.runtime.util`
  - AddonHelper
  - CompositeBuilderHelper
  - GriffonApplicationHelper
    Rationale: Classes under `griffon.swing` prepare Griffon core for a future Swing plugin. Classes under `org.codehaus.griffon.runtime.util` should be used by the application's runtime only. Also, classes under packages `griffon.*` form part of the supported API and are visible to application developers. Classes under packages `org.codehaus.griffon.*` are considered internal details, they may change in the future without notice.
- Also moved @EventPublisher and EventPublisherASTTransformation to `Buildtime/org.codehaus.griffon.ast` as having a runtime dependency for this AST transformation make no sense.

Moved methods

- GriffonApplicationHelper.createJFrameApplication => SwingUtils.createApplicationFrame
- GriffonNameUtils.getScriptName => GriffonUtil.getScriptName
- GriffonNameUtils.getNameFromScript => GriffonUtil.getNameFromScript
- GriffonNameUtils.getPluginName => GriffonUtil.getPluginName

Removed properties

- `app.appFrames` is no longer available. Use `app.windowManager.windows` instead.

Sample Applications

Griffon 0.9 ships with sample applications of varying levels of complexity demonstrating various parts of the framework. In order of complexity they are:

File Viewer

File View is a simple demonstration of creating new MVCGroups on the fly.

Source: git

To run the sample from source, change into the source directory and run `griffon run-app` from the command prompt.

Font Picker

Font Picker demonstrates form based data binding to adjust the sample rendering of system fonts.

Source: git

To run the sample from source, change into the source directory and run `griffon run-app` from the command prompt.

Greet

Greet, a full featured Griffon Application, is a Twitter client. It shows Joint Java/Groovy compilation, richer MVCGroup interactions, and network service based data delivery.

Source: git

To run the sample from source, change into the source directory and run `griffon run-webstart` from the command prompt. Because Greet uses JNLP APIs for browser integration using `run-app` will prevent web links from working.

SwingPad

SwingPad, a full featured Griffon Application, is a scripting console for rendering Groovy SwingBuilder views.

0.9 Release Notes

Griffon 0.9 Resolved Issues (51 issues)

T	Key	Summary
	GRIFFON-207	"griffon run-app" gives java.lang.UnsatisfiedLinkError after a fresh install and creation of app
	GRIFFON-216	Can't run an application on Linux
	GRIFFON-121	Rearrange config files
	GRIFFON-215	Sample apps do not run with 0.9-SNAPSHOT
	GRIFFON-83	Integration tests do not initialize the application fully
	GRIFFON-89	Update core dependencies
	GRIFFON-107	Upgrade to JUnit 4
	GRIFFON-133	Desktop icons look really ugly on Ubuntu
	GRIFFON-141	Port AntBuildListener from Grails
	GRIFFON-142	Upgrade testing facilities
	GRIFFON-143	Allow plugins to be released with zip-only flag
	GRIFFON-145	Remove dead code copied from Grails
	GRIFFON-148	Events scripts provided by plugins should honor plugin dependency order
	GRIFFON-155	Command-line takes, but does not honor, --non-interactive
	GRIFFON-159	Allow addons to contribute methods/properties to other artifacts
	GRIFFON-172	Dock icon looks ugly
	GRIFFON-174	Add a GDSL for all Swing, Threading and MVC nodes and methods
	GRIFFON-177	Allow javaOpts to be specified on the command line when executing run-app
	GRIFFON-181	Include LICENSE and README files when packaging plugins
	GRIFFON-182	Exclude *GriffonPlugin.class from application jar file

Showing 20 out of 51 issues