# Griffon Guide - Reference Documentation

**Authors:** Andres Almiray

**Version:** 0.9.5-rc2

*Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.*

## Table of Contents

# 1. Introduction

Developing desktop/RIA applications on the JVM is a hard task. You have to make choices upfront during application design that might complicate the implementation, compromising the user experience; not to mention the amount of configuration needed.
RCP solutions like Eclipse RCP and NetBeans RCP are great for developing desktop applications, not so much for RIAs and applets. Griffon is a framework inspired by Grails, whose aim is to overcome the problems you may encounter while developing all these types of applications. It brings along popular concepts like

- Convention over Configuration
- Don't Repeat Yourself (DRY)
- Pervasive MVC
- Task automation
- Testing supported "out of the box"

Griffon relies on the power of the Groovy language to glue all things together. The framework is quite extensible via plugins also.
This documentation will take you through getting started with Griffon and building desktop/RIA applications with the Griffon framework.

**Credits and Acknowledgements**
This guide is heavily influenced by the Grails Guide. It simply would not have been possible without the great efforts made by: Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown and their sponsor: SpringSource. The Griffon team would like to thank them all (and the Grails community too!) for making such a great framework and bringing the fun back to programming applications.

# 2. Getting Started

## 2.1 Downloading and Installing

The first step to getting up and running with Griffon is to install the distribution. To do so follow these steps:

- Download a binary distribution of Griffon and extract the resulting zip file to a location of your choice
- Set the GRIFFON_HOME environment variable to the location where you extracted the zip
  - On Unix/Linux based systems this is typically a matter of adding something like the following `export GRIFFON_HOME=/path/to/griffon` to your profile
  - On Windows this is typically a matter of setting an environment variable under `My Computer/Advanced/Environment Variables`
- Now you need to add the `bin` directory to your `PATH` variable:
  - On Unix/Linux base system this can be done by doing a `export PATH="$PATH:$GRIFFON_HOME/bin"`
  - On windows this is done by modifying the `Path` environment variable under `My Computer/Advanced/Environment Variables`

If Griffon is working correctly you should now be able to type `griffon` in the terminal window and see output similar to the below:

```
Welcome to Griffon 0.9.5-rc2 - http://griffon.codehaus.org/
Licensed under Apache Standard License 2.0
Griffon home is set to: /usr/local/griffon-0.9.5-rc2
No script name specified. Use 'griffon help' for more info
```

## 2.2 Creating an Application

To create a Griffon application you first need to familiarize yourself with the usage of the `griffon` command which is used in the following manner:

```
griffon [command name]
```

In this case the command you need to execute is create-app:

```
griffon create-app demoConsole
```

This will create a new directory inside the current one that contains the project. You should now navigate to this directory in terminal:

```
cd demoConsole
```

## 2.3 A Groovy Console Example

The "create-app" target created a Griffon MVC Triad for you in the models, views, and controllers directory named after the application. Hence you already have a model class DemoConsoleModel in the models directory.
The application model for this application is simple: it contains properties that hold the script to be evaluated and the results of the evaluation. Make sure you paste the following code into `griffon-app/models/democonsole/DemoConsoleModel.groovy`.

```
package democonsole
import groovy.beans.Bindable
class DemoConsoleModel {
    String scriptSource
    @Bindable def scriptResult
    @Bindable boolean enabled = true
}
```

The controller is also trivial: throw the contents of the script from the model at a groovy shell, then store the result back into the model. Make sure you paste the following code into `griffon-app/controllers/democonsole/DemoConsoleController.groovy`.

```
package democonsole
class DemoConsoleController {
    private GroovyShell shell = new GroovyShell()
    // these will be injected by Griffon
    def model
    def view
    def executeScript = { evt = null ->
        model.enabled = false
        def result
        try {
            result = shell.evaluate(model.scriptSource)
        } finally {
            model.enabled = true
            model.scriptResult = result
        }
    }
}
```

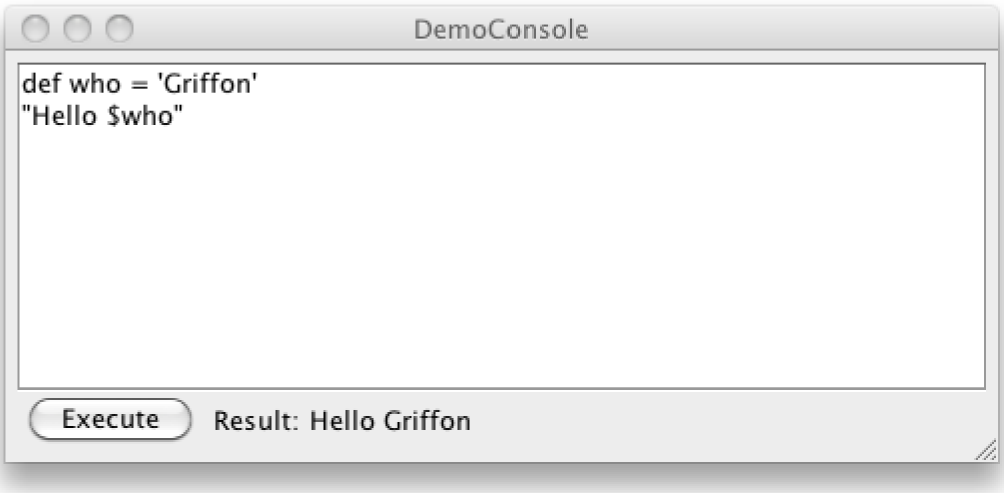
The Griffon framework will inject references to the other portions of the MVC triad if fields named model, view, and controller are present in the model or controller. This allows us to access the view widgets and the model data if needed
The executeScript method will be used in the view for the button action. Hence the `evt` parameter, and the default value so it can be called without an action event.
Finally, the Griffon framework can be configured to inject portions of the builders it uses. By default, the Threading classes are injected into the controller, allowing the use of the `edt`, `doOutside` and `doLater` methods from SwingBuilder.
You may notice that there's no explicit threading management. All Swing developers know they must obey the Swing Rule: long running computations must run outside of the EDT; all UI components should be queried/modified inside the EDT. It turns out Griffon is aware of this rule, making sure an action is called outside of the EDt by default, all bindings made to UI components via the model will be updated inside the EDT also. We'll setup the bindings in the next example.
The view classes contain the visual components for your application. Please paste the following code into `griffon-app/views/democonsole/DemoConsoleView.groovy`.

```
package democonsole
application(title:'DemoConsole', pack:true,
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.png').image,
  iconImages: [imageIcon('/griffon-icon-48x48.png').image,
               imageIcon('/griffon-icon-32x32.png').image,
               imageIcon('/griffon-icon-16x16.png').image]) {
    panel(border:emptyBorder(6)) {
        borderLayout()
        scrollPane(constraints:CENTER) {
            textArea(text:bind(target: model, 'scriptSource'),
                enabled: bind { model.enabled },
                columns: 40, rows: 10)
        }
        hbox(constraints:SOUTH) {
            button("Execute", actionPerformed: controller.executeScript,
                enabled: bind { model.enabled })
            hstrut 5
            label 'Result:'
            hstrut 5
            label text: bind { model.scriptResult }
        }
    }
}
```

The view script is a fairly straightforward SwingBuilder script. Griffon will execute these groovy scripts in context of it's UberBuilder (a composite of the SwingBuilder and whatever else is thrown in).
Now to get the application running. You can do this by calling the run-app command:

```
griffon run-app
```

This command should compile all sources and package the application, you'll see a similar result as depicted by the following screenshot after a few seconds



```
def who = 'Griffon'
"Hello $who"
```

Standalone mode is not the only way to run your application, try the following command to run it in webstart mode: run-webstart. Conversely run-applet will run your application in applet mode. The best of all is that you did not have to touch a single line of configuration in order to switch modes!

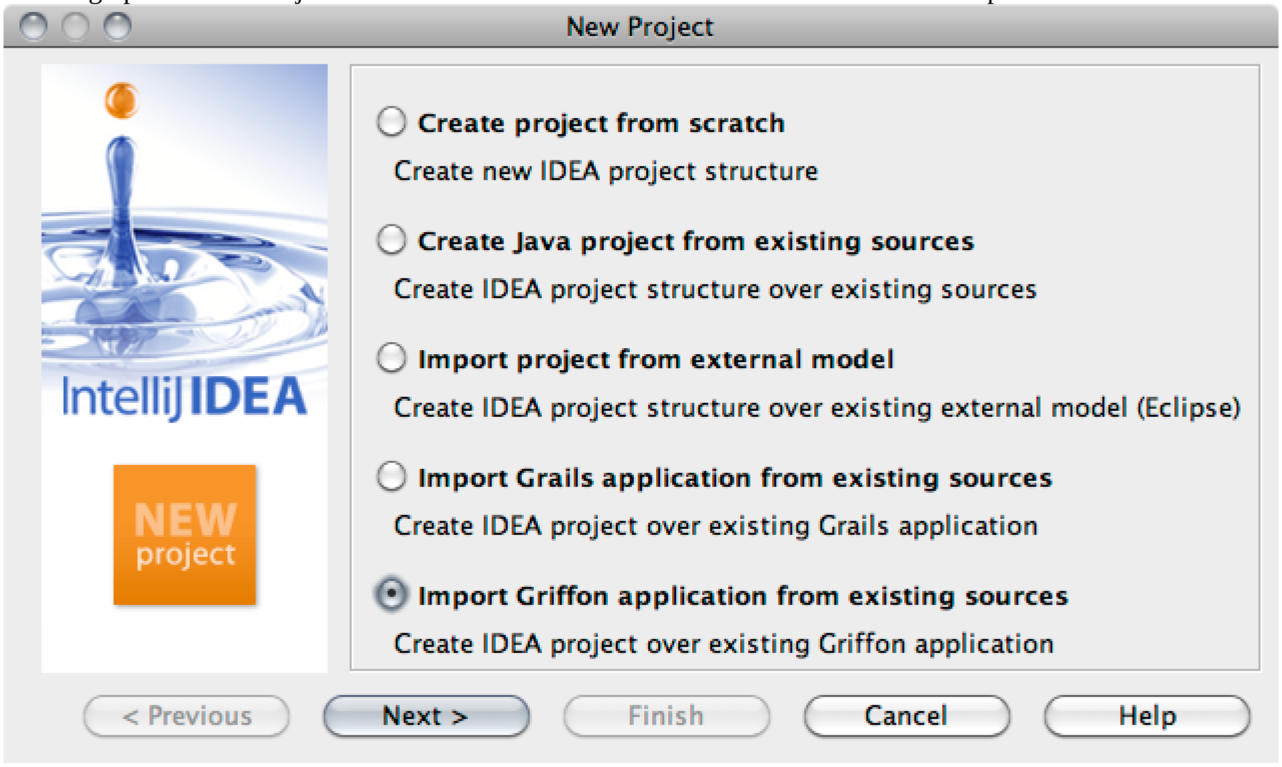
## 2.4 Getting Set-up in an IDE

**IntelliJ IDEA**
IntelliJ IDEA and the JetGroovy plug-in offer good support for Groovy/Grails/Griffon developers. Refer to the section on Groovy and Grails support on the JetBrains website for a feature overview.

**Integrating an existing Griffon project**
To integrate Griffon with IntelliJ run the following command to generate appropriate project files:

```
griffon integrate-with --intellij
```

**Creating a new Griffon project**
Follow these steps to create and run a new Griffon project with IDEA
**#1** Bring up the "New Project" wizard. You should see Griffon as one of the available options


New Project

( ) **Create project from scratch**
Create new IDEA project structure

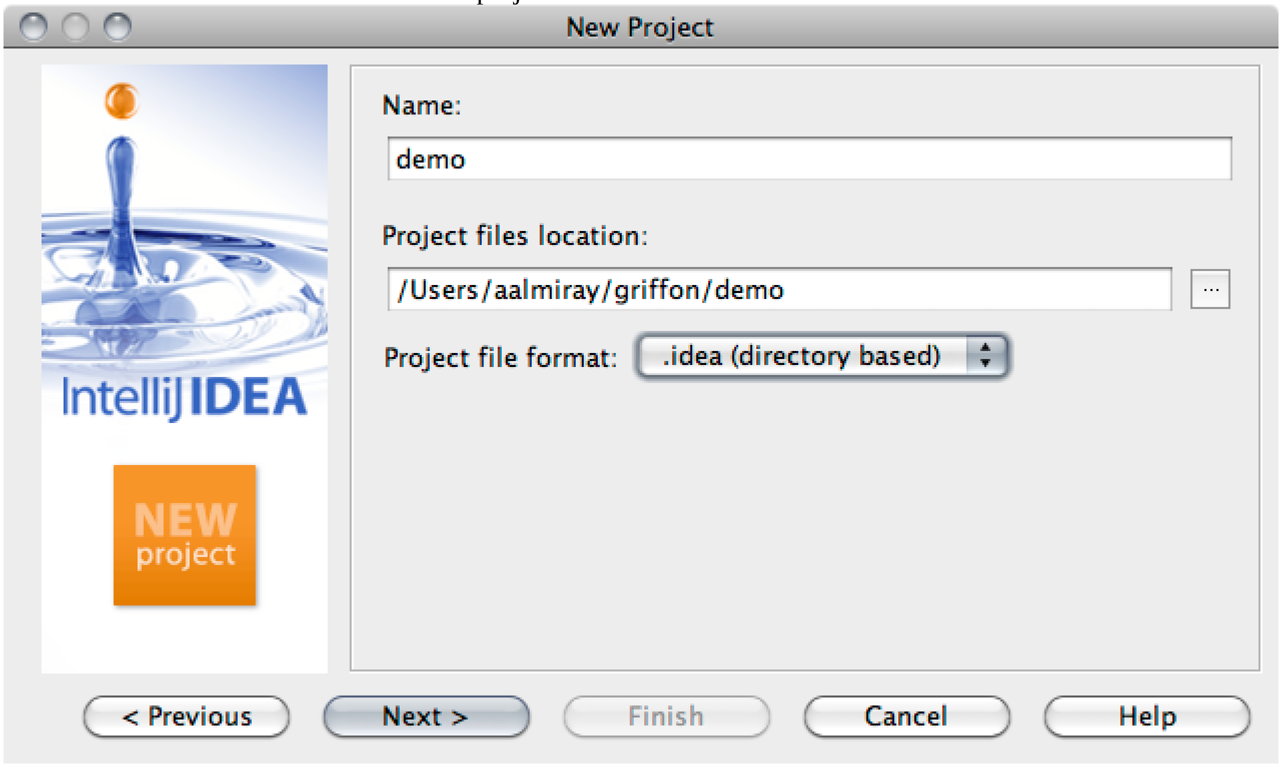
( ) **Create Java project from existing sources**
Create IDEA project structure over existing sources

( ) **Import project from external model**
Create IDEA project structure over existing external model (Eclipse)

( ) **Import Grails application from existing sources**
Create IDEA project over existing Grails application

[✓ **Import Griffon application from existing sources**]
Create IDEA project over existing Griffon application

**#2** Choose name and location for the new project

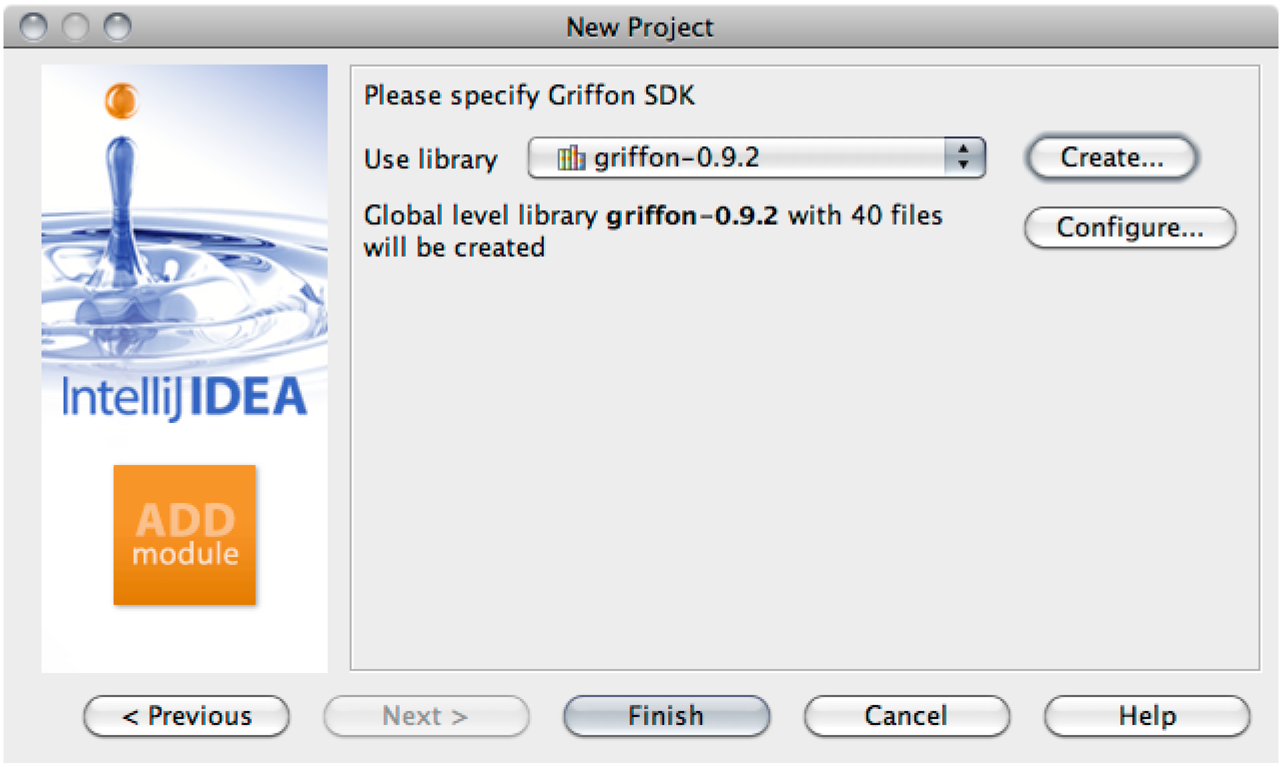

New Project

Name: demo

Project files location: /Users/aalmiray/griffon/demo

Project file format: .idea (directory based)

**#3** Configure a Griffon SDK if you haven't done so already

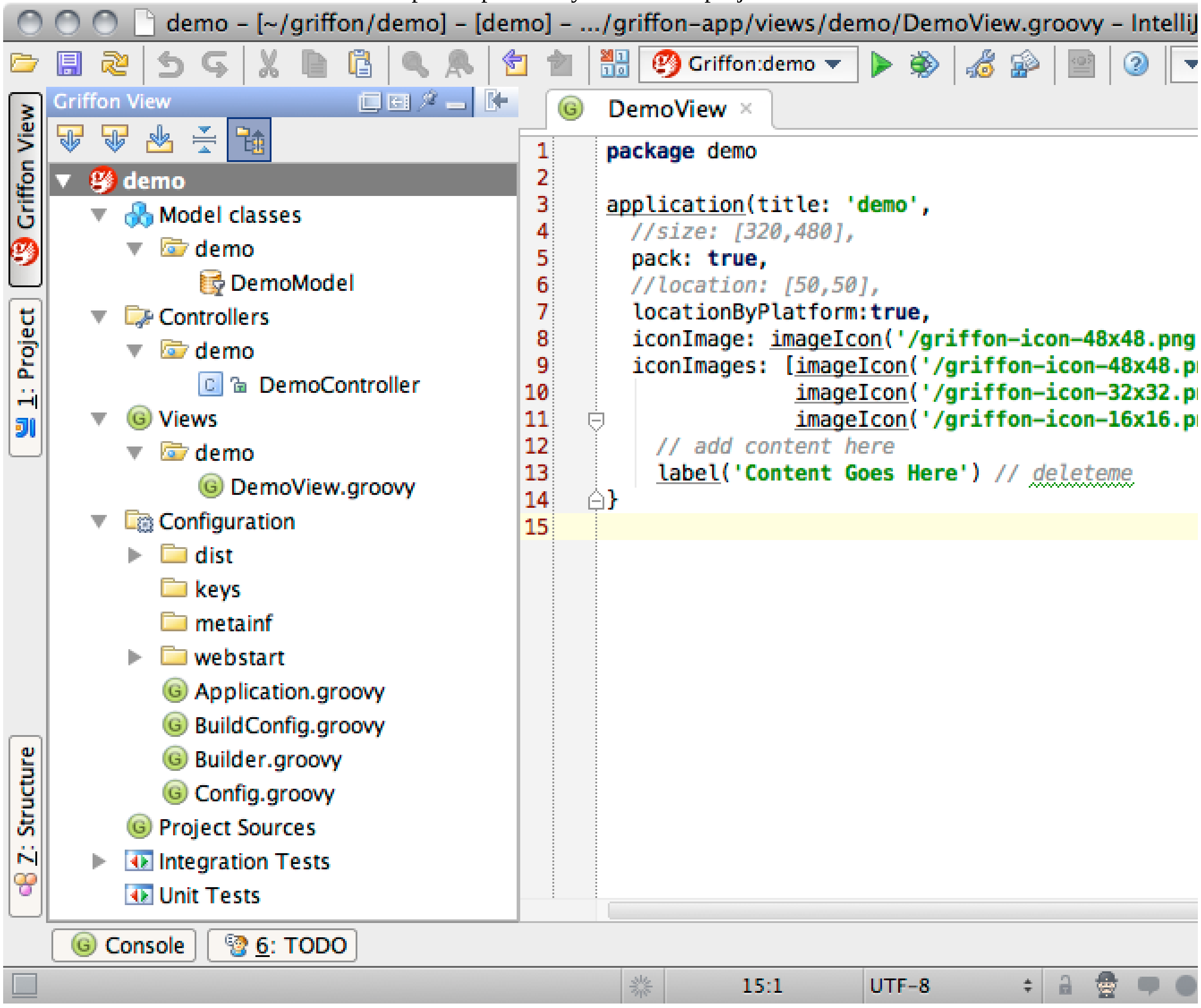


Please specify Griffon SDK

Use library: griffon-0.9.2

Global level library **griffon-0.9.2** with 40 files will be created

**#4** Click on the Finish button and develop with pleasure your Griffon project

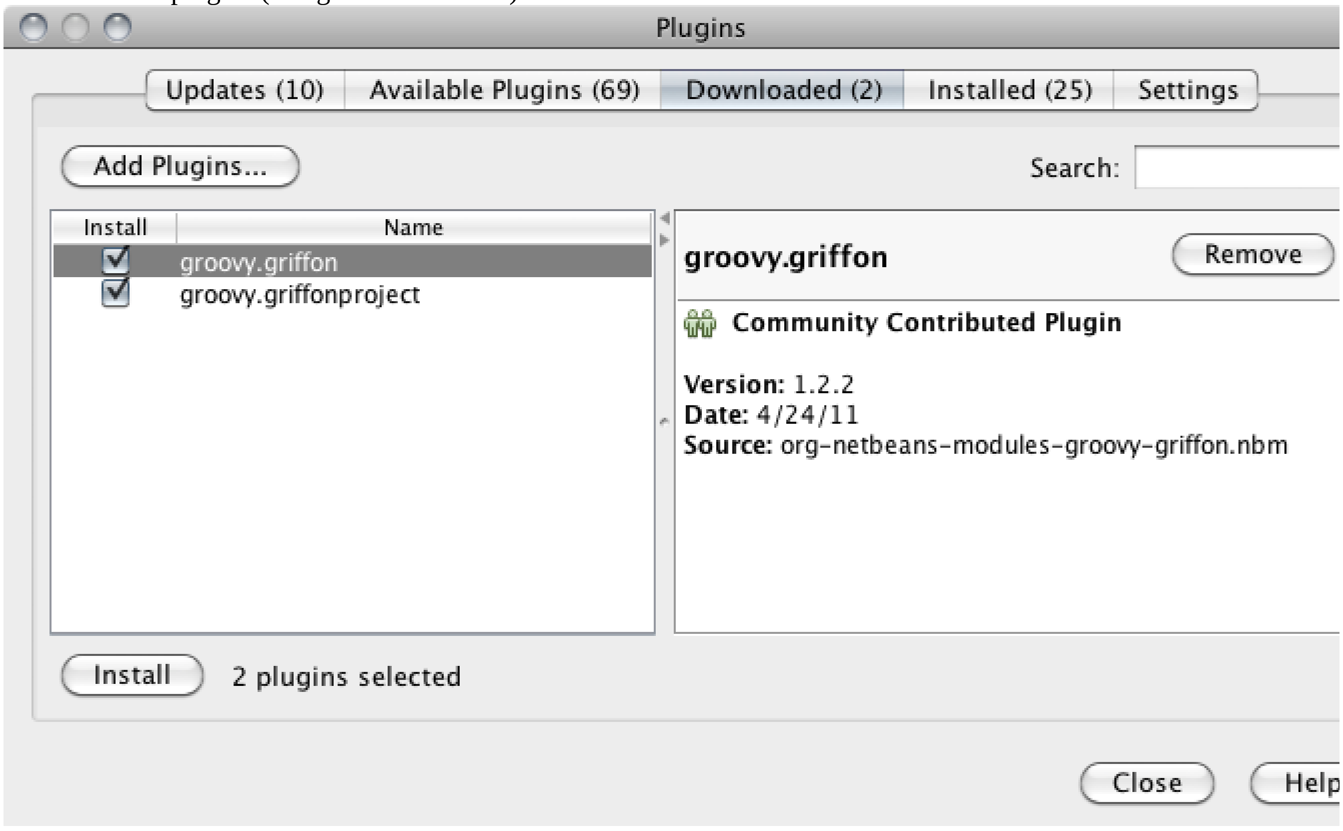


```
package demo

application(title: 'demo',
  //size: [320,480],
  pack: true,
  //location: [50,50],
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.png
  iconImages: [imageIcon('/griffon-icon-48x48.p
               imageIcon('/griffon-icon-32x32.p
               imageIcon('/griffon-icon-16x16.p
    // add content here
    label('Content Goes Here') // deleteme
}
```

**NetBeans**

A good Open Source alternative is Oracle's NetBeans, which provides a Groovy/Griffon plugin that automatically recognizes Griffon projects and provides the ability to run Griffon applications in the IDE, code completion and integration with Oracle's Glassfish server.

**Integrating an existing Griffon project**
NetBeans does not require any special integration support, it understands the layout of a Griffon project as long as the Griffon plugin is installed. Just select "Open" from the menu and locate the folder that contains your project. It's that simple. Follow these steps to install the Griffon NetBeans plugin.
**Prerequisites**: Java, Groovy and Grails plugins installed and up to date.
**#1** Download the plugin
Follow this link to download the latest zip distribution of the plugin.
**#2** Unpack the zip file into a directory of your choosing
**#3** Open the plugin manager dialog. Go to the "Downloaded" tab, then click on the "Add Plugins..." button. Locate and select the NBM files that were uncompressed in the previous step.
**#4** Select both plugins (using the checkboxes) and click on "Install".



| Install | Name |
|---|---|
| ✓ | groovy.griffon |
| ✓ | groovy.griffonproject |

groovy.griffon

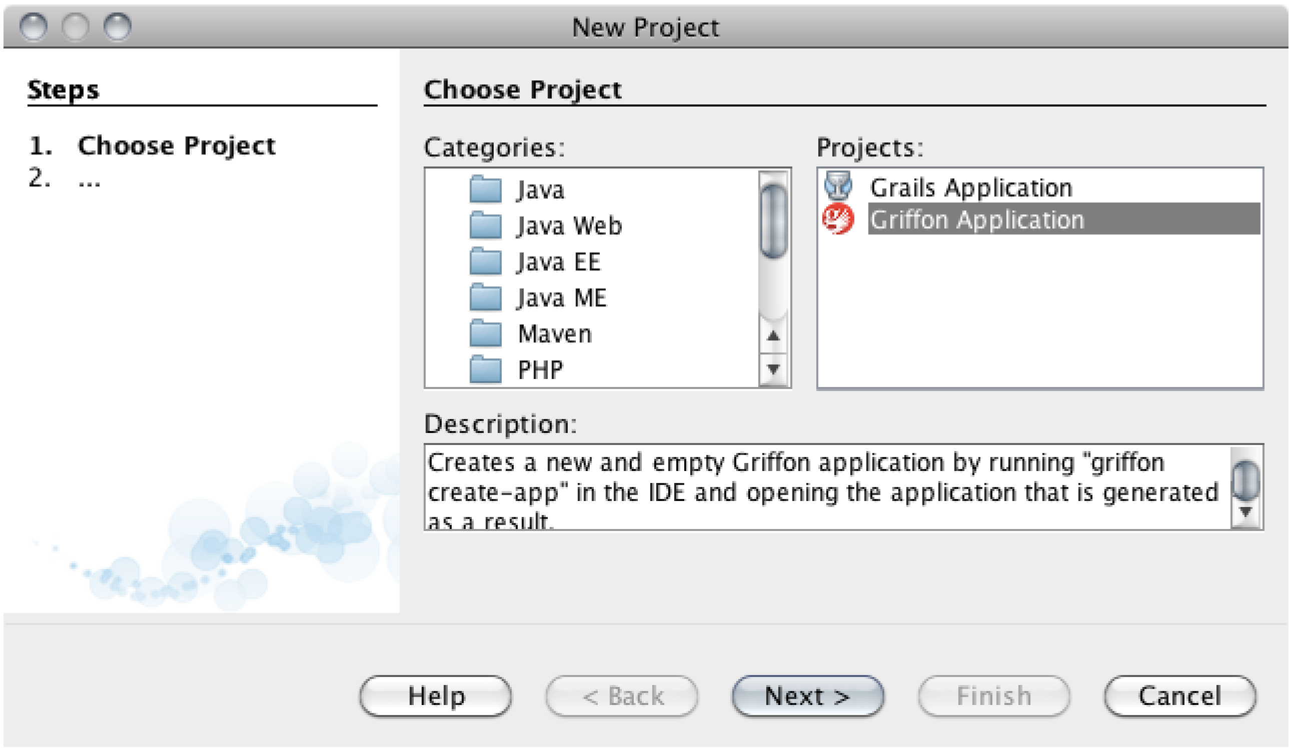
Community Contributed Plugin

**Version:** 1.2.2
**Date:** 4/24/11
**Source:** org-netbeans-modules-groovy-griffon.nbm

2 plugins selected

**#5** Restart your IDE and enjoy!

**Creating a new Griffon project**
**Prerequisites**: You must have the Griffon plugin installed. Follow the steps explained in the previous section to get the job done.
**#1** Bring up the "New Project" wizard. Click on "Groovy" then on "Griffon Application".


New Project

**Steps**

1. **Choose Project**
2. ...

**Choose Project**

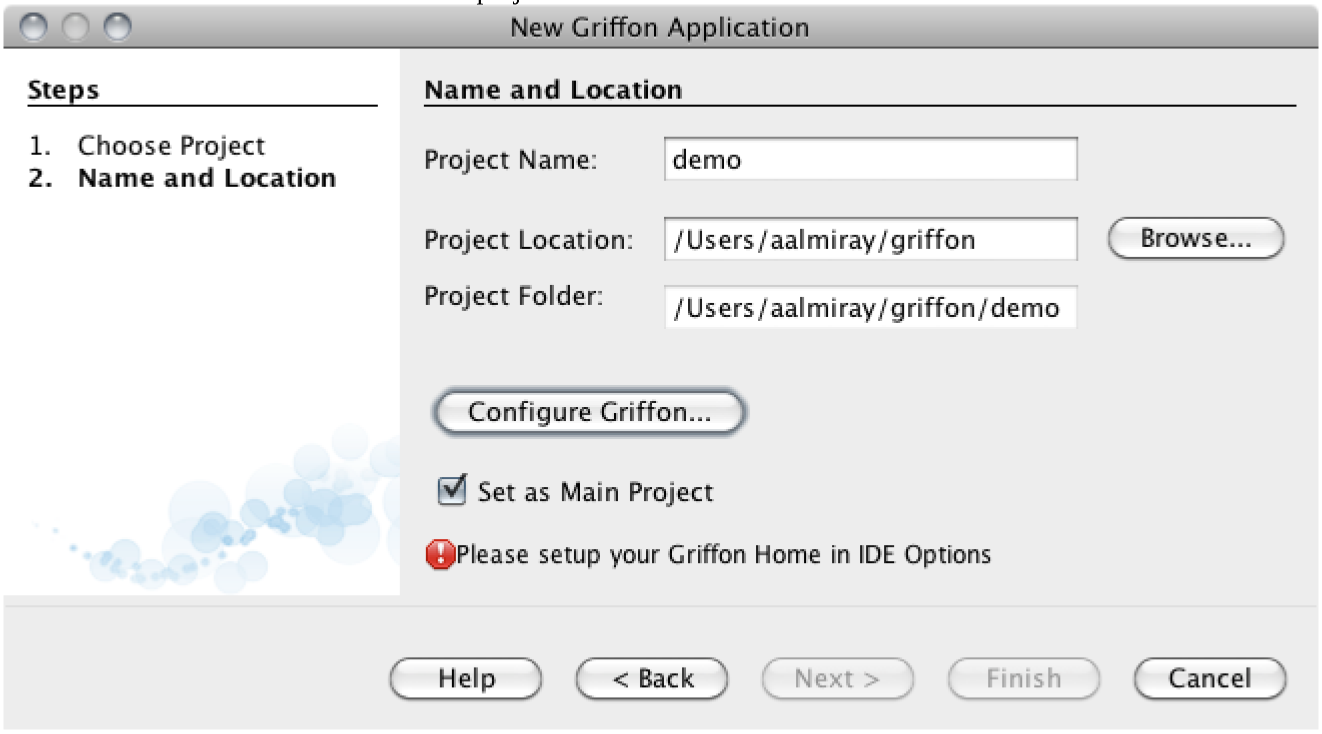
Categories: Java, Java Web, Java EE, Java ME, Maven, PHP

Projects: Grails Application, Griffon Application

Description:

Creates a new and empty Griffon application by running "griffon create-app" in the IDE and opening the application that is generated as a result.

**#2** Choose name and location for the new project


New Griffon Application

**Steps**

1. Choose Project
2. **Name and Location**

**Name and Location**

Project Name: demo

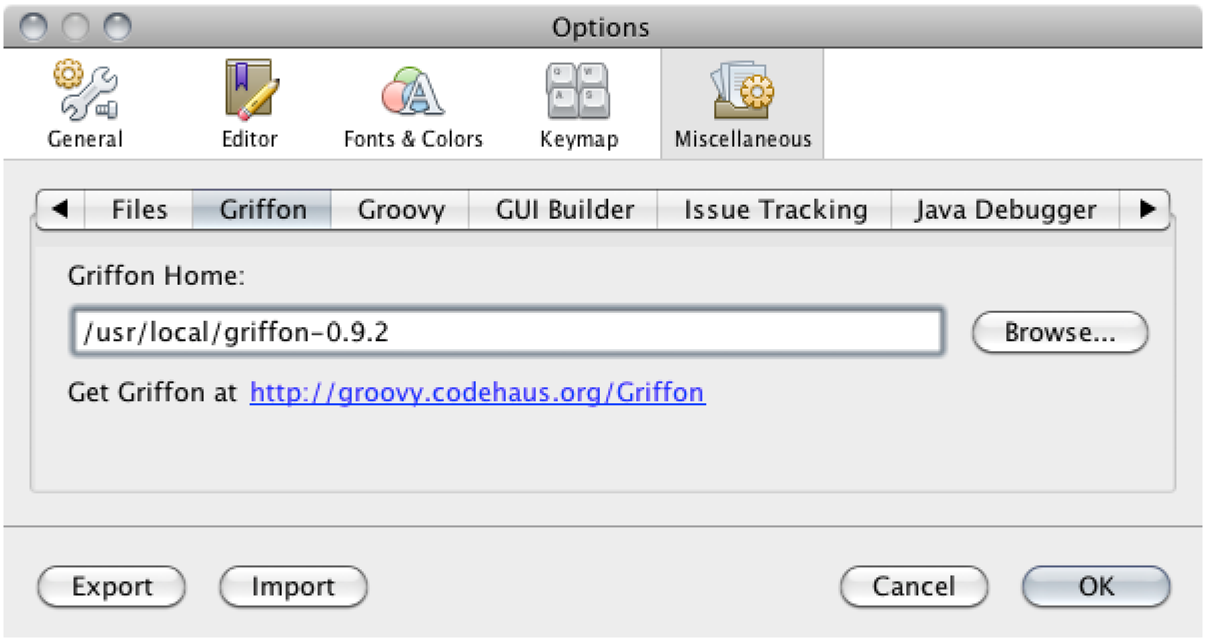
Project Location: /Users/aalmiray/griffon

Project Folder: /Users/aalmiray/griffon/demo

☑ Set as Main Project

Please setup your Griffon Home in IDE Options

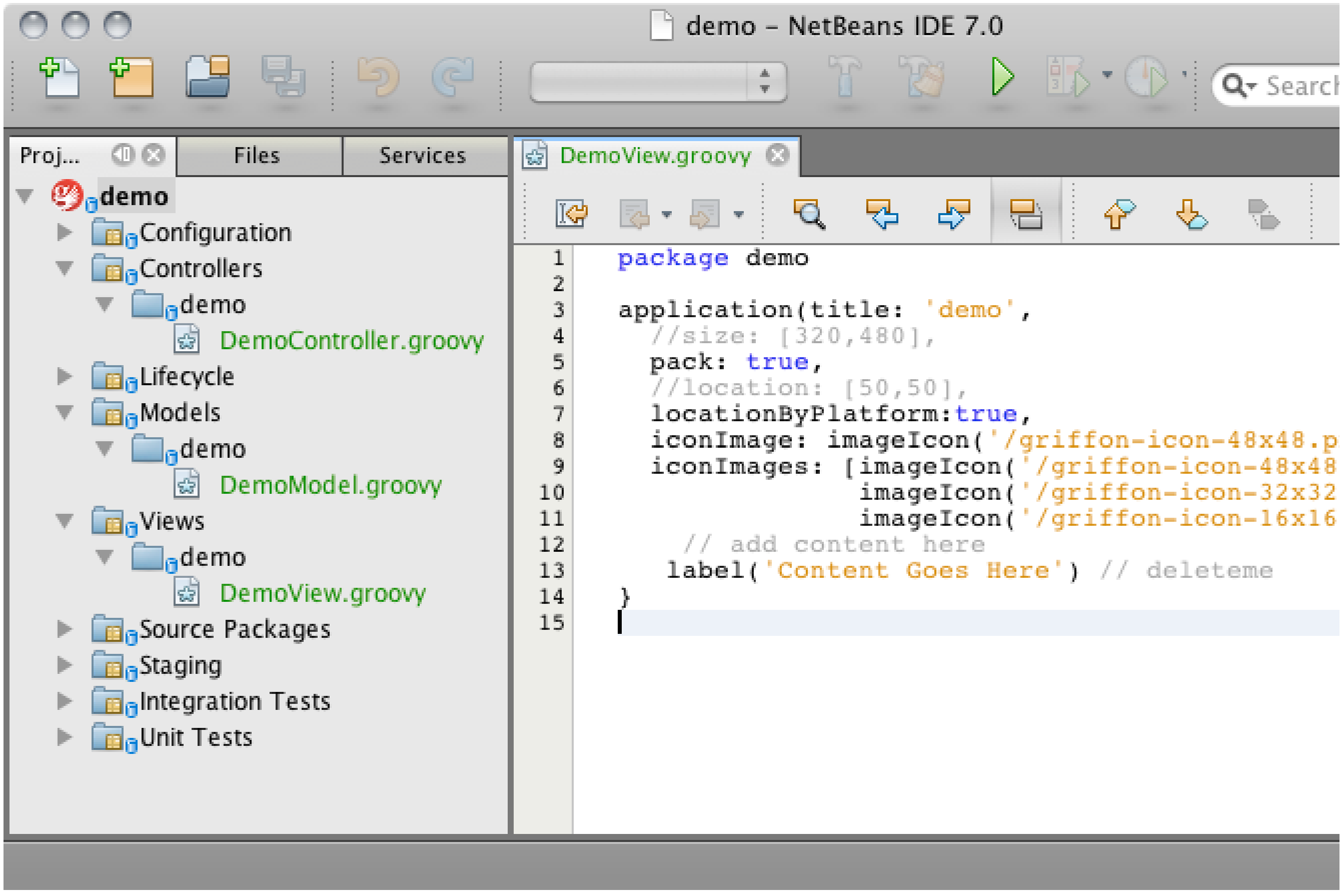
**#3** Configure a Griffon SDK if you haven't done so already



Griffon Home: /usr/local/griffon-0.9.2

Get Griffon at http://groovy.codehaus.org/Griffon

**#4** Click on the Finish button



```
package demo

application(title: 'demo',
  //size: [320,480],
  pack: true,
  //location: [50,50],
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.p
  iconImages: [imageIcon('/griffon-icon-48x48
               imageIcon('/griffon-icon-32x32
               imageIcon('/griffon-icon-16x16
    // add content here
   label('Content Goes Here') // deleteme
}
```

**Eclipse**

We recommend that users of Eclipse looking to develop Griffon application take a look at SpringSource Tool Suite, which offers built in support for Groovy.

**Integrating an existing Griffon project**

To integrate Griffon with Eclipse run the following command to generate appropriate project files:

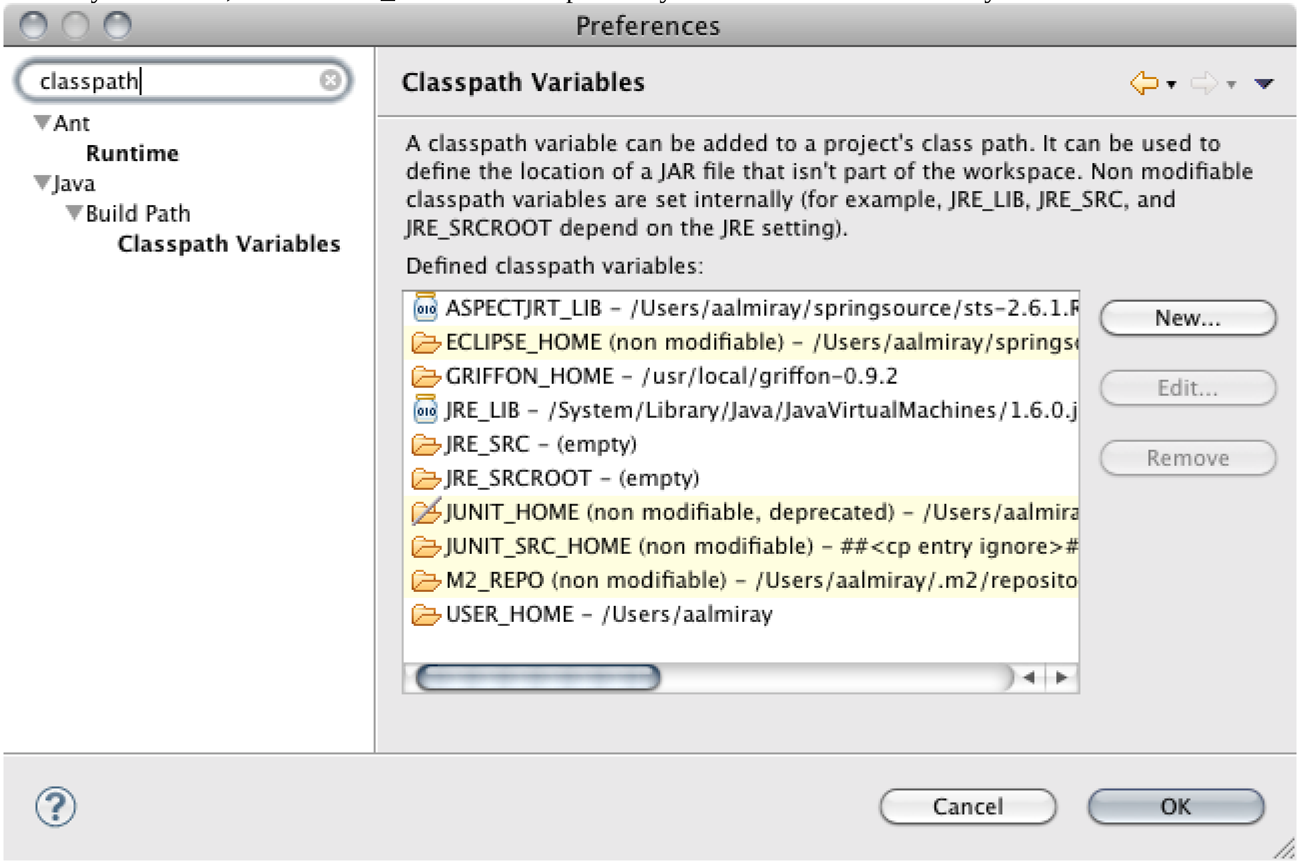
```
griffon integrate-with --eclipse
```

Then follow these steps to fully integrate and run the application

**#1** Install the Eclipse Support plugin

```
griffon install-plugin eclipse-support
```

**#2** Configure a pair Classpath User Variables in the preferences dialog. GRIFFON_HOME should point to the install directory of Griffon, while USER_HOME should point to your account's home directory.



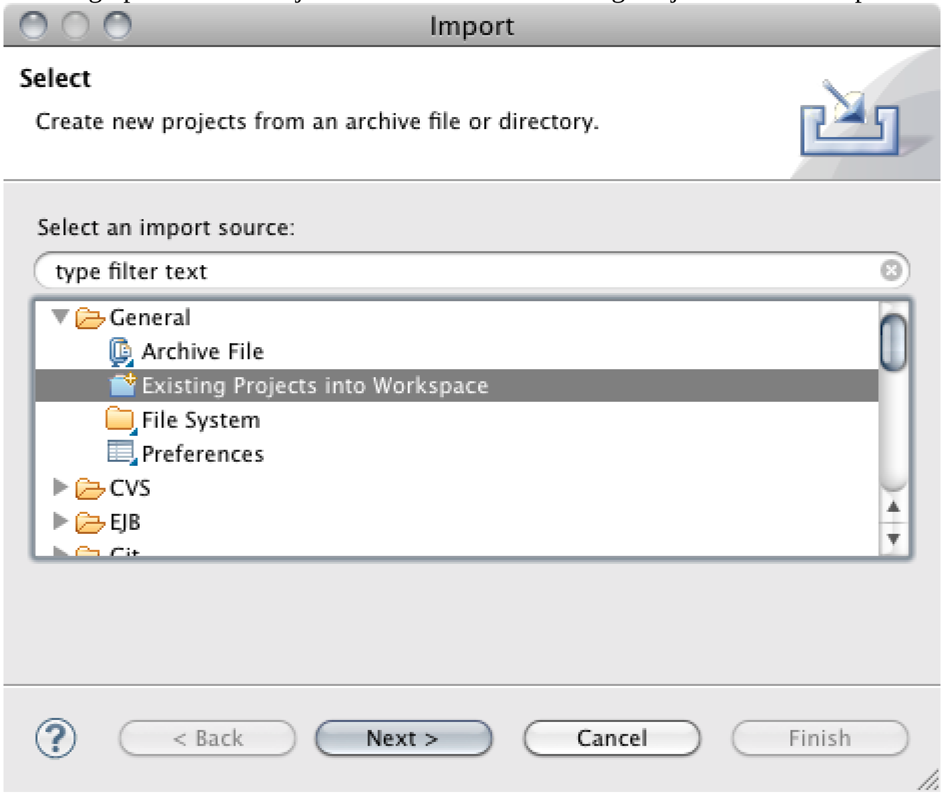
Classpath Variables

A classpath variable can be added to a project's class path. It can be used to define the location of a JAR file that isn't part of the workspace. Non modifiable classpath variables are set internally (for example, JRE_LIB, JRE_SRC, and JRE_SRCROOT depend on the JRE setting).

Defined classpath variables:

ASPECTJRT_LIB - /Users/aalmiray/springsource/sts-2.6.1.R
ECLIPSE_HOME (non modifiable) - /Users/aalmiray/springs
GRIFFON_HOME - /usr/local/griffon-0.9.2
JRE_LIB - /System/Library/Java/JavaVirtualMachines/1.6.0.j
JRE_SRC - (empty)
JRE_SRCROOT - (empty)
JUNIT_HOME (non modifiable, deprecated) - /Users/aalmira
JUNIT_SRC_HOME (non modifiable) - ##<cp entry ignore>#
M2_REPO (non modifiable) - /Users/aalmiray/.m2/reposito
USER_HOME - /Users/aalmiray

**#3** Bring up the "New Project" wizard. Select "Existing Projects into Workspace"

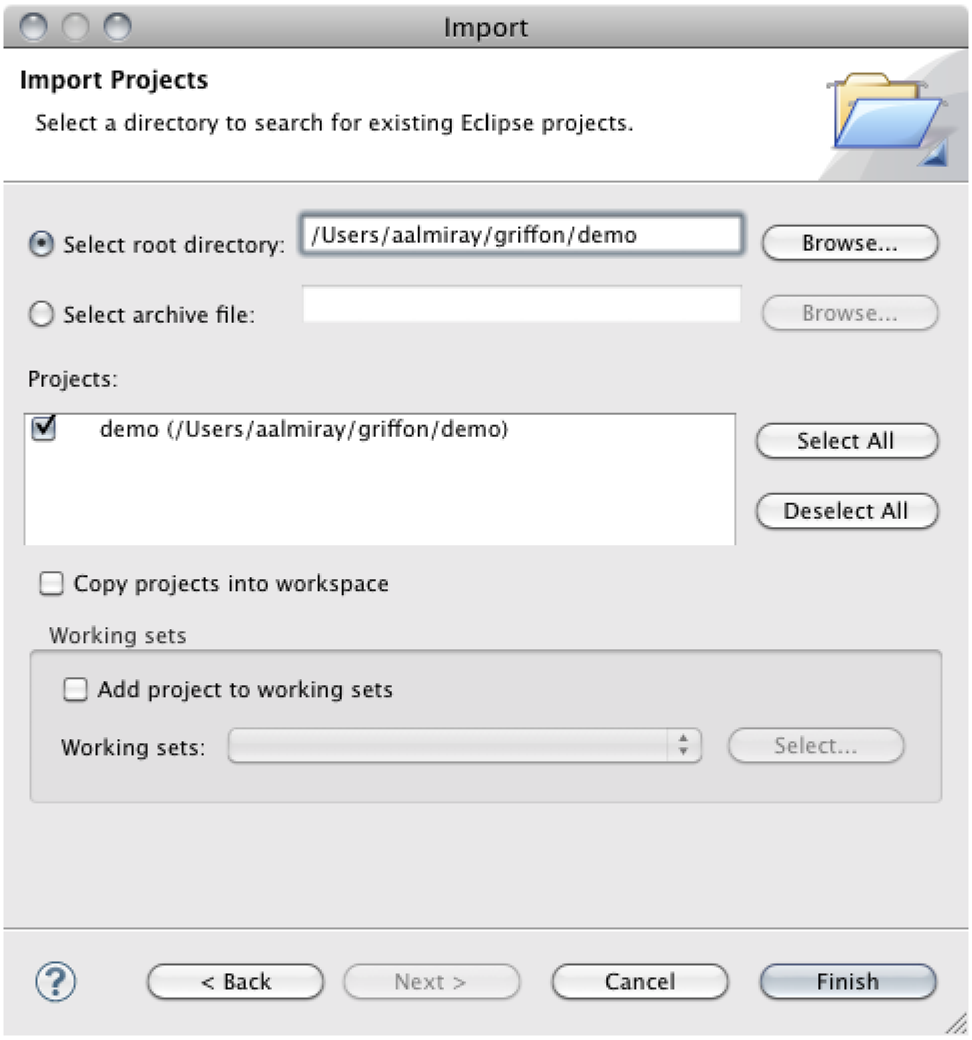


Select

Create new projects from an archive file or directory.

Select an import source:

**#4** Select the directory of the application that contains .project/.classpath files



**Import Projects**

Select a directory to search for existing Eclipse projects.

[✓ Select root directory:] /Users/aalmiray/griffon/demo

Select archive file:

Projects:

- [✓] demo (/Users/aalmiray/griffon/demo)

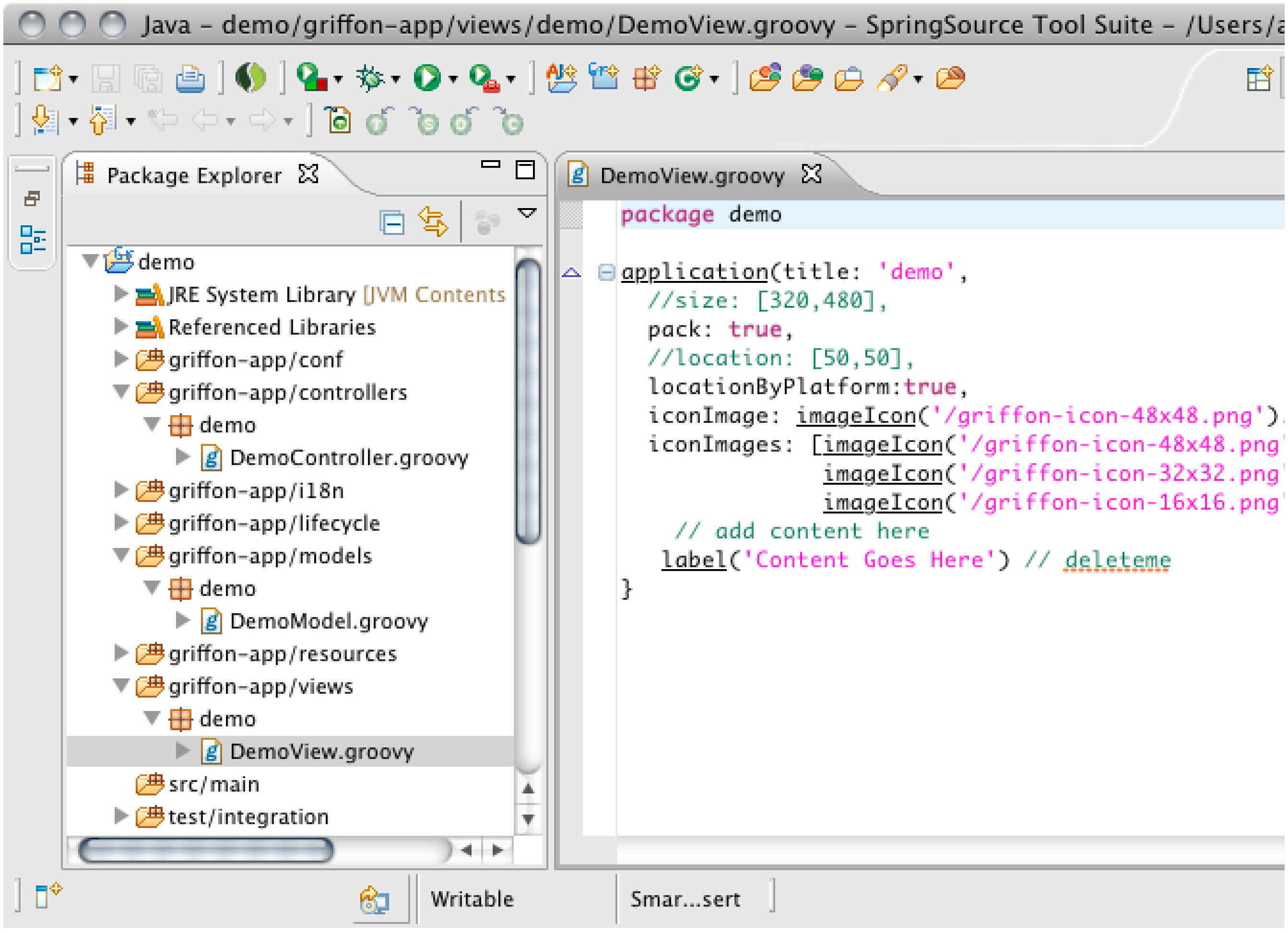
- [ ] Copy projects into workspace

Working sets

- [ ] Add project to working sets

Working sets:

**#4** Click on the Finish button



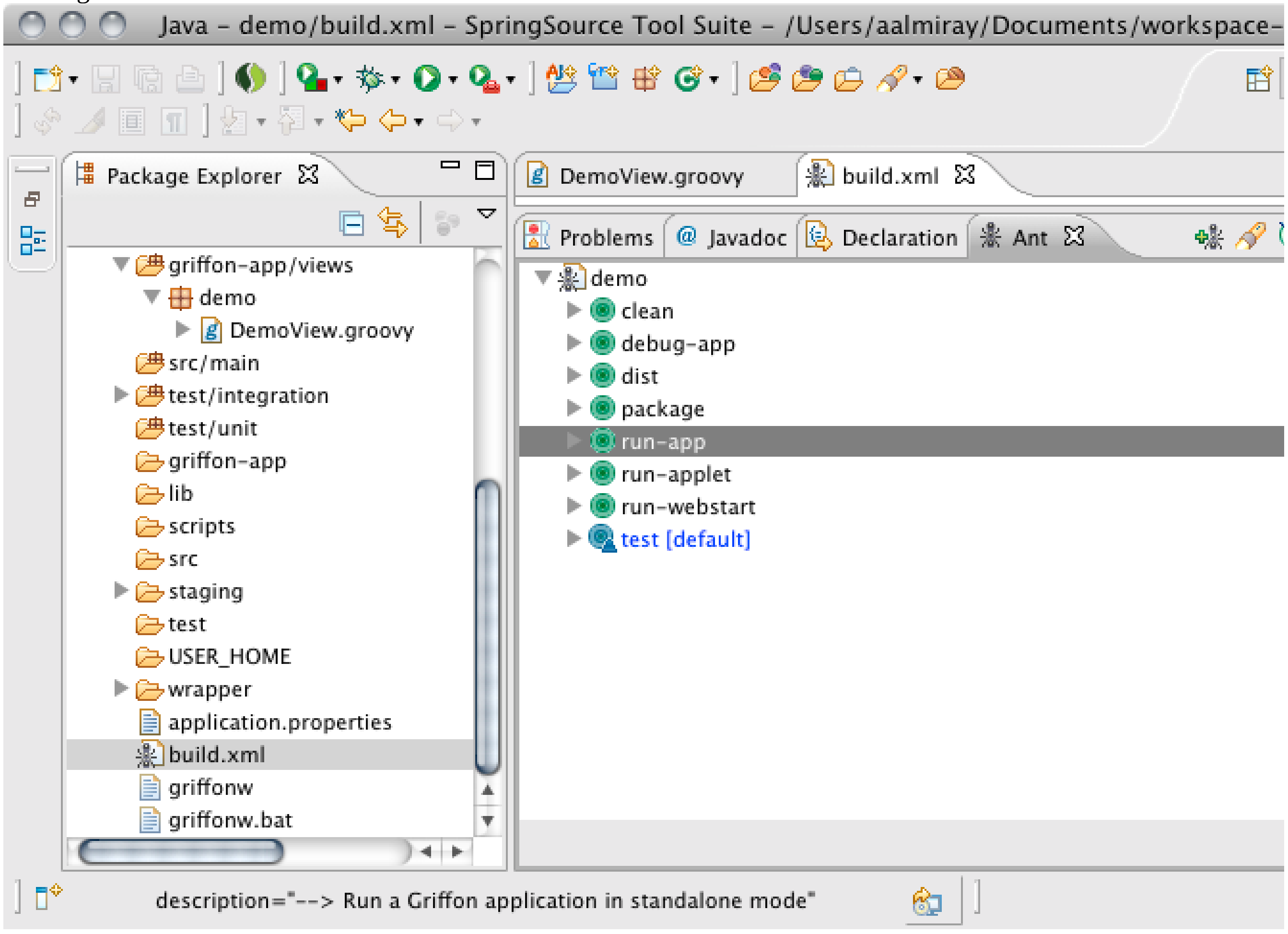
```
package demo

application(title: 'demo',
  //size: [320,480],
  pack: true,
  //location: [50,50],
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.png').
  iconImages: [imageIcon('/griffon-icon-48x48.png
               imageIcon('/griffon-icon-32x32.png
               imageIcon('/griffon-icon-16x16.png
   // add content here
  label('Content Goes Here') // deleteme
}
```

**Running Griffon commands within Eclipse**

We'll rely on Eclipse's Ant support to get the job done, but first we need to generate an Ant build file

```
griffon integrate-with --ant
```

Refresh the contents of your project. Open the build file in the Ant View. Select any target and execute by double clicking on it.



**TextMate**
Since Griffon' focus is on simplicity it is often possible to utilize more simple editors and TextMate on the Mac has an excellent Groovy/Griffon bundle available.
Follow these steps to install the Groovy bundle
**#1** Create a local bundle directory

```
mkdir ~/Library/Application Support/TextMate/Bundles/
```

**#2a** If you have git installed then just clone the repository

```
cd ~/Library/Application Support/TextMate/Bundles/
git clone https://github.com/textmate/groovy.tmbundle.git
```

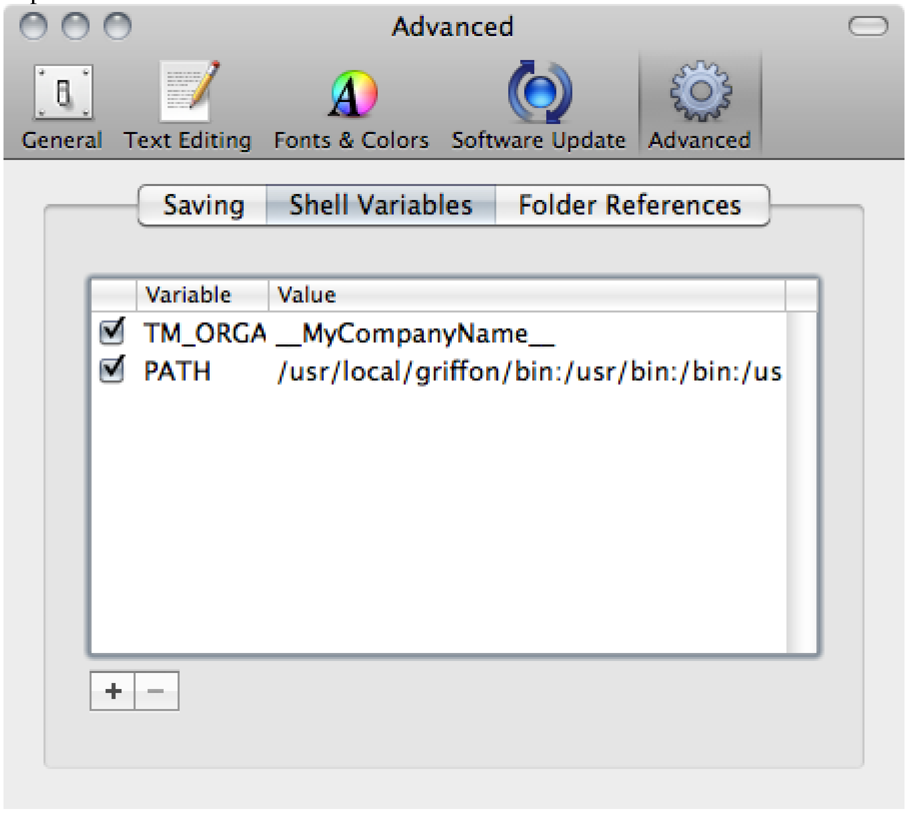
**#2b** Alternatively download a copy of the latest version from github as a zip and unpack it. Rename the unpacked directory to `groovy.tmbundle`.
Follow these steps to install the Griffon bundle
**#1** Create a local bundle directory

```
mkdir ~/Library/Application Support/TextMate/Bundles/
```

**#2a** If you have git installed then just clone the repository

```
cd ~/Library/Application Support/TextMate/Bundles/
git clone https://github.com/griffon/griffon.tmbundle.git
```

**#2b** Alternatively download a copy of the latest version from github as a zip and unpack it. Rename the unpacked directory to `griffon.tmbundle`.
Now configure the `PATH` environment variable within TextMate. Make sure that `$GRIFFON_HOME/bin` in expanded form is set



| | Variable | Value |
|---|---|---|
| ✓ | TM_ORGA | __MyCompanyName__ |
| ✓ | PATH | /usr/local/griffon/bin:/usr/bin:/bin:/us |

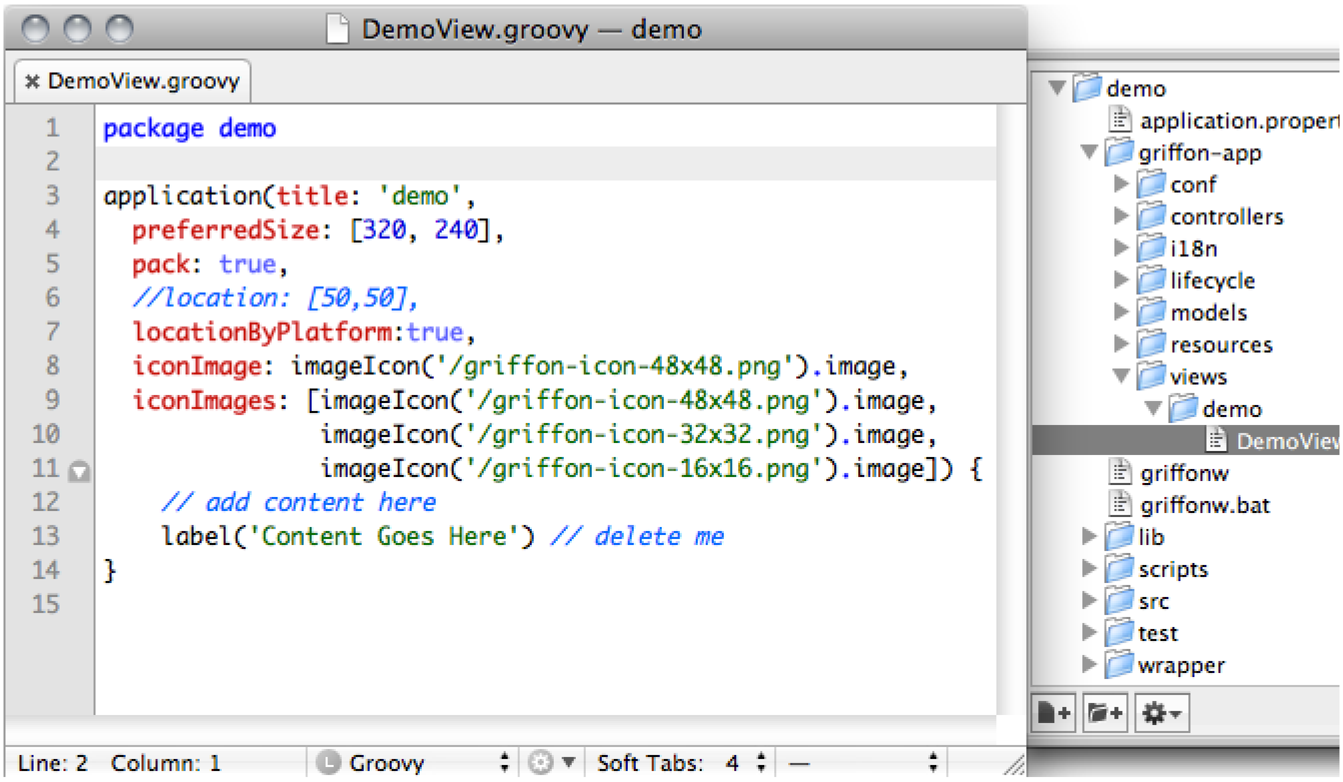
**Integrating an existing Griffon project**
To integrate Griffon with TextMate run the following command to generate appropriate project files:

```
griffon integrate-with --textmate
```

Alternatively TextMate can easily open any project with its command line integration by issuing the following command from the root of your project:

```
mate .
```

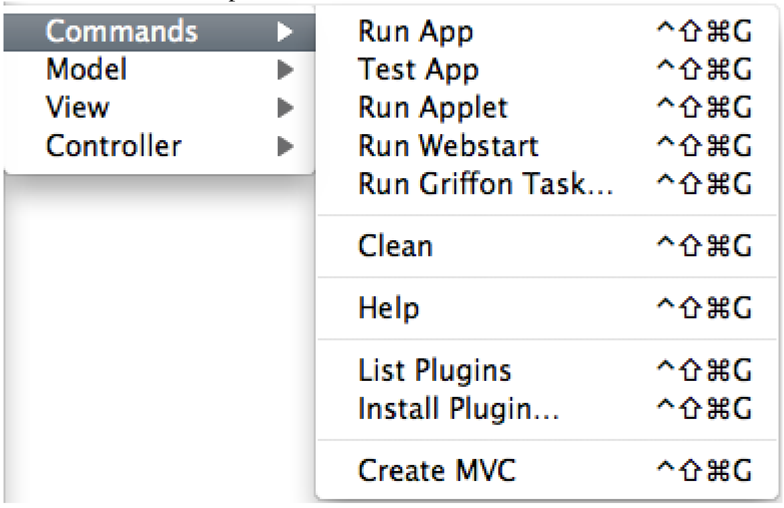
You should see a similar display like the next one


DemoView.groovy — demo

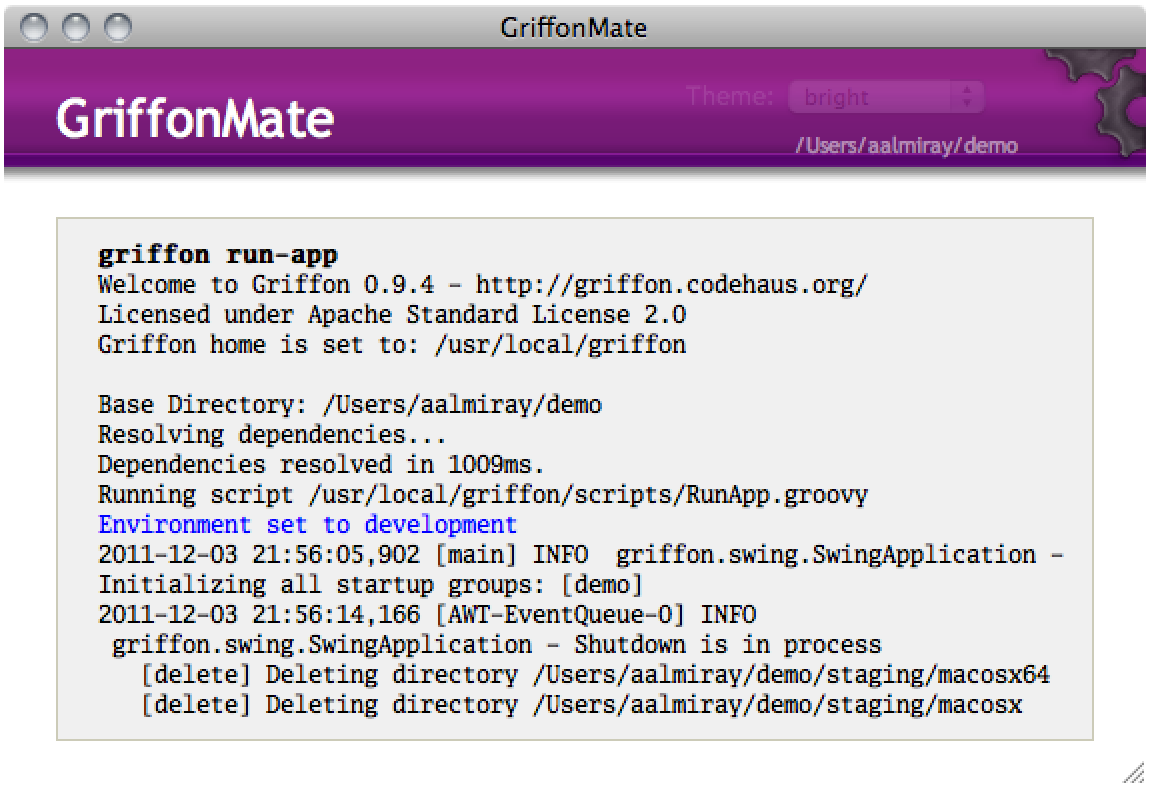
```
package demo

application(title: 'demo',
  preferredSize: [320, 240],
  pack: true,
  //location: [50,50],
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.png').image,
  iconImages: [imageIcon('/griffon-icon-48x48.png').image,
               imageIcon('/griffon-icon-32x32.png').image,
               imageIcon('/griffon-icon-16x16.png').image]) {
    // add content here
    label('Content Goes Here') // delete me
}
```

**Running Griffon commands within TextMate**

The Griffon bundle provides new commands under the "Bundles" menu. Search for the "Griffon submenu".



Selecting "Run App" will execute the run-app command on the currently open project



```
griffon run-app
Welcome to Griffon 0.9.4 - http://griffon.codehaus.org/
Licensed under Apache Standard License 2.0
Griffon home is set to: /usr/local/griffon

Base Directory: /Users/aalmiray/demo
Resolving dependencies...
Dependencies resolved in 1009ms.
Running script /usr/local/griffon/scripts/RunApp.groovy
Environment set to development
2011-12-03 21:56:05,902 [main] INFO  griffon.swing.SwingApplication -
Initializing all startup groups: [demo]
2011-12-03 21:56:14,166 [AWT-EventQueue-0] INFO
 griffon.swing.SwingApplication - Shutdown is in process
   [delete] Deleting directory /Users/aalmiray/demo/staging/macosx64
   [delete] Deleting directory /Users/aalmiray/demo/staging/macosx
```

### 2.5 Convention over Configuration

Griffon uses "convention over configuration" to configure itself. This typically means that the name and location of files is used instead of explicit configuration, hence you need to familiarize yourself with the directory structure provided by Griffon.
Here is a breakdown and links to the relevant sections:

- `griffon-app` - top level directory for Groovy sources.
  - `conf` - Configuration sources.
  - `models` - Models.
  - `views` - Views.
  - `controllers` - Controllers.
  - `services` - Services.
  - `resources` - Images, properties files, etc.
  - `i18n` - Support for internationalization (i18n).
- `scripts` - Gant scripts.
- `src` - Supporting sources.
  - `main` - Other Groovy/Java sources.
- `test` - Unit and integration tests.

### 2.6 Running an Application

Griffon applications can be run in standalone mode using the run-app command:

```
griffon run-app
```

Or in applet mode using the run-applet command:

```
griffon run-applet
```

Or in webstart mode using the run-webstart command:

```
griffon run-webstart
```

More information on the run-app command can be found in the reference guide.

## 2.7 Testing an Application

The `create-*` commands in Griffon automatically create integration tests for you within the `test/integration` directory. It is of course up to you to populate these tests with valid test logic, information on which can be found in the section on Testing. However, if you wish to execute tests you can run the test-app command as follows:

```
griffon test-app
```

Griffon also automatically generates an Ant `build.xml` which can also run the tests by delegating to Griffon' test-app command:

```
ant test
```

This is useful when you need to build Griffon applications as part of a continuous integration platform such as CruiseControl.

## 2.8 Creating Artefacts

Griffon ships with a few convenience targets such as create-mvc, create-script and so on that will create Controllers and different artifact types for you.

> These are merely for your convenience and you can just as easily use an IDE or your favorite text editor.

There are many such `create-*` commands that can be explored in the command line reference guide.

# 3. Configuration

It may seem odd that in a framework that embraces "convention-over-configuration" that we tackle this topic now, but since what configuration there is typically a one off, it is best to get it out the way.

## 3.1 Basic Configuration

For general configuration Griffon provides a file called `griffon-app/conf/Config.groovy`. This file uses Groovy's ConfigSlurper which is very similar to Java properties files except it is pure Groovy hence you can re-use variables and use proper Java types!
You can add your own configuration in here, for example:

```
foo.bar.hello = "world"
```

Then later in your application you can access these settings via the GriffonApplication object, which is available as a variable in mvc members

```
assert "world" == app.config.foo.bar.hello
```

### 3.1.1 Logging

**The Basics**
Griffon uses its common configuration mechanism to provide the settings for the underlying Log4j log system, so all you have to do is add a `log4j` setting to the file `griffon-app/conf/Config.groovy`.
So what does this `log4j` setting look like? Here's a basic example:

```
log4j = {
    error  'org.codehaus.griffon'
    info   'griffon.util',
           'griffon.core',
           'griffon.swing',
           'griffon.app'
}
```

This says that for the 'org.codehaus.griffon' logger, only messages logged at 'error' level and above will be shown. The loggers whose category start with 'griffon' show messages at the 'info' level. What does that mean? First of all, you have to understand how levels work.

**Logging levels**
The are several standard logging levels, which are listed here in order of descending priority:

1. off
2. fatal
3. error
4. warn
5. info
6. debug
7. trace
8. all

When you log a message, you implicitly give that message a level. For example, the method `log.error(msg)` will log a message at the 'error' level. Likewise, `log.debug(msg)` will log it at 'debug'. Each of the above levels apart from 'off' and 'all' have a corresponding log method of the same name.
The logging system uses that *message* level combined with the configuration for the logger (see next section) to determine whether the message gets written out. For example, if you have an 'org.example.domain' logger configured like so:

```
warn 'org.example.domain'
```

then messages with a level of 'warn', 'error', or 'fatal' will be written out. Messages at other levels will be ignored. Before we go on to loggers, a quick note about those 'off' and 'all' levels. These are special in that they can only be used in the configuration; you can't log messages at these levels. So if you configure a logger with a level of 'off', then no messages will be written out. A level of 'all' means that you will see all messages. Simple.

**Loggers**

Loggers are fundamental to the logging system, but they are a source of some confusion. For a start, what are they? Are they shared? How do you configure them?

A logger is the object you log messages to, so in the call `log.debug(msg)`, `log` is a logger instance (of type Logger). These loggers are uniquely identified by name and if two separate classes use loggers with the same name, those loggers are effectively the same instance.

There are two main ways to get hold of a logger:

1. use the `log` instance injected into artifacts such as domain classes, controllers and services;
2. use the Slf4j API directly.

If you use the dynamic `log` property, then the name of the logger is 'griffon.app.<type>.<className>', where `type` is the type of the artifact, say 'controller' or 'service, and `className` is the fully qualified name of the artifact. For example, let's say you have this service:

```
package org.example
class MyService {
    …
}
```

then the name of the logger will be 'griffon.app.service.org.example.MyService'.

For other classes, the typical approach is to store a logger based on the class name in a constant static field:

```
package org.other
import org.slf4j.Logger
import org.slf4j.LoggerFactory
class MyClass {
    private static final Logger log = LoggerFactory.getLogger(MyClass)
    …
}
```

This will create a logger with the name 'org.other.MyClass' - note the lack of a 'griffon.app.' prefix. You can also pass a name to the `getLog()` method, such as "myLogger", but this is less common because the logging system treats names with dots ('.') in a special way.

**Configuring loggers**

You have already seen how to configure a logger in Griffon:

```
log4j = {
    error   'org.codehaus.griffon.runtime'
}
```

This example configures a logger named 'org.codehaus.griffon.runtime' to ignore any messages sent to it at a level of 'warn' or lower. But is there a logger with this name in the application? No. So why have a configuration for it? Because the above rule applies to any logger whose name *begins with* 'org.codehaus.griffon.runtime.' as well. For example, the rule applies to both the `org.codehaus.griffon.runtime.core.DefaultArtifactManager` class and the `org.codehaus.griffon.runtime.util.GriffonApplicationHelper` one.

In other words, loggers are effectively hierarchical. This makes configuring them by package much, much simpler than it would otherwise be.

The most common things that you will want to capture log output from are your controllers, services, and other artifacts. To do that you'll need to use the convention mentioned earlier: *griffon.app.<artifactType>.<className>* . In particular the class name must be fully qualifed, i.e. with the package if there is one:

```
log4j = {
    // Set level for all application artifacts
    info "griffon.app"
    // Set for a specific controller
    debug "griffon.app.controller.YourController"
    // Set for a specific service class
    debug "griffon.app.service.org.example.SampleService"
    // Set for all models
    info "griffon.app.model"
}
```

The standard artifact names used in the logging configuration are:

- `model` - For model classes
- `controller` - For controllers
- `view` - For views
- `service` - For service classes

Griffon itself generates plenty of logging information and it can sometimes be helpful to see that. Here are some useful loggers from Griffon internals that you can use, especially when tracking down problems with your application:

- `org.codehaus.griffon.runtime.core` - Core internal information such as MVC group instantiation, etc.
- `griffon.swing` - Swing related initialization and application life cycle.

So far, we've only looked at explicit configuration of loggers. But what about all those loggers that *don't* have an explicit configuration? Are they simply ignored? The answer lies with the root logger.

**The Root Logger**

All logger objects inherit their configuration from the root logger, so if no explicit configuration is provided for a given logger, then any messages that go to that logger are subject to the rules defined for the root logger. In other words, the root logger provides the default configuration for the logging system.
Griffon automatically configures the root logger to only handle messages at 'error' level and above, and all the messages are directed to the console (stdout for those with a C background). You can customise this behaviour by specifying a 'root' section in your logging configuration like so:

```
log4j = {
    root {
        info()
    }
    …
}
```

The above example configures the root logger to log messages at 'info' level and above to the default console appender. You can also configure the root logger to log to one or more named appenders (which we'll talk more about shortly):

```
log4j = {
    appenders {
        file name:'file', file:'/var/logs/mylog.log'
    }
    root {
        debug 'stdout', 'file'
    }
}
```

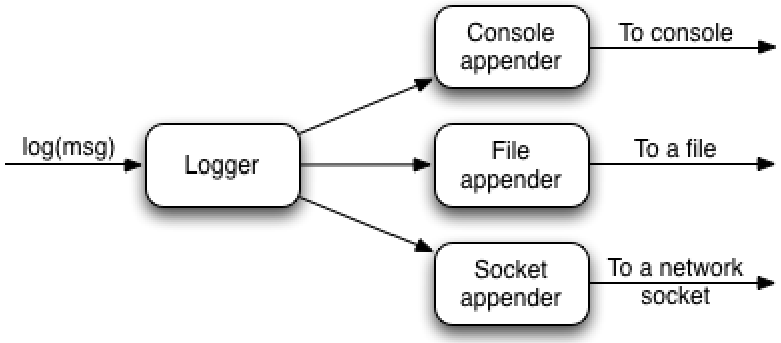
In the above example, the root logger will log to two appenders - the default 'stdout' (console) appender and a custom 'file' appender.
For power users there is an alternative syntax for configuring the root logger: the root `org.apache.log4j.Logger` instance is passed as an argument to the log4j closure. This allows you to work

with the logger directly:

```
log4j = { root ->
    root.level = org.apache.log4j.Level.DEBUG
    …
}
```

For more information on what you can do with this `Logger` instance, refer to the Log4j API documentation.
Those are the basics of logging pretty well covered and they are sufficient if you're happy to only send log messages to the console. But what if you want to send them to a file? How do you make sure that messages from a particular logger go to a file but not the console? These questions and more will be answered as we look into appenders.

**Appenders**
Loggers are a useful mechanism for filtering messages, but they don't physically write the messages anywhere. That's the job of the appender, of which there are various types. For example, there is the default one that writes messages to the console, another that writes them to a file, and several others. You can even create your own appender implementations!
This diagram shows how they fit into the logging pipeline:



As you can see, a single logger may have several appenders attached to it. In a standard Griffon configuration, the console appender named 'stdout' is attached to all loggers through the default root logger configuration. But that's the only one. Adding more appenders can be done within an 'appenders' block:

```
log4j = {
    appenders {
        rollingFile name: "myAppender", maxFileSize: 1024, file: "/tmp/logs/myApp.log"
    }
}
```

The following appenders are available by default:

- console (ConsoleAppender) - Logs to the console.
- file (FileAppender) - Logs to a single file.
- rollingFile (RollingFileAppender) - Logs to rolling files, for example a new file each day.
- event (GriffonApplicationEventAppender) - Logs to application events. Event name is "LogEvent"; args are log level (as String), log message and optional throwable.

Each named argument passed to an appender maps to a property of the underlying Appender implementation. So the previous example sets the `name`, `maxFileSize` and `file` properties of the `RollingFileAppender` instance.
You can have as many appenders as you like - just make sure that they all have unique names. You can even have multiple instances of the same appender type, for example several file appenders that log to different files.
If you prefer to create the appender programmatically or if you want to use an appender implementation that's not available via the above syntax, then you can simply declare an `appender` entry with an instance of the appender you want:

```
import org.apache.log4j.*
log4j = {
    appenders {
        appender new RollingFileAppender(name: "myAppender", maxFileSize: 1024, file: "/tmp/
    }
}
```

This approach can be used to configure `JMSAppender`, `SocketAppender`, `SMTPAppender`, and more.
Once you have declared your extra appenders, you can attach them to specific loggers by passing the name as a key to one of the log level methods from the previous section:

```
error myAppender: "griffon.app.controller.BookController"
```

This will ensure that the 'org.codehaus.groovy.griffon.commons' logger and its children send log messages to 'myAppender' as well as any appenders configured for the root logger. If you want to add more than one appender to the logger, then add them to the same level declaration:

```
error myAppender:      "griffon.app.controller.BookController",
      myFileAppender:  ["griffon.app.controller.BookController", "griffon.app.service.BookSe
      rollingFile:     "griffon.app.controller.BookController"
```

The above example also shows how you can configure more than one logger at a time for a given appender ( `myFileAppender`) by using a list.
Be aware that you can only configure a single level for a logger, so if you tried this code:

```
error myAppender:      "griffon.app.controller.BookController"
debug myFileAppender:  "griffon.app.controller.BookController"
fatal rollingFile:     "griffon.app.controller.BookController"
```

you'd find that only 'fatal' level messages get logged for 'griffon.app.controller.BookController'. That's because the last level declared for a given logger wins. What you probably want to do is limit what level of messages an appender writes.
Let's say an appender is attached to a logger configured with the 'all' level. That will give us a lot of logging information that may be fine in a file, but makes working at the console difficult. So, we configure the console appender to only write out messages at 'info' level or above:

```
log4j = {
    appenders {
        console name: "stdout", threshold: org.apache.log4j.Level.INFO
    }
}
```

The key here is the `threshold` argument which determines the cut-off for log messages. This argument is available for all appenders, but do note that you currently have to specify a `Level` instance - a string such as "info" will not work.

**Custom Layouts**
By default the Log4j DSL assumes that you want to use a PatternLayout. However, there are other layouts available including:

- `xml` - Create an XML log file
- `html` - Creates an HTML log file
- `simple` - A simple textual log
- `pattern` - A Pattern layout

You can specify custom patterns to an appender using the `layout` setting:

```
log4j = {
    appenders {
        console name: "customAppender", layout: pattern(conversionPattern: "%c{2} %m%n")
    }
}
```

This also works for the built-in appender "stdout", which logs to the console:

```
log4j = {
    appenders {
        console name: "stdout", layout: pattern(conversionPattern: "%c{2} %m%n")
    }
}
```

**Environment-specific configuration**
Since the logging configuration is inside `Config.groovy`, you can of course put it inside an environment-specific block. However, there is a problem with this approach: you have to provide the full logging configuration each time you define the `log4j` setting. In other words, you cannot selectively override parts of the configuration - it's all or nothing.
To get round this, the logging DSL provides its own environment blocks that you can put anywhere in the configuration:

```
log4j = {
    appenders {
        console name: "stdout", layout: pattern(conversionPattern: "%c{2} %m%n")
        environments {
            production {
                rollingFile name: "myAppender", maxFileSize: 1024, file: "/tmp/logs/myApp.lo
            }
        }
    }
    root {
        //…
    }
    // other shared config
    info "griffon.app.controller"
    environments {
        production {
            // Override previous setting for 'griffon.app.controller'
            error "griffon.app.controller"
        }
    }
}
```

The one place you can't put an environment block is *inside* the `root` definition, but you can put the `root` definition inside an environment block.

**Full stacktraces**
When exceptions occur, there can be an awful lot of noise in the stacktrace from Java and Groovy internals. Griffon filters these typically irrelevant details and restricts traces to non-core Griffon/Groovy class packages.
When this happens, the full trace is always logged to the `StackTrace` logger, which by default writes its output to a file called `stacktrace.log`. As with other loggers though, you can change its behaviour in the configuration. For example if you prefer full stack traces to go to the console, add this entry:

```
error stdout: "StackTrace"
```

This won't stop Griffon from attempting to create the stacktrace.log file - it just redirects where stack traces are written to. An alternative approach is to change the location of the 'stacktrace' appender's file:

```
log4j = {
    appenders {
        rollingFile name: "stacktrace", maxFileSize: 1024, file: "/var/tmp/logs/myApp-stackt
    }
}
```

or, if you don't want to the 'stacktrace' appender at all, configure it as a 'null' appender:

```
log4j = {
    appenders {
        'null' name: "stacktrace"
    }
}
```

You can of course combine this with attaching the 'stdout' appender to the 'StackTrace' logger if you want all the output in the console.
Finally, you can completely disable stacktrace filtering by setting the `griffon.full.stacktrace` VM property to `true`:

```
griffon -Dgriffon.full.stacktrace=true run-app
```

**Logger inheritance**
Earlier, we mentioned that all loggers inherit from the root logger and that loggers are hierarchical based on '.'-separated terms. What this means is that unless you override a parent setting, a logger retains the level and the appenders configured for that parent. So with this configuration:

```
log4j = {
    appenders {
        file name:'file', file:'/var/logs/mylog.log'
    }
    root {
        debug 'stdout', 'file'
    }
}
```

all loggers in the application will have a level of 'debug' and will log to both the 'stdout' and 'file' appenders. What if you only want to log to 'stdout' for a particular logger? In that case, you need to change the 'additivity' for a logger. Additivity simply determines whether a logger inherits the configuration from its parent. If additivity is false, then its not inherited. The default for all loggers is true, i.e. they inherit the configuration. So how do you change this setting? Here's an example:

```
log4j = {
    appenders {
        …
    }
    root {
        …
    }
    info additivity: false
         stdout: ["griffon.app.controller.BookController", "griffon.app.service.BookService"
}
```

So when you specify a log level, add an 'additivity' named argument. Note that you when you specify the additivity, you must configure the loggers for a named appender. The following syntax will *not* work:

```
info additivity: false, "griffon.app.controller.BookController", "griffon.app.service.BookSe
```

## 3.2 Environments

**Per Environment Configuration**

Griffon supports the concept of per environment configuration. The `BuildConfig.groovy` file within the `griffon-app/conf` directory can take advantage of per environment configuration using the syntax provided by ConfigSlurper . As an example consider the following default packaging definitions provided by Griffon:

```
environments {
    development {
        signingkey {
            params {
                sigfile = 'GRIFFON'
                keystore = "${basedir}/griffon-app/conf/keys/devKeystore"
                alias = 'development'
                storepass = 'BadStorePassword'
                keypass   = 'BadKeyPassword'
                lazy      = true // only sign when unsigned
            }
        }
    }
    test {
        griffon {
            jars {
                sign = false
                pack = false
            }
        }
    }
    production {
        signingkey {
            params {
                sigfile = 'GRIFFON'
                keystore = 'CHANGE ME'
                alias = 'CHANGE ME'
                lazy = false // sign, regardless of existing signatures
            }
        }
        griffon {
            jars {
                sign = true
                pack = true
                destDir = "${basedir}/staging"
            }
            webstart {
                codebase = 'CHANGE ME'
            }
        }
    }
}
griffon {
    jars {
        sign = false
        pack = false
        destDir = "${basedir}/staging"
        jarName = "${appName}.jar"
    }
}
```

Notice how the common configuration is provided at the bottom level (it actually can be placed before the `environments` block too), the `environments` block specifies per environment settings for the `jars` property.

**Packaging and Running for Different Environments**

Griffon' command line has built in capabilities to execute any command within the context of a specific environment. The format is:

```
griffon [environment] [command name]
```

In addition, there are 3 preset environments known to Griffon: `dev`, `prod`, and `test` for `development`, `production` and `test`. For example to package an application for the `development` (avoiding jar signing by default) environment you could do:

```
griffon dev package
```

If you have other environments that you need to target you can pass a `griffon.env` variable to any command:

```
griffon -Dgriffon.env=UAT run-app
```

**Programmatic Environment Detection**
Within your code, such as in a Gant script or a bootstrap class you can detect the environment using the Environment class:

```
import griffon.util.Environment
...
switch(Environment.current) {
    case Environment.DEVELOPMENT:
        configureForDevelopment()
        break
    case Environment.PRODUCTION:
        configureForProduction()
    break
}
```

**Generic Per Environment Execution**
You can use the `griffon.util.Environment` class to execute your own environment specific logic:

```
Environment.executeForCurrentEnvironment {
    production {
        // do something in production
    }
    development {
        // do something only in development
    }
}
```

## 3.3 Versioning

**Versioning Basics**
Griffon has built in support for application versioning. When you first create an application with the create-app command the version of the application is set to `0.1`. The version is stored in the application meta data file called `application.properties` in the root of the project.
To change the version of your application you can run the set-version command:

```
griffon set-version 0.2
```

The version is used in various commands including the package command which will append the application version to the end of the created distribution zip files.

**Detecting Versions at Runtime**
You can detect the application version using Griffon' support for application metadata using the app class. For example within controllers there is an implicit app variable that can be used:

```
def version = app.metadata['app.version']
```

If it is the version of Griffon you need you can use:

```
def griffonVersion = app.metadata['app.griffon.version']
```

## 3.4 Dependency Resolution

In order to control how JAR dependencies are resolved Griffon features (since version 0.9) a dependency resolution DSL that allows you to control how dependencies for applications and plugins are resolved.
Inside the `griffon-app/conf/BuildConfig.groovy` file you can specify a `griffon.project.dependency.resolution` property that configures how dependencies are resolved:

```
griffon.project.dependency.resolution = {
   // config here
}
```

The default configuration looks like the following:

```
griffon.project.dependency.resolution = {
    // inherit Griffon' default dependencies
    inherits("global") {
    }
    log "warn" // log level of Ivy resolver, either 'error', 'warn', 'info', 'debug' or 'ver
    repositories {
        griffonHome()
        // uncomment the below to enable remote dependency resolution
        // from public Maven repositories
        //mavenLocal()
        //mavenCentral()
        //mavenRepo "http://snapshots.repository.codehaus.org"
        //mavenRepo "http://repository.codehaus.org"
        //mavenRepo "http://download.java.net/maven/2/"
        //mavenRepo "http://repository.jboss.com/maven2/"
    }
    dependencies {
        // specify dependencies here under either 'build', 'compile', 'runtime' or 'test' sc
        // runtime 'mysql:mysql-connector-java:5.1.5'
    }
}
```

The details of the above will be explained in the next few sections.

### 3.4.1 Configurations and Dependencies

Griffon features 5 dependency resolution configurations (or 'scopes') which you can take advantage of:

- `build`: Dependencies for the build system only
- `compile`: Dependencies for the compile step
- `runtime`: Dependencies needed at runtime but not for compilation (see above)
- `test`: Dependencies needed for testing but not at runtime (see above)

Within the `dependencies` block you can specify a dependency that falls into one of these configurations by calling the equivalent method. For example if your application requires the MySQL driver to function at `runtime` you can specify as such:

```
runtime 'com.mysql:mysql-connector-java:5.1.5'
```

The above uses the string syntax which is `group:name:version`. You can also use a map-based syntax:

```
runtime group:'com.mysql', name:'mysql-connector-java', version:'5.1.5'
```

Multiple dependencies can be specified by passing multiple arguments:

```
runtime 'com.mysql:mysql-connector-java:5.1.5',
        'commons-lang:commons-lang:2.6'
// Or
runtime(
    [group: 'com.mysql',     name: 'mysql-connector-java', version: '5.1.5'],
    [group: 'commnons-lang', name: 'commons-lang',         version: '2.6']
)
```

You may specify a classifier too

```
runtime 'net.sf.json-lib:json-lib:2.4:jdk15'
// Or
runtime group: 'net.sf.json-lib' name: 'json-lib', version: '2.4', classifier: 'jdk15'
```

## 3.4.2 Dependency Repositories

**Remote Repositories**
Griffon, when installed, does not use any remote public repositories. There is a default `griffonHome()` repository that will locate the JAR files Griffon needs from your Griffon installation. If you want to take advantage of a public repository you need to specify as such inside the `repositories` block:

```
repositories {
    mavenCentral()
}
```

In this case the default public Maven repository is specified. To use the SpringSource Enterprise Bundle Repository you can use the `ebr()` method:

```
repositories {
    ebr()
}
```

You can also specify a specific Maven repository to use by URL:

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

**Local Resolvers**
If you do not wish to use a public Maven repository you can specify a flat file repository:

```
repositories {
    flatDir name:'myRepo', dirs:'/path/to/repo'
}
```

**Custom Resolvers**
If all else fails since Griffon builds on Apache Ivy you can specify an Ivy resolver:

```
repositories {
    resolver new URLResolver(...)
}
```

**Authentication**
If your repository requires some form of authentication you can specify as such using a `credentials` block:

```
credentials {
    realm = ".."
    host = "localhost"
    username = "myuser"
    password = "mypass"
}
```

The above can also be placed in your `USER_HOME/.griffon/settings.groovy` file using the `griffon.project.ivy.authentication` setting:

```
griffon.project.ivy.authentication = {
    credentials {
        realm = ".."
        host = "localhost"
        username = "myuser"
        password = "mypass"
    }
}
```

### 3.4.3 Debugging Resolution

If you are having trouble getting a dependency to resolve you can enable more verbose debugging from the underlying engine using the `log` method:

```
// log level of Ivy resolver, either 'error', 'warn', 'info', 'debug' or 'verbose'
log "warn"
```

### 3.4.4 Inherited Dependencies

By default every Griffon application inherits a bunch of framework dependencies. This is done through the line:

```
inherits "global"
```

Inside the `BuildConfig.groovy` file. If you wish exclude certain inherited dependencies then you can do so using the `excludes` method:

```
inherits("global") {
    excludes "oscache", "ehcache"
}
```

### 3.4.5 Dependency Reports

As mentioned in the previous section a Griffon application consists of dependencies inherited from the framework,

the plugins installed and the application dependencies itself.
To obtain a report of an application's dependencies you can run the dependency-report command:

```
griffon dependency-report
```

This will output a report to the `target/dependency-report` directory by default. You can specify which configuration (scope) you want a report for by passing an argument containing the configuration name:

```
griffon dependency-report runtime
```

### 3.4.6 Plugin JAR Dependencies

**Specifying Plugin JAR dependencies**
The way in which you specify dependencies for a plugin is identical to how you specify dependencies in an application. When a plugin is installed into an application the application automatically inherits the dependencies of the plugin.
If you want to define a dependency that is resolved for use with the plugin but not *exported* to the application then you can set the `exported` property of the dependency:

```
compile('org.hibernate:hibernate-core:3.3.1.GA') {
    exported = false
}
```

In this can the `hibernate-core` dependency will be available only to the plugin and not resolved as an application dependency.

**Overriding Plugin JAR Dependencies in Your Application**
If a plugin is using a JAR which conflicts with another plugin, or an application dependency then you can override how a plugin resolves its dependencies inside an application using exclusions. For example:

```
plugins {
    compile("org.codehaus.griffon.plugins:miglayout:0.3" ) {
        excludes "miglayout"
    }
}
dependencies {
    String miglayoutVersion = '4.2'
    compile "com.miglayout:miglayout-core:$miglayoutVersion",
            "com.miglayout:miglayout-swing:$miglayoutVersion"
}
```

In this case the application explicitly declares a dependency on the "miglayout" plugin and specifies an exclusion using the `excludes` method, effectively excluding the miglayout library as a dependency.

### 3.4.7 Plugin Dependencies

As of Griffon 0.9 you can declaratively specify dependencies on plugins rather than using the install-plugin command:

```
plugins {
    runtime ':artifacts:0.2'
}
```

If you don't specify a group id the default plugin group id of `org.griffon.plugins` is used. You can specify to use the latest version of a particular plugin by using "latest.integration" as the version number:

```
plugins {
    runtime ':artifacts:latest.integration'
}
```

**Integration vs. Release**
The "latest.integration" version label will also include resolving snapshot versions. If you don't want to include snapshot versions then you can use the "latest.release" label:

```
plugins {
    runtime ':artifacts:latest.release'
}
```

> The "latest.release" label only works with Maven compatible repositories. If you have a regular SVN-based Griffon repository then you should use "latest.integration".

And of course if you are using a Maven repository with an alternative group id you can specify a group id:

```
plugins {
    runtime 'mycompany:artifacts:latest.integration'
}
```

**Plugin Exclusions**
You can control how plugins transitively resolves both plugin and JAR dependencies using exclusions. For example:

```
plugins {
    runtime( ':weceem:0.8' ) {
        excludes "searchable"
    }
}
```

Here we have defined a dependency on the "weceem" plugin which transitively depends on the "searchable" plugin. By using the `excludes` method you can tell Griffon *not* to transitively install the searchable plugin. You can combine this technique to specify an alternative version of a plugin:

```
plugins {
    runtime( ':weceem:0.8' ) {
        excludes "searchable" // excludes most recent version
    }
    runtime ':searchable:0.5.4' // specifies a fixed searchable version
}
```

You can also completely disable transitive plugin installs, in which case no transitive dependencies will be resolved:

```
plugins {
    runtime( ':weceem:0.8' ) {
        transitive = false
    }
    runtime ':searchable:0.5.4' // specifies a fixed searchable version
}
```

## 3.5 Project Documentation

Since Griffon 0.9, the documentation engine that powers the creation of this documentation is available to your Griffon projects.
The documentation engine uses a variation on the Textile syntax to automatically create project documentation with smart linking, formatting etc.

**Creating project documentation**
To use the engine you need to follow a few conventions. Firstly you need to create a `src/docs/guide` directory and then have numbered text files using the `gdoc` format. For example:

```
+ src/docs/guide/1. Introduction.gdoc
+ src/docs/guide/2. Getting Started.gdoc
```

The title of each chapter is taken from the file name. The order is dictated by the numerical value at the beginning of the file name.

**Creating reference items**
Reference items appear in the left menu on the documentation and are useful for quick reference documentation. Each reference item belongs to a category and a category is a directory located in the `src/docs/ref` directory. For example say you defined a new method called `renderPDF`, that belongs to a category called `Controllers` this can be done by creating a gdoc text file at the following location:

```
+ src/ref/Controllers/renderPDF.gdoc
```

**Configuring Output Properties**
There are various properties you can set within your `griffon-app/conf/BuildConfig.groovy` file that customize the output of the documentation such as:

- **griffon.doc.authors -** The authors of the documentation
- **griffon.doc.license -** The license of the software
- **griffon.doc.copyright -** The copyright message to display
- **griffon.doc.footer -** The footer to use

Other properties such as the name of the documentation and the version are pulled from your project itself.

**Generating Documentation**
Once you have created some documentation (refer to the syntax guide in the next chapter) you can generate an HTML version of the documentation using the command:

```
griffon doc
```

This command will output an `docs/manual/index.html` which can be opened to view your documentation.

**Documentation Syntax**
As mentioned the syntax is largely similar to Textile or Confluence style wiki markup. The following sections walk you through the syntax basics.

**Basic Formatting**

Monospace: `monospace`

```
@monospace@
```

Italic: *italic*

```
_italic_
```

Bold: **bold**

```
*bold*
```

Image:

```
!http://dist.codehaus.org/griffon/media/griffon.png!
```

**Linking**

There are several ways to create links with the documentation generator. A basic external link can either be defined using confluence or textile style markup:

```
[Griffon|http://griffon.codehaus.org/] or "Griffon":http://griffon.codehaus.org/
```

For links to other sections inside the user guide you can use the `guide:` prefix:

```
[Intro|guide:1. Introduction]
```

The documentation engine will warn you if any links to sections in your guide break. Sometimes though it is preferable not to hard code the actual names of guide sections since you may move them around. To get around this you can create an alias inside `griffon-app/conf/BuildConfig.groovy`:

```
griffon.doc.alias.intro="1. Introduction"
```

And then the link becomes:

```
[Intro|guide:intro]
```

This is useful since if you linked the to "1. Introduction" chapter many times you would have to change all of those links.
To link to reference items you can use a special syntax:

```
[controllers|renderPDF]
```

In this case the category of the reference item is on the left hand side of the | and the name of the reference item on the right.
Finally, to link to external APIs you can use the `api:` prefix. For example:

```
[String|api:java.lang.String]
```

The documentation engine will automatically create the appropriate javadoc link in this case. If you want to add additional APIs to the engine you can configure them in `griffon-app/conf/BuildConfig.groovy`. For example:

```
griffon.doc.api.org.hibernate="http://docs.jboss.org/hibernate/stable/core/api"
```

The above example configures classes within the `org.hibernate` package to link to the Hibernate website's API docs.

#### Lists and Headings

Headings can be created by specifying the letter 'h' followed by a number and then a dot:

```
h3.<space>Heading3
h4.<space>Heading4
```

Unordered lists are defined with the use of the * character:

```
* item 1
** subitem 1
** subitem 2
* item 2
```

Numbered lists can be defined with the # character:

```
# item 1
```

Tables can be created using the `table` macro:

| Name | Number |
|---|---|
| Albert | 46 |
| Wilma | 1348 |
| James | 12 |

```
{table}
 *Name* | *Number*
 Albert | 46
 Wilma | 1348
 James | 12
{table}
```

#### Code and Notes

You can define code blocks with the `code` macro:

```
class Book {
    String title
}
```

```
{code}
class Book {
    String title
}
{code}
```

The example above provides syntax highlighting for Java and Groovy code, but you can also highlight XML markup:

```
<hello>world</hello>
```

```
{code:xml}
<hello>world</hello>
{code}
```

There are also a couple of macros for displaying notes and warnings:
Note:

> This is a note!

```
{note}
This is a note!
{note}
```

Warning:

> This is a warning!

```
{warning}
This is a warning!
{warning}
```

# 4. The Command Line

Griffon' command line system is built on Gant - a simple Groovy wrapper around Apache Ant.
However, Griffon takes it a bit further through the use of convention and the `griffon` command. When you type:

```
griffon [command name]
```

Griffon does a search in the following directories for Gant scripts to execute:

- `USER_HOME/.griffon/scripts`
- `PROJECT_HOME/scripts`
- `PROJECT_HOME/plugins/*/scripts`
- `GRIFFON_HOME/scripts`

Griffon will also convert command names that are in lower case form such as run-app into camel case. So typing

```
griffon run-app
```

Results in a search for the following files:

- `USER_HOME/.griffon/scripts/RunApp.groovy`
- `PROJECT_HOME/scripts/RunApp.groovy`
- `PLUGINS_HOME/*/scripts/RunApp.groovy`
- `GRIFFON_HOME/scripts/RunApp.groovy`

If multiple matches are found Griffon will give you a choice of which one to execute. When Griffon executes a Gant script, it invokes the "default" target defined in that script. If there is no default, Griffon will quit with an error.
To get a list and some help about the available commands type:

```
griffon help
```

Which outputs usage instructions and the list of commands Griffon is aware of:

```
Usage (optionals marked with *):
griffon [environment]* [target] [arguments]*
Examples:
griffon dev run-app
griffon create-app books
Available Targets (type griffon help 'target-name' for more info):
griffon clean
griffon compile
griffon package
...
```

The command interpreter is able to expand abbreviations following a camel case convention.
Examples:

```
griffon tA // expands to test-app
griffon cAd // expands to create-addon
griffon cIT // expands to create-integration-test
```

Refer to the Command Line reference in left menu of the reference guide for more information about individual commands

## 4.1 Creating Gant Scripts

You can create your own Gant scripts by running the create-script command from the root of your project. For example the following command:

```
griffon create-script compile-sources
```

Will create a script called `scripts/CompileSources.groovy`. A Gant script itself is similar to a regular Groovy script except that it supports the concept of "targets" and dependencies between them:

```
target(default:"The default target is the one that gets executed by Griffon") {
    depends(clean, compile)
}
target(clean:"Clean out things") {
    ant.delete(dir:"output")
}
target(compile:"Compile some sources") {
    ant.mkdir(dir:"mkdir")
    ant.javac(srcdir:"src/main", destdir:"output")
}
```

As demonstrated in the script above, there is an implicit `ant` variable that allows access to the Apache Ant API. You can also "depend" on other targets using the `depends` method demonstrated in the `default` target above.

**The default target**
In the example above, we specified a target with the explicit name "default". This is one way of defining the default target for a script. An alternative approach is to use the `setDefaultTarget()` method:

```
target("clean-compile": "Performs a clean compilation on the app's source files.") {
    depends(clean, compile)
}
target(clean:"Clean out things") {
    ant.delete(dir:"output")
}
target(compile:"Compile some sources") {
    ant.mkdir(dir:"mkdir")
    ant.javac(srcdir:"src/java", destdir:"output")
}
setDefaultTarget("clean-compile")
```

This allows you to call the default target directly from other scripts if you wish. Also, although we have put the call to `setDefaultTarget()` at the end of the script in this example, it can go anywhere as long as it comes *after* the target it refers to ("clean-compile" in this case).
Which approach is better? To be honest, you can use whichever you prefer - there don't seem to be any major advantages in either case. One thing we would say is that if you want to allow other scripts to call your "default" target, you should move it into a shared script that doesn't have a default target at all. We'll talk some more about this in the next section.

## 4.2 Re-using Griffon scripts

Griffon ships with a lot of command line functionality out of the box that you may find useful in your own scripts (See the command line reference in the reference guide for info on all the commands). Of particular use are the compile and package scripts.

**Pulling in targets from other scripts**
Gant allows you to pull in all targets (except "default") from another Gant script. You can then depend upon or invoke those targets as if they had been defined in the current script. The mechanism for doing this is the `includeTargets` property. Simply "append" a file or class to it using the left-shift operator:

```
includeTargets << new File("/path/to/my/script.groovy")
includeTargets << gant.tools.Ivy
```

Don't worry too much about the syntax using a class, it's quite specialized. If you're interested, look into the Gant documentation.

**Core Griffon targets**
As you saw in the example at the beginning of this section, you use neither the File- nor the class-based syntax for `includeTargets` when including core Griffon targets. Instead, you should use the special `griffonScript()` method that is provided by the Griffon command launcher (note that this is not available in normal Gant scripts, just Griffon ones).
The syntax for the `griffonScript()` method is pretty straightforward: simply pass it the name of the Griffon script you want to include, without any path information. Here is a list of Griffon scripts that you may want to re-use:

| Script | Description |
|---|---|
| _GriffonSettings | You really should include this! Fortunately, it is included automatically by all other Griffon scripts bar one (_GriffonProxy), so you usually don't have to include it explicitly. |
| _GriffonEvents | If you want to fire events, you need to include this. Adds an `event(String eventName, List args)` method. Again, included by almost all other Griffon scripts. |
| _GriffonClasspath | Sets up compilation, test, and runtime classpaths. If you want to use or play with them, include this script. Again, included by almost all other Griffon scripts. |
| _GriffonProxy | If you want to access the internet, include this script so that you don't run into problems with proxies. |
| _GriffonArgParsing | Provides a `parseArguments` target that does what it says on the tin: parses the arguments provided by the user when they run your script. Adds them to the `argsMap` property. |
| _GriffonTest | Contains all the shared test code. Useful if you want to add any extra tests. |
| RunApp | Provides all you need to run the application in standalone mode. |
| RunApplet | Provides all you need to run the application in applet mode. |
| RunWebstart | Provides all you need to run the application in webstart mode. |

There are many more scripts provided by Griffon, so it is worth digging into the scripts themselves to find out what kind of targets are available. Anything that starts with an "_" is designed for re-use.

**Script architecture**
You maybe wondering what those underscores are doing in the names of the Griffon scripts. That is Griffon' way of determining that a script is *internal* , or in other words that it has not corresponding "command". So you can't run "griffon _griffon-settings" for example. That is also why they don't have a default target.
Internal scripts are all about code sharing and re-use. In fact, we recommend you take a similar approach in your own scripts: put all your targets into an internal script that can be easily shared, and provide simple command scripts that parse any command line arguments and delegate to the targets in the internal script. Say you have a script that runs some functional tests - you can split it like this:

```
./scripts/FunctionalTests.groovy:
includeTargets << new File("${basedir}/scripts/_FunctionalTests.groovy")
target(default: "Runs the functional tests for this project.") {
    depends(runFunctionalTests)
}
./scripts/_FunctionalTests.groovy:
includeTargets << griffonScript("_GriffonTest")
target(runFunctionalTests: "Run functional tests.") {
    depends(...)
    …
}
```

Here are a few general guidelines on writing scripts:

- Split scripts into a "command" script and an internal one.
- Put the bulk of the implementation in the internal script.
- Put argument parsing into the "command" script.
- To pass arguments to a target, create some script variables and initialize them before calling the target.
- Avoid name clashes by using closures assigned to script variables instead of targets. You can then pass

arguments direct to the closures.

## 4.3 Hooking into Events

Griffon provides the ability to hook into scripting events. These are events triggered during execution of Griffon target and plugin scripts.
The mechanism is deliberately simple and loosely specified. The list of possible events is not fixed in any way, so it is possible to hook into events triggered by plugin scripts, for which there is no equivalent event in the core target scripts.

**Defining event handlers**
Event handlers are defined in scripts called `_Events.groovy`. Griffon searches for these scripts in the following locations:

- `USER_HOME/.griffon/scripts` - user-specific event handlers
- `PROJECT_HOME/scripts` - application-specific event handlers
- `PLUGINS_HOME/*/scripts` - plugin-specific event handlers

Whenever an event is fired, *all* the registered handlers for that event are executed. Note that the registration of handlers is performed automatically by Griffon, so you just need to declare them in the relevant `_Events.groovy` file.
Event handlers are blocks defined in `_Events.groovy`, with a name beginning with "event". The following example can be put in your /scripts directory to demonstrate the feature:

```
eventCreatedArtefact = { type, name ->
   println "Created $type $name"
}
eventStatusUpdate = { msg ->
   println msg
}
eventStatusFinal = { msg ->
   println msg
}
```

You can see here the three handlers `eventCreatedArtefact`, `eventStatusUpdate`, `eventStatusFinal`. Griffon provides some standard events, which are documented in the command line reference guide. For example the compile command fires the following events:

- `CompileStart` - Called when compilation starts, passing the kind of compile - source or tests
- `CompileEnd` - Called when compilation is finished, passing the kind of compile - source or tests

**Triggering events**
To trigger an event simply call the event() closure:

```
event("StatusFinal", ["Super duper plugin action complete!"])
```

**Common Events**
Below is a table of some of the common events that can be leveraged:

| Event | Parameters | Description |
|---|---|---|
| StatusUpdate | message | Passed a string indicating current script status/progress |
| StatusError | message | Passed a string indicating an error message from the current script |
| StatusFinal | message | Passed a string indicating the final script status message, i.e. when completing a target, even if the target does not exit the scripting environment |
| CreatedArtefact | artefactType,artefactName | Called when a create-xxxx script has completed and created an artifact |
| CreatedFile | fileName | Called whenever a project source filed is created, not including files constantly managed by Griffon |
| Exiting | returnCode | Called when the scripting environment is about to exit cleanly |
| PluginInstalled | pluginName | Called after a plugin has been installed |
| CompileStart | kind | Called when compilation starts, passing the kind of compile - source or tests |
| CompileEnd | kind | Called when compilation is finished, passing the kind of compile - source or tests |
| DocStart | kind | Called when documentation generation is about to start - javadoc or groovydoc |
| DocEnd | kind | Called when documentation generation has ended - javadoc or groovydoc |

## 4.4 Customising the build

Griffon is most definitely an opinionated framework and it prefers convention to configuration, but this doesn't mean you *can't* configure it. In this section, we look at how you can influence and modify the standard Griffon build.

**The defaults**
In order to customize a build, you first need to know *what* you can customize. The core of the Griffon build configuration is the `griffon.util.BuildSettings` class, which contains quite a bit of useful information. It controls where classes are compiled to, what dependencies the application has, and other such settings.
Here is a selection of the configuration options and their default values:

| Property | Config option | Default value |
|---|---|---|
| griffonWorkDir | griffon.work.dir | $USER_HOME/.griffon/<griffonVersion> |
| projectWorkDir | griffon.project.work.dir | <griffonWorkDir>/projects/<baseDirName> |
| classesDir | griffon.project.class.dir | <projectWorkDir>/classes |
| testClassesDir | griffon.project.test.class.dir | <projectWorkDir>/test-classes |
| testReportsDir | griffon.project.test.reports.dir | <projectWorkDir>/test/reports |
| resourcesDir | griffon.project.resource.dir | <projectWorkDir>/resources |
| projectPluginsDir | griffon.plugins.dir | <projectWorkDir>/plugins |

The `BuildSettings` class has some other properties too, but they should be treated as read-only:

| Property | Description |
|---|---|
| baseDir | The location of the project. |
| userHome | The user's home directory. |
| griffonHome | The location of the Griffon installation in use (may be null). |
| griffonVersion | The version of Griffon being used by the project. |
| griffonEnv | The current Griffon environment. |
| compileDependencies | A list of compile-time project dependencies as `File` instances. |
| testDependencies | A list of test-time project dependencies as `File` instances. |
| runtimeDependencies | A list of runtime-time project dependencies as `File` instances. |

Of course, these properties aren't much good if you can't get hold of them. Fortunately that's easy to do: an instance of `BuildSettings` is available to your scripts via the `griffonSettings` script variable. You can also access it from your code by using the `griffon.util.BuildSettingsHolder` class, but this isn't recommended.

**Overriding the defaults**
All of the properties in the first table can be overridden by a system property or a configuration option - simply use the "config option" name. For example, to change the project working directory, you could either run this command:

```
griffon -Dgriffon.project.work.dir=work compile
```

or add this option to your `griffon-app/conf/BuildConfig.groovy` file:

```
griffon.project.work.dir = "work"
```

Note that the default values take account of the property values they depend on, so setting the project working directory like this would also relocate the compiled classes, test classes, resources, and plugins.
What happens if you use both a system property and a configuration option? Then the system property wins because it takes precedence over the `BuildConfig.groovy` file, which in turn takes precedence over the default values.

**Available build settings**

| Name | Description |
|---|---|
| griffon.compiler.dependencies | Legacy approach to adding extra dependencies to the compiler classpath. Set it to a closure containing "fileset()" entries. |
| griffon.testing.patterns | A list of Ant path patterns that allow you to control which files are included in the tests. The patterns should not include the test case suffix, which is set by the next property. |
| griffon.testing.nameSuffix | By default, tests are assumed to have a suffix of "Tests". You can change it to anything you like but setting this option. For example, another common suffix is "Test". |

## 4.5 Command Tools Integration

If all the other projects in your team or company are built using a standard build tool such as Ant or Maven, you become the black sheep of the family when you use the Griffon command line to build your application. Fortunately, you can easily integrate the Griffon build system into the main build tools in use today (well, the ones in use in Java projects at least).

**Ant Integration**
When you invoke the integrate-with command with the -ant option enabled

```
griffon integrate-with --ant
```

Griffon creates an Apache Ant `build.xml` file for you containing the following targets:

- `clean` - Cleans the Griffon application
- `debug-app` - Runs the application in debug mode
- `test` - Runs the unit tests
- `run-app` - Equivalent to "griffon run-app"
- `run-applet` - Equivalent to "griffon run-applet"
- `run-webstart` - Equivalent to "griffon run-webstart"
- `dist` - Packages the application for production

Each of these can be run by Ant, for example:

```
ant clean
```

The build file is all geared up to use Apache Ivy for dependency management, which means that it will automatically download all the requisite Griffon JAR files and other dependencies on demand. You don't even have to install Griffon locally to use it! That makes it particularly useful for continuous integration systems such as CruiseControl or Jenkins

It uses the Griffon Ant task to hook into the existing Griffon build system. The task allows you to run any Griffon script that's available, not just the ones used by the generated build file. To use the task, you must first declare it:

```
<taskdef name="griffonTask"
         classname="griffon.ant.GriffonTask"
         classpathref="griffon.classpath"/>
```

This raises the question: what should be in "griffon.classpath"? The task itself is in the "griffon-cli" JAR artifact, so that needs to be on the classpath at least. You should also include the "groovy-all" JAR. With the task defined, you just need to use it! The following table shows you what attributes are available:

| Attribute | Description | Required |
|---|---|---|
| home | The location of the Griffon installation directory to use for the build. | Yes, unless classpath is specified. |
| classpathref | Classpath to load Griffon from. Must include the "griffon-bootstrap" artifact and should include "griffon-scripts". | Yes, unless `home` is set or you use a `classpath` element. |
| script | The name of the Griffon script to run, e.g. "TestApp". | Yes. |
| args | The arguments to pass to the script, e.g. "-unit -xml". | No. Defaults to "". |
| environment | The Griffon environment to run the script in. | No. Defaults to the script default. |
| includeRuntimeClasspath | Advanced setting: adds the application's runtime classpath to the build classpath if true. | No. Defaults to true. |

The task also supports the following nested elements, all of which are standard Ant path structures:

- `classpath` - The build classpath (used to load Gant and the Griffon scripts).
- `compileClasspath` - Classpath used to compile the application's classes.
- `runtimeClasspath` - Classpath used to run the application and package the WAR. Typically includes everything in @compileClasspath.
- `testClasspath` - Classpath used to compile and run the tests. Typically includes everything in `runtimeClasspath`.

How you populate these paths is up to you. If you are using the `home` attribute and put your own dependencies in the `lib` directory, then you don't even need to use any of them. For an example of their use, take a look at the generated Ant build file for new apps.

**Maven Integration**
TBD

**Gradle Integration**
When you invoke the integrate-with command with the -gradle option enabled

```
griffon integrate-with --gradle
```

Griffon creates a Gradle `build.gradle` file for you. From here you can call the standard Gradle commands such as `clean`, `assemble` and `build` to build your application. You can also use `griffon` as a command prefix to execute any of the regular Griffon command targets such as

```
gradle griffon-run-app
```

## 4.6 The Griffon Wrapper

This neat feature lets you execute Griffon commands without having a previously installing Griffon in your environment. This is a perfect fit for running tests in a continuous integration environment like Jenkins as there are no other requirements than a matching JDK.
When an application or plugin are created you'll get also the hooks for calling the wrapper, even configuring it in case you need it to point to a different Griffon release. These files are

- `griffonw`
- `griffonw.bat`
- `wrapper/griffon-wrapper.jar`
- `wrapper/griffon-wrapper.properties`

The first 2 files define platform dependent launch scripts. The third file contains the required classes to bootstrap the wrapper itself. The last file defines the configuration that the wrapper requires to work properly.
The wrapper works in the same way as the Griffon command, this means you can feed it every single command target and parameter the Griffon command accepts, like the following ones

```
./griffonw run-app
```

Compiles and runs the application in standalone mode.

```
./griffonw list-plugin-updates -install
```

Displays a list of available updates for all plugins installed and proceeds to update them if the confirmation is successful.

## 4.7 Command Line Options

The following command line options only have meaning while building the project. They have no effect when running the application once it has been packaged.
It's worth noting that all of the following options can also be specified in either `griffon-app/conf/BuildConfig.groovy` (local to project) or `$USER_HOME/.griffon/settings.groovy` (global to all projects), with the caveat that values specified at the command prompt will have precedence over those specified in the config file.

### 4.7.1 Verbose Output

Scripts have the choice of printing to the console whenever they need to communicate something to the developer.

They would normally use a standard `println` sentence. Sometimes it's useful to know what a script is doing with further detail but it makes no sense to see that information every single time. A conditional output is required.
All scripts inherit a `debug()` closure that will print its argument to stdout if an only if the following flag is enabled: `griffon.cli.verbose`. As an example, the following script has two print outs

```
includeTargets << griffonScript("Init")
target(main: "The description of the script goes here!") {
    println 'Hello world!'
    debug 'Hello World (debug)'
}
setDefaultTarget(main)
```

Running the script without the flag will print out 'Hello World!' all the time but never the second one

```
$ griffon hello
Welcome to Griffon 0.9.5-rc2 - http://griffon.codehaus.org/
Licensed under Apache Standard License 2.0
Griffon home is set to: /usr/local/griffon
…
Environment set to development
Hello world!
```

The second message will only appear if you specify the verbose flag

```
$ griffon -Dgriffon.cli.verbose=true hello
Welcome to Griffon 0.9.5-rc2 - http://griffon.codehaus.org/
Licensed under Apache Standard License 2.0
Griffon home is set to: /usr/local/griffon
…
Environment set to development
Hello world!
[11/11/10 4:43:04 PM] Hello World (debug)
```

### 4.7.2 Disable AST Injection

Since Griffon 0.9.1 all artifacts now share a common interface (GriffonArtifact). They may implement additional interfaces that define their role in a better way. For example controllers implement GriffonController whereas models implement GriffonModel. Despite this, you are not forced to implement these interfaces yourself, the Griffon compiler can do the work for you. It will even inject the appropriate behavior to classes that extend from base types other than `Object`. All this is done by leveraging the powerful AST Transformations framework introduced in Groovy 1.6.
If this feature ever gets in the way then you can disable it with the following command flag

```
griffon -Dgriffon.disable.ast.injection=true compile
```

Be sure to clean the project before using this flag, otherwise some classes may still have the AST additions weaved into their bytecode.

### 4.7.3 Disable Default Imports

Another feature introduced in Griffon 0.9.1 is the ability to define default imports for artifacts and scripts.
If this feature proves to be a disadvantage then disable it with the following command flag

```
griffon -Dgriffon.disable.default.imports=true compile
```

### 4.7.4 Disable Conditional Logging Injection

Griffon 0.9.1 added a log property to all artifacts, and enabled logging on addons. Groovy 1.8 adds a new set of AST transformations, @Log being one of them. It's job is to transform an unguarded logging statement into a guarded one. Starting with 0.9.2, Griffon can do the same without the need of annotating artifacts or addons with @Log.
If this feature proves to be a disadvantage then disable it with the following command flag

```
griffon -Dgriffon.disable.logging.injection=true compile
```

### 4.7.5 Disable Threading Injection

Griffon 0.9.2 adds the option for all controller actions to be executed off the UI thread automatically. This feature breaks backward compatibility with previous releases.
In order to regain the previous behavior you can disable this feature by specifying the following command flag

```
griffon -Dgriffon.disable.threading.injection=true compile
```

### 4.7.6 Default Answer in Non Interactive Mode

Sometimes a command may require the user to specify a missing value. When the build is run in interactive mode (the default mode) then it's just a matter of typing the value in the console. However, if the build is run in non-interactive mode then it's very likely it will fail.
For this reason, the Griffon build accepts the definition of a default answer if the `griffon.noninteractive.default.answer` key is specified, like this

```
griffon -Dgriffon.noninteractive.default.answer=y release-plugin
```

Be warned that this setting applies to every single input asked by a command.

### 4.7.7 Plugin Install Failure Strategies

Failures may occur during plugin installation. It may be the case that a plugin could not be found in the configured repositories, or a JAR dependency failed to be resolved. When this happens the build will try its best cope with the error, usually by continuing installing remainder plugin dependencies (if any).
This behavior can be altered by specifying a value for `griffon.install.failure`. Accepted values are:

| Value | Description |
|---|---|
| abort | Aborts the installation sequence, even if there are other plugins left to be installed. It will also delete all installed plugins in the current session. |
| continue | Continues with the next plugin in the list. this is the default behavior. |
| retry | Retries failed plugins a second time. A second failure skips the plugin from being installed but does not affect any other plugins that have been successfully installed or are yet to be installed. |

For example, to return the build to its previous behavior (abort on failures) you'll type the following in your command prompt

```
griffon -Dgriffon.install.failure='abort' compile
```

### 4.7.8 Default Artifact Repository for Searching

The Griffon build assumes `griffon-legacy` to be the default Artifact Repository to be searched when querying

for artifacts (either to list them, get some info or install them). This setting can be altered by specifying a value for `griffon.artifact.repository.default.search`. The value must be a valid repository name available in the configuration files.
For example, a local repository identified by the name '`my-local-repo`' can be set as the default search repository like so

```
griffon -Dgriffon.artifact.repository.default.search='my-local-repo' install-plugin cool-plu
```

### 4.7.9 Default Artifact Repository for Caching

When a plugin or archetype is downloaded from an artifact repository the Griffon build will place a copy of it in the `griffon-local` repository. This speeds up searches and further plugin installations. If you would like to specify a different local repository to be used as a cache then define a value for the `griffon.artifact.repository.default.install` key.
Assuming that '`my-local-repo`' is configured in the project's settings then the following command will download the miglayout plugin and place a copy in that specific repository.

```
griffon -Dgriffon.artifact.repository.default.install='my-local-repo' install-plugin miglayo
```

### 4.7.10 Disable Automatic Local Repository Synchronization

Section 4.7.9 describes that copies of plugins and archetypes will be placed in a local repository whenever they are downloaded from other repositories. You can disable this feature altogether by specifying a value for `griffon.disable.local.repository.sync` as true, like the following example shows

```
griffon -Dgriffon.disable.local.repository.sync=true install-archetype scala
```

## 4.8 The Griffon Shell

Starting with Griffon 0.9.5 there's a new command line tool at your disposal: the Griffon Shell or `griffonsh` for short. This is an interactive shell that can be kept running in the background, this way you don't pay the penalty of starting a new JVM every time you invoke a command. Other benefits are the bypass of dependency resolution if dependencies have not changed from the last command invocation. Here's a sample usage session:

```
$ griffonsh
Welcome to Griffon 0.9.5-rc2 - http://griffon.codehaus.org/
Licensed under Apache Standard License 2.0
Griffon home is set to: /usr/local/griffon
Type 'exit' or ^D to terminate this interactive shell

griffon> compile
Resolving dependencies…
Dependencies resolved in 903ms.
Environment set to development
Resolving plugin dependencies …
Plugin dependencies resolved in 1502 ms.
    [mkdir] Created dir: /Users/joe/.griffon/0.9.5-rc2/projects/sample/classes/cli
    [mkdir] Created dir: /Users/joe/.griffon/0.9.5-rc2/projects/sample/classes/main
    [mkdir] Created dir: /Users/joe/.griffon/0.9.5-rc2/projects/sample/classes/test
    [mkdir] Created dir: /Users/joe/.griffon/0.9.5-rc2/projects/sample/test-classes
    [mkdir] Created dir: /Users/joe/.griffon/0.9.5-rc2/projects/sample/test-resources
    [mkdir] Created dir: /Users/joe/.griffon/0.9.5-rc2/projects/sample/resources
 [griffonc] Compiling 8 source files to /Users/joe/.griffon/0.9.5-rc2/projects/sample/classe
 [griffonc] Compiling 4 source files to /Users/joe/.griffon/0.9.5-rc2/projects/sample/classe
griffon> run-app
Resolving dependencies…
Dependencies resolved in 1ms.
    [mkdir] Created dir: /Users/joe/.griffon/0.9.5-rc2/projects/sample/resources/griffon-app
    [mkdir] Created dir: /Users/joe/.griffon/0.9.5-rc2/projects/sample/resources/griffon-app
     [copy] Copying 1 file to /Users/joe/.griffon/0.9.5-rc2/projects/sample/resources/griffo
     [copy] Copying 8 files to /Users/joe/.griffon/0.9.5-rc2/projects/sample/resources/griff
     [copy] Copying 1 file to /Users/joe/.griffon/0.9.5-rc2/projects/sample/classes/main
     [copy] Copying 11 files to /Users/joe/.griffon/0.9.5-rc2/projects/sample/resources
     [copy] Copied 8 empty directories to 8 empty directories under /Users/joe/.griffon/0.9.
     [copy] Copying 1 file to /projects/sample/staging
     [copy] Copying 1 file to /projects/sample/staging
     [copy] Copying 1 file to /projects/sample/staging
     [copy] Copying 1 file to /projects/sample/staging
     [copy] Copying 1 file to /projects/sample/staging
     [copy] Copying 1 file to /projects/sample/staging
     [copy] Copying 1 file to /projects/sample/staging
     [copy] Copying 1 file to /projects/sample/staging
Launching application …
2012-02-07 17:27:11,007 [main] INFO  griffon.swing.SwingApplication - Initializing all start
2012-02-07 17:27:16,555 [AWT-EventQueue-0] INFO  griffon.swing.SwingApplication - Shutdown i
   [delete] Deleting directory /projects/sample/staging/macosx64
   [delete] Deleting directory /projects/sample/staging/macosx
griffon> clean
Resolving dependencies…
Dependencies resolved in 1ms.
   [delete] Deleting directory /Users/joe/.griffon/0.9.5-rc2/projects/sample/classes/cli
   [delete] Deleting directory /Users/joe/.griffon/0.9.5-rc2/projects/sample/classes/main
   [delete] Deleting directory /Users/joe/.griffon/0.9.5-rc2/projects/sample/classes/test
   [delete] Deleting directory /Users/joe/.griffon/0.9.5-rc2/projects/sample/test-classes
   [delete] Deleting directory /Users/joe/.griffon/0.9.5-rc2/projects/sample/test-resources
   [delete] Deleting directory /Users/joe/.griffon/0.9.5-rc2/projects/sample/resources
   [delete] Deleting directory /projects/sample/staging
griffon>
```

This command environment accepts all commands available to the `griffon` command (except those that let you create a new project) plus a few more that are unique to the griffon shell. Please refer to the help command for more information on those extra commands.

# 5. Application Overview

## 5.1 Directory Structure

Here's a more detailed explanation of each directory within the application's structure

- `griffon-app` - top level directory for Groovy sources.
    - `conf` - Configuration sources.
        - `webstart` - Webstart resources.
        - `keys` - Jar signing keys.
        - `dist` - Package specific files.
            - `shared` - Common files to all packaging targets (like LICENSE.txt)
        - `metainf` - Files that should go in META-INF inside the application/addon jar.
    - `models` - Models.
    - `views` - Views.
    - `controllers` - Controllers.
    - `services` - Services.
    - `resources` - Images, properties files, etc.
    - `i18n` - Support for internationalization (i18n).
- `scripts` - Gant scripts.
- `src` - Supporting sources.
    - `main` - Other Groovy/Java sources.
- `test` - Unit and integration tests.
    - `unit` - Directory for unit tests.
    - `integration` - Directory for integration tests.
    - `cli` - Directory for command line tests (Scripts).

## 5.2 The MVC Pattern

All Griffon applications operate with a basic unit called the MVC group. An MVC group is comprised of 3 member parts: Models, Views and Controllers. However it is possible to add (or even remove) members from an MVC group by carefully choosing a suitable configuration.
MVC groups configuration is setup in `Application.groovy` located inside `griffon-app/conf`. This file holds an entry for every MVC group that the application has (not counting those provided by plugins/addons).
Here's how a typical MVC group configuration looks like

```
mvcGroups {
    // MVC Group for "sample"
    'sample' {
        model      = 'sample.SampleModel'
        view       = 'sample.SampleView'
        controller = 'sample.SampleController'
    }
}
```

The order of the members is very important, it determines the order in which each member will be initialized. In the previous example both `model` and `view` will be initialized before the `controller`. Do not mistake initialization for instantiation, as initialization relies on calling mvcGroupInit on the group member.
MVC group configurations accept a special key that defines additional configuration for that group, as it can be seen in the following snippet

```
mvcGroups {
    // MVC Group for "sample"
    'sample' {
        model      = 'sample.SampleModel'
        view       = 'sample.SampleView'
        controller = 'sample.SampleController'
    }
    // MVC Group for "foo"
    'foo' {
        model      = 'sample.FooModel'
        view       = 'sample.FooView'
        controller = 'sample.FooController'
        config {
            key = 'bar'
        }
    }
}
```

Values placed under this key become available to MVC members during the call to mvcGroupInit, as part of the arguments sent to that method. Here's how the `FooController` can reach the key defined in the configuration

```
class FooController {
    void mvcGroupInit(Map args) {
        println args.configuration.config.key
    }
}
```

While being able to set additional values under this key is a certainly an advantage it would probably be better if those values could be mutated or tweaked, probably treating them as variables, effectively making a group configuration work as a template. For that we'll have to discuss the mvcGroupManager first.

### 5.2.1 MVCGroupManager

This class is responsible for holding the configuration of all MVC groups no matter how they were defined, which can be either in `Application.groovy` or in an addon descriptor.
During the startup sequence an instance of `MVCGroupManager` will be created and initialized. Later the application will instruct this instance to create all startup groups as required. `MVCGroupManager` has a handful set of methods that deal with MVC group configuration alone; however those that deal with group instantiation come in 3 versions, with 2 flavors each (one Groovy, the other Java friendly).
Locating a group configuration is easily done by specifying the name you're interested in finding

```
def configuration = app.mvcGroupManager.findConfiguration('foo')
```

Once you have a configuration reference you can instantiate a group with it by calling any of the variants of the `create` method

```
def configuration = app.mvcGroupManager.findConfiguration('foo')
def group1 = configuration.create('foo1')
def group2 = configuration.create('foo2', [someKey: 'someValue'])
// the following will make the group's id match its name
def group3 = configuration.create()
def group4 = configuration.create(someKey: 'someValue')
```

Be careful that creating groups with the same name is usually not a good idea. The default MVCGroupManager will complain when this happens and will automatically spit out an exception. This behavior may be changed by setting a configuration key in `Config.groovy`

```
griffon.mvcid.collision = 'warning' // accepted values are 'warning', 'exception' (default)
```

The manager will log a warning and destroy the previously existing group before instantiating the new one when 'warning' is the preferred strategy
Now, even though you can create group instances based on their configurations the preferred way is to call any of createMVCGroup, buildMVCGroup or withMVCGroup methods. These methods are available to the app property every GriffonArtifact has, which points to the currently running application. Griffon artifacts also inherit these methods as part of their default contract. Finally, any class annotated with the MVCAware AST transformation will also gain access to these methods.
Groups will be available by id regardless of how they were instantiated. You can ask the mvcGroupManager for a particular group at any time, for example

```
def g1 = app.mvcGroupManager.groups.foo1
def g2 = app.mvcGroupManager.findGroup('foo1')
def g3 = app.mvcGroupManager.foo1
assert g1 == g2
assert g1 == g3
```

It's also possible to query all models, views, controllers and builders on their own. Say you'd want to inspect all currently instantiated models, this is how it can be done

```
app.mvcGroupManager.models.each { model ->
    // do something with model
}
```

### 5.2.2 MVCGroups and Configuration

Now that you know there are several ways to instantiate MVC groups we can go back to customizing them.
The simples way is to pass in new values as part of the arguments map that mvcGroupInit receives, for example

```
def group = app.mvcGroupManager.buildMVCGroup('foo', [key: 'foo'])
```

However is you wish to use the special `config` key that every MVC group configuration may have then you must instantiate the group in the following way

```
def configuration = app.mvcGroupManager.cloneMVCConfiguration('foo', [key: 'someValue'])
def group = configuration.create()
```

Note that you can still send custom arguments to the `create()` method.

### 5.2.3 Configuration Options

The following options are available to all MVC groups as long as you use the `config` key.

**Disabling Lifecycle Events**
Every MVC group triggers a few events during the span of its lifetime. These events will be sent to the event bus even if no component is interested in handling them. There may be times when you don't want these events to be placed in the event bus in order to speed up group creation/destruction. Use the following configuration to gain this effect:

```
mvcGroups {
    // MVC Group for "sample"
    'sample' {
        model      = 'sample.SampleModel'
        view       = 'sample.SampleView'
        controller = 'sample.SampleController'
        config {
            events {
                lifecycle = false
            }
        }
    }
}
```

The following events will be disabled with this setting:

- `InitializeMVCGroup`
- `CreateMVCGroup`
- `DestroyMVCGroup`

**Disabling Instantiation Events**
The Griffon runtime will trigger an event for every artifact it manages. As with the previous events this one will be sent to the event bus even if no component handles it. Skipping publication of this event may result in a slight increase of speed during group instantiation. Use the following configuration to gain this effect:

```
mvcGroups {
    // MVC Group for "sample"
    'sample' {
        model      = 'sample.SampleModel'
        view       = 'sample.SampleView'
        controller = 'sample.SampleController'
        config {
            events {
                instantiation = false
            }
        }
    }
}
```

The following events will be disabled with this setting:

- `NewInstance`

**Disabling Controllers as Application Event Listeners**
Controllers are registered as application event handlers by default when a group in instantiated. This makes it very convenient to have them react to events placed in the application's event bus. However you may want to avoid this automatic registration altogether, as it can lead to performance improvements. You can disable this feature with the following configuration:

```
mvcGroups {
    // MVC Group for "sample"
    'sample' {
        model      = 'sample.SampleModel'
        view       = 'sample.SampleView'
        controller = 'sample.SampleController'
        config {
            events {
                listener = false
            }
        }
    }
}
```

You can still manually register a controller as an application event handler at any time, with the caveat that it's now your responsibility to unregister it when the time is appropriate, most typically during the group's destroy sequence.

## 5.3 Application Lifecycle

Every Griffon application goes through the same life cycle phases no matter in which mode they are running, with the exception of applet mode where there is an additional phase due to the intrinsic nature of applets. The application's lifecycle has been inspired by JSR-296, the Swing Application Framework.
Every phase has an associated life cycle script that will be invoked at the appropriate time. These scripts are guaranteed to be invoked inside the UI thread (the Event Dispatch Thread in Swing). The script names match each phase name; you'll find them inside `griffon-app/lifecycle`.

### 5.3.1 Initialize

The initialization phase is the first to be called by the application's life cycle. The application instance has just been created and its configuration has been read. No other artifact has been created at this point, which means that event publishing and the `ArtifactManager` are not yet available to the script's binding.
This phase is typically used to tweak the application for the current platform, including its Look & Feel.
Addons will be initialized during this phase.

> The `Initialize` script will be called right after the configuration has been read but before addons are initialized. You **wont** have access to addon contributions.

### 5.3.2 Startup

This phase is responsible for instantiating all MVC groups that have been defined in the application's configuration ( `Application.groovy`) and that also have been marked as startup groups in the same configuration file.

> The `Startup` script will be called **after** all MVC groups have been initialized.

### 5.3.3 Ready

This phase will be called right after Startup with the condition that no pending events are available in the UI queue. The application's main frame will be displayed at the end of this phase.

### 5.3.4 Shutdown

Called when the application is about to close. Any artifact can invoke the shutdown sequence by calling `shutdown()` on the `app` instance.

> The `Shutdown` script will be called before any `ShutdownHandler` or event handler interested in the `ShutdownStart` event.

### 5.3.5 Stop

This phase is only available when running on applet mode. It will be called when the applet container invokes `destroy()` on the applet instance.

## 5.4 Application Events

Applications have the ability to publish events from time to time to communicate that something of interest has happened at runtime. Events will be triggered by the application during each of its life cycle phases, also when MVC groups are created and destroyed.

> All application event handlers are guaranteed to be called in the same thread that originated the event.

Any artifact or class can trigger an application event, by routing it through the reference to the current running application instance. All artifacts posses an instance variable that points to that reference. All other classes can use ApplicationHolder to gain access to the current application's instance.
Publishing an event can be done synchronously on the current thread or asynchronously relative to the UI thread. For example, the following snippet will trigger an event that will be handled in the same thread, which could be the UI thread itself

```
app.event('MyEventName', ['arg0', 'arg1'])
```

Whereas the following snippet guarantees that all event handlers that are interested in an event of type `MyEventName` will be called outside of the UI thread

```
app.eventOutside('MyEventName', ['arg0', 'arg1'])
```

Finally, if you'd want event notification to be handed in a thread that is not the current one (regardless if the current one is the UI thread or not) then use the following method

```
app.eventAsync('MyEventName', ['arg0', 'arg1'])
```

There may be times when event publishing must be stopped for a while. If that's the case then you can instruct the application to stop delivering events by invoking the following code

```
app.eventPublishingEnabled = false
```

Any events sent through the application's event bus will be discarded after that call; there's no way to get them back or replay them. When it's time to enable the event bus again simply call

```
app.eventPublishingEnabled = true
```

### 5.4.1 Life Cycle Events

The following events will be triggered by the application during each one of its phases

- `Log4jConfigStart[config]` - during the Initialize phase.
- `BootstrapStart[app]` - after logging configuration has been setup, during the Initialize phase.
- `BootstrapEnd[app]` - at the end of the Initialize phase.
- `StartupStart[app]` - at the beginning of the Startup phase.
- `StartupEnd[app]` - at the end of the Startup phase.
- `ReadyStart[app]` - at the beginning of the Startup phase.
- `ReadyEnd[app]` - at the end of the Startup phase.
- `ShutdownRequested[app]` - before the Shutdown begins.
- `ShutdownAborted[app]` - if a Shutdown Handler prevented the application from entering the Shutdown phase.
- `ShutdownStart[app]` - at the beginning of the Shutdown phase.

### 5.4.2 Artifact Events

The following events will be triggered by the application when dealing with artifacts

- `NewInstance[klass, type, instance]` - when a new artifact is created.
- `LoadAddonsStart[app]` - before any addons are initialized, during the Initialize phase.

- `LoadAddonsEnd[app, addons]` - after all addons have been initialized, during the Initialize phase. `addons` is a Map of <name, instance> pairs.
- `LoadAddonStart[name, addon, app]` - before an addon is initialized, during the Initialize phase.
- `LoadAddonEnd[name, addon, app]` - after an addon has been initialized, during the Initialize phase.

These events will be triggered when dealing with MVC groups

- `InitializeMVCGroup[configuration, group]` - when a new MVC group is initialized. `configuration` is of type MVCGroupConfiguration; `group` is of type MVCGroup.
- `CreateMVCGroup[group]` - when a new MVC group is created. `configuration` is of type MVCGroupConfiguration; `group` is of type MVCGroup.
- `DestroyMVCGroup[group]` - when an MVC group is destroyed. `group` is of type MVCGroup.

### 5.4.3 Miscellaneous Events

These events will be triggered when a specific condition is reached

- `UncaughtExceptionThrown[exception]` - when an uncaught exception bubbles up to GriffonExceptionHandler.
- `WindowShown[window]` - triggered by the WindowManager when a Window is shown.
- `WindowHidden[window]` - triggered by the WindowManager when a Window is hidden.

### 5.4.4 Custom Events

Any artifact that holds a reference to the current application may trigger events at its leisure by calling the `event()` or `eventAsync` methods on the application instance. The following example demonstrates how a Controller triggers a "Done" event after an action has finished

```
class MyController {
    def action = { evt = null ->
        // do some work
        app.event('Done')
    }
}
```

There are two versions of the `event()` method. The first takes just the name of the event to be published; the second accepts an additional argument which should be a List of parameters to be sent to every event handler. Event handlers notified by this method are guaranteed to process the event in the same thread that published it. However, if what you need is to post a new event and return immediately then use the `eventAsync` variants. If you want the event to be handled outside of the UI thread then use the `eventOutsideUI()` variants.

### 5.4.5 Event Handlers

Any artifact or class that abides to the following conventions can be registered as an application listener, those conventions are:

- it is a Script, class, Map, RunnableWithArgs or closure.
- in the case of scripts or classes, they must define an event handler whose name matches **on<EventName>**, this handler can be a method, RunnableWithArgs or a closure property.
- in the case of a Map, each key maps to <EventName>, the value must be a RunnableWithArgs or a closure.
- scripts, classes and maps can be registered/unregistered by calling `addApplicationListener`/ `removeApplicationListener` on the app instance.
- RunnableWithArgs and closure event handlers must be registered with an overloaded version of `addApplicationListener`/`removeApplicationListener` that takes **<EventName>** as the first parameter, and the runnable/closure itself as the second parameter.

There is a general, per application, script that can provide event handlers. If you want to take advantage of this feature you must define a script named `Events.groovy` inside `griffon-app/conf`. Lastly both Controller and Service instances are automatically registered as application event listeners. This is the only way to declare event listeners for `Log4jConfigStart` and `BootstrapStart` events.

> You can also write a class named `Events.java` in `src/main` as an alternative to `griffon-app/conf/Events.groovy`, but not both!

These are some examples of event handlers:

- Display a message right before default MVC groups are instantiated

File: griffon-app/conf/Events.groovy

```
onBootstrapEnd = { app ->
  println """Application configuration has finished loading.
MVC Groups will be initialized now."""
}
```

- Quit the application when an uncaught exception is thrown

File: src/main/Events.java

```
import griffon.util.ApplicationHolder;
public class Events {
    public void onUncaughtExceptionThrown(Exception e) {
        ApplicationHolder.getApplication().shutdown();
    }
}
```

- Print the name of the application plus a short message when the application is about to shut down.

File: griffon-app/controller/MyController.groovy

```
class MyController {
  def onShutdownStart = { app ->
    println "${app.config.application.title} is shutting down"
  }
}
```

- Print a message every time the event "Foo" is published

File: griffon-app/controller/MyController.groovy

```
class MyController {
  void mvcGroupInit(Map args) {
    app.addApplicationListener([
      Foo: {-> println 'got foo!' }
    ])
  }
  def fooAction = { evt = null ->
    // do something
    app.event('Foo')
  }
}
```

- An alternative to the previous example using a closure event handler

File: griffon-app/controller/MyController.groovy

```
class MyController {
  void mvcGroupInit(Map args) {
    app.addApplicationListener('Foo'){-> println 'got foo!' }
  }
  def fooAction = { evt = null ->
    // do something
    app.event('Foo')
  }
}
```

- Second alternative to the previous example using a RunnableWithArgs event handler

File: griffon-app/controller/MyController.java

```
import java.util.Map;
import griffon.util.RunnableWithArgs;
import org.codehaus.griffon.runtime.core.AbstractGriffonController;
public class MyController extends AbstractGriffonController {
  public void mvcGroupInit(Map<String, Object> params) {
    getApp().addApplicationListener("Foo", new RunnableWithArgs() {
        public void run(Object[] args) {
            System.out.println("got foo!");
        }
    });
  }
  public void fooAction() {
    // do something
    getApp().event("Foo");
  }
}
```

### 5.4.6 Custom Event Publishers

As the name implies application events are sent system wide. However there is an option to create localized event publishers. Griffon provides a @griffon.transform.EventPublisher AST transformation that you can apply to any class that wishes to be an event publisher.
This AST transformation will inject the following methods to the annotated classes:

- addEventListener(Object)
- addEventListener(String, Closure)
- addEventListener(String, RunnableWithArgs)
- removeEventListener(Object)
- removeEventListener(String, Closure)
- removeEventListener(String, RunnableWithArgs)
- publishEvent(String)
- publishEvent(String,List)
- publishEventOutsideUI(String)
- publishEventOutsideUI(String,List)
- publishEventAsync(String)
- publishEventAsync(String,List)
- isEventPublishingEnabled()
- setEventPublishingEnabled(boolean)

Event listeners registered with these classes should follow the same rules as application event handlers (they can be Scripts, classes, maps or closures, and so on).
The following example shows a trivial usage of this feature

```
@griffon.transform.EventPublisher
class Publisher {
   void doit(String name) {
      publishEvent('arg', [name])
   }
   void doit() {
      publishEvent('empty')
   }
}
class Consumer {
   String value
   void onArg(String arg) { value = 'arg = ' + arg }
   void onEmpty() { value = 'empty' }
}
p = new Publisher()
c = new Consumer()
p.addEventListener(c)
assert !c.value
p.doit()
assert c.value == 'empty'
p.doit('Groovy')
assert c.value == 'arg = Groovy'
```

## 5.5 Application Features

The GriffonApplication interface defines the base contract for all Griffon applications. However there are some meta enhancements done at runtime to all applications. The following methods become available before the Initialize phase is executed:

- MVC
  - newInstance
  - buildMVCGroup
  - createMVCGroup
  - destroyMVCGroup
  - withMVCGroup
- Threading
  - execInsideUISync
  - execInsideUIAsync
  - execOutsideUI
  - execFuture
  - isUIThread

### 5.5.1 Runtime Configuration

The application's runtime configuration is available through the `config` property of the application instance. This is a `ConfigObject` whose contents are obtained by merging `Application.groovy` and `Config.groovy`. Builder configuration is available through the `builderConfig` property and reflects the contents of `Builder.groovy`.
However starting with Griffon 0.9.2 there's an alternative for defining the application's configuration. You can now use properties files instead of Groovy scripts. If both properties files and Groovy scripts are available in the classpath then the settings of the scripts will be overriden by those set in the properties file. Each properties file must match the name of the configuration script. The following table shows the conventions

| Script File | Properties File |
|---|---|
| Application.groovy | Application.properties |
| Config.groovy | Config.properties |
| Builder.groovy | Builder.properties |

An application can change the name of the configuration script but it **can not** change the name of the configuration properties file.

### 5.5.2 Metadata

Access to the application's metadata file (`application.properties`) is available by querying the `griffon.util.Metadata` singleton. Here's a snippet of code that shows how to setup a welcome message that displays the application's name and version, along with its Griffon version

```
import griffon.util.Metadata
def meta = Metadata.current
application(title: "Some app", package: true) {
    gridLayout cols: 1, rows: 2
    label "Hello, I'm ${meta['app.name']}-${meta['app.version']}"
    label "Built with Griffon ${meta['app.griffon.version']}"
}
```

There are also a few helpful methods on this class

- `getApplicationName()` - same result as `meta['app.name']`
- `getApplicationVersion()` - same result as `meta['app.version']`
- `getGriffonVersion()` - same result as `meta['app.griffon.version']`
- `getGriffonStartDir()` - returns the value of `'griffon.start.dir'` from the System properties
- `getGriffonWorkingDir()` - returns a File that points to `'griffon.start.dir'` if the value is set and the file is writable, otherwise returns a File pointing to the current location if it is writable; if that fails then attempts to return a File pointing to `'user.dir'`; if all fail it will return the location to a temporal file, typically `'/tmp/${griffonAppName}'`.

### 5.5.3 Environment

A Griffon application can run in several environments, default ones being DEVELOPMENT, TEST and PRODUCTION. An application can inspect its current running environment by means of the `griifon.util.Environment` enum.
The following example enhances the previous one by displaying the current running environment

```
import griffon.util.Metadata
import griffon.util.Environment
def meta = Metadata.current
application(title: "Some app", package: true) {
    gridLayout cols: 1, rows: 3
    label "Hello, I'm ${meta['app.name']}-${meta['app.version']}"
    label "Built with Griffon ${meta['app.griffon.version']}"
    label "Current environment is ${Environment.current}"
}
```

### 5.5.4 Running Mode

Applications can run in any of the following modes: STANDALONE, WEBSTART or APPLET. The `griffon.util.RunMode` enum allows access to the current running mode.
This example extends the previous one by adding information on the current running mode

```
import griffon.util.Metadata
import griffon.util.Environment
import griffon.util.RunMode
def meta = Metadata.current
application(title: "Some app", package: true) {
    gridLayout cols: 1, rows: 3
    label "Hello, I'm ${meta['app.name']}-${meta['app.version']}"
    label "Built with Griffon ${meta['app.griffon.version']}"
    label "Current environment is ${Environment.current}"
    label "Current running mode is ${RunMode.current}"
}
```

### 5.5.5 Shutdown Handlers

Applications have the option to let particular artifacts abort the shutdown sequence and/or perform a task while the shutdown sequence is in process. Artifacts that desire to be part of the shutdown sequence should implement the `griffon.core.ShutdownHandler` interface and register themselves with the application instance.
The contract of a `ShutdownHandler` is very simple

- `boolean canShutdown(GriffonApplication app)` - return **`false`** to abort the shutdown sequence.
- `void onShutdown(GriffonApplication app)` - called if the shutdown sequence was not aborted.

There are no default ShutdownHandlers registered with an application.

### 5.5.6 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()` method on an application instance. Valid values are defined by the ApplicationPhase enum : `INITIALIZE`, `STARTUP`, `READY`, `MAIN` and `SHUTDOWN`.

### 5.5.7 Application Locale

Starting with Griffon 0.9 applications have a bound `locale` property that is initialized to the default Locale. Components can listen to Locale changes by registering themselves as PropertyChangeListeners on the application instance.

### 5.5.8 Default Imports

Since Griffon 0.9.1 default imports per artifacts are supported. All Groovy based artifacts will resolve classes from the `griffon.core` and `griffon.util` packages automatically, there is no longer a need to define imports on those classes unless you require an static import or define an alias. An example of this feature would be as follows.

```
class MyController {
    void mvcGroupInit(Map args) {
        println Metadata.current.'app.name'
    }
}
```

The `Metadata` class is defined in package `griffon.util`. There are additional imports per artifact type, here's the list of default definitions

- Model
    - groovy.beans -> @Bindable, @Vetoable
    - java.beans -> useful for all PropertyChange* classes
- View (when using Swing)
    - java.awt
    - java.awt.event
    - javax.swing
    - javax.swing.event
    - javax.swing.table
    - javax.swing.text
    - javax.swing.tree

The list of imports per artifacts can be tweaked or changed completely at will. You only need to specify a file named `META-INF/griffon-default-imports.properties` with the following format

```
<artifact_type> = <comma_separated_package_list>
```

These are the contents of the default file

```
views = javax.swing., javax.swing.event., javax.swing.table., javax.swing.text., javax.swing
models = groovy.beans., java.beans.
```

Imports are cumulative, this means you a package can't be removed from the default list provided by Griffon.

### 5.5.9 Startup Arguments

Command line arguments can be passed to the application and be accessed by calling `getStartupArgs()` on the application instance. This will return a copy of the args (if any) defined at the command line.
Here's a typical example of this feature in development mode

```
griffon run-app arg0 arg1 argn
```

Here's another example demonstrating that the feature can be used once the application has been packaged, in this case as a single jar

```
griffon dev package jar
java -jar dist/jars/app.jar arg0 arg1 argn
```

### 5.5.10 Locating Resources

Resources can be loaded form the classpath using the standard mechanism provided by the Java runtime, that is, ask a `ClassLoader` instance to load a resource `URL` or obtain an `InputStream` that points to the resource.
But the code can get quite verbose, take for example the following view code that locates a text file and displays it on a text component

```
scrollPane {
    textArea(columns: 40, rows: 20,
        text: this.class.classLoader.getResource('someTextFile.txt').text)
}
```

In order to reduce visual clutter, also to provide an abstraction over resource location, both `GriffonApplication` and `GriffonArtifact` have a new pair of methods that simply working with resources. Those methods are provided by `ResourceHandler`:

- `URL getResourceAsURL(String resourceName)`
- `InputStream getResourceAsStream(String resourceName)`
- `List<URL> getResources(String resourceName)`

Thus, the previous example can be rewritten this way

```
scrollPane {
    textArea(columns: 40, rows: 20,
        text: getResourceAsURL('someTextFile.txt').text)
}
```

In the future Griffon may switch to a modularized runtime, this abstraction will shield you from any problems when the underlying mechanism changes.
These methods can be attached to any non-artifact class at compile time if you apply the @griffon.transform.ResourcesAware AST transformation.

### 5.5.10 Uncaught Exceptions

There are times when an exception catches you off guard. The JVM provides a mechanism for handling these kind of exceptions: Thread.UncaughtExceptionHandler. You can register an instance that implements this interface with a Thread or ThreadGroup, however it's very likely that exceptions thrown inside the EDT will not be caught by such instance. Furthermore, it might be the case that other components would like to be notified when an uncaught exception is thrown. This is precisely what GriffonExceptionHandler does.
Stack traces will be sanitized by default, in other words, you won't see a long list containing Groovy internals. However you can bypass the filtering process and instruct Griffon to leave the stack traces untouched by specifying the following flag either in the command line with `-D` switch or in `Config.groovy`

```
griffon.full.stacktrace = true
```

Exceptions will be automatically logged using the `error` level. Should you choose to disable logging but still have some output when an exception occurs then configure the following flag

```
griffon.exception.output = true
```

Any exception caught by GriffonExceptionHandler will trigger a pair of events. The event names match the following convention

- Uncaught<exception.class.simpleName>
- UncaughtExceptionThrown

The exception is sent as the sole argument of these events. As an example, assume that a service throws an `IllegalArgumentException` during the invocation of a service method. This method was called from within a Controller which defines a handler for this exception, like this

```
class SampleService {
    void work() {
        throw new IllegalArgumentException('boom!')
    }
}
class SampleController {
    def sampleService
    def someAction = {
        sampleService.work()
    }
    def onUncaughtIllegalArgumentException = { iae ->
        // process exception
    }
}
```

As mentioned before, the name of an event handler matches the type of the exception it will handle, polymorphism is not supported. In other words, you wont be able to handle an `IllegalArgumentException` if you declare a handler for `RuntimeException`. You can however, rely on the second event triggered by this mechanism. Be aware that every single exception will trigger 2 events each time it is caught. It is best to use a synchronization approach to keep track of the last exception caught, like so

```
import groovy.transform.Synchronized
class SampleController {
    private lastCaughtException
    @Synchronized
    void onUncaughtRuntimeException(RuntimeException e) {
        lastCaughtException = e
        // handle runtime exception only
    }
    @Synchronized
    void onUncaughtExceptionThrown(e) {
        if(lastCaughtException == e) return
        lastCaughtException = e
        // handle any other exception types
    }
}
```

As a final remark, any exceptions that might occur during the handling of the events wont trigger `GriffonExceptionHandler` again, they will simply be logged and discarded instead.

## 5.6 Swing specific

The following features are available to Swing based applications only.

### 5.6.1 WindowManager

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.

**WindowManager DSL**
Starting with Griffon 0.9.2 there's a new DSL for configuring show/hide behavior per window. This configuration can be set in `griffon-app/conf/Config.groovy`, and here is how it looks

```
swing {
    windowManager {
        myWindowName = [
            show: {window, app -> … },
            hide: {window, app -> … }
        ]
        myOtherWindowName = [
            show: {window, app -> … }
        ]
    }
}
```

The name of each entry must match the value of the Window's name: property. Each entry may have the following options

- **show -** used to show the window to the screen. It must be a closure that takes two parameters: the window to display and the current application.
- **hide -** used to hide the window from the screen. It must be a closure that takes two parameters: the window to hide and the current application.
- **handler -** a custom `WindowDisplayHandler`.

The first two options have priority over the third one. If one is missing then the WindowManager will invoke the default behavior. There is one last option that can be used to override the default behavior provided to all windows

```
swing {
    windowManager {
        defaultHandler = new MyCustomWindowDisplayHandler()
    }
}
```

You can go a bit further by specifying a global show or hide behavior as shown in the following example

```
swing {
    windowManager {
        defaultShow = {window, app -> … }
        // defaultHide = {window, app -> … }
        someWindowName = [
            hide: {window, app -> … }
        ]
    }
}
```

**Custom WindowDisplayHandlers**
The following example shows how you can animate all managed 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.DefaultWindowDisplayHandler
import griffon.core.GriffonApplication
import griffon.effects.Effects
class Dropper extends DefaultWindowDisplayHandler {
    void show(Window window, GriffonApplication app) {
        SwingUtils.centerOnScreen(window)
        app.execOutsideUI {
            Effects.dropIn(window, wait: true)
        }
    }
    void hide(Window window, GriffonApplication app) {
        app.execOutsideUI {
            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()
}
```

Custom `WindowDisplayHandler` implementations set in this manner will be called for all managed windows. You'll loose the ability of using the WindowManager DSL.

Alternatively, you could specify an instance of `Dropper` as the default handler by changing the `WindowManager` 's configuration to

```
swing {
    windowManager {
        defaultHandler = new Dropper()
    }
}
```

The `WindowDisplayHandler` interface also defines show/hide methods that can manage `JInternalFrame` instances.

**Starting Window**
Previous to Griffon 0.9.2 the first window to be displayed during the Ready phase was determined by a simple algorithm: picking the first available window from the managed windows list. With 0.9.2 however, it's now possible to configure this behavior by means of the WindowManager DSL. Simply specify a value for `swing.windowManager.startingWindow`, like this

```
swing {
    windowManager {
        startingWindow = 'primary'
    }
}
```

This configuration flag accepts two types of values:

- a String that defines the name of the Window. You must make sure the Window has a matching name property.
- a Number that defines the index of the Window in the list of managed windows.

If no match is found then the default behavior will be executed.

## 5.7 Artifact API

The Artifact API provides introspection capabilities on the conventions set on each artifact type. The following sections explain further what you can do with this API.

### 5.7.1 Evaluating Conventions

Every Griffon application exposes all information about its artifacts and addons via a pair of helper classes

- `AddonManager` - used for all installed addons
- `ArtifactManager` - used for all remaining artifacts

**ArtifactManager**
The `ArtifactManager` class provides methods to evaluate the conventions within the project and internally stores references to all classes within a GriffonApplication using subclasses of GriffonClass class.
A `GriffonClass` represents a physical Griffon resources such as a controller or a service. For example to get all `GriffonClass` instances you can call:

```
app.artifactManager.allClasses.each { println it.name }
```

There are a few "magic" properties that the `ArtifactManager` instance possesses that allow you to narrow the type of artifact you are interested in. For example if you only need to deal with controllers you can do:

```
app.artifactManager.controllerClasses.each { println it.name }
```

Dynamic method conventions are as follows:

- `get*Classes` - Retrieves all the classes for a particular artifact type. Example `app.artifactManager.getControllerClasses()`.
- `*Classes` - Retrieves all the classes for a particular artifact type. Example `app.artifactManager.controllerClasses`.
- `is*Class` - Returns true if the given class is of the given artifact type. Example `app.artifactManager.isControllerClass(ExampleController)`

The `GriffonClass` interface itself provides a number of useful methods that allow you to further evaluate and work with the conventions. These include:

- `newInstance` - Creates a new instance of the enclosed class.
- `getName` - Returns the logical name of the class in the application without the trailing convention part if applicable
- `getClazz` - Returns the artifact class
- `getType` - Returns the type of the artifact, i.e "controller"
- `getTrailing` - Returns the suffix (if any) of the artifact, i.e "Controller"

For a full reference refer to the javadoc API.

**AddonManager**
The `AddonManager` class is responsible for holding references to all addons (which are of type griffon.core.GriffonAddon), as well as providing metainformation on each addon via an addon descriptor. The latter can be used to know at runtime the name and version of a particular addon, useful for building a dynamic About dialog for example.
All addons have the same behavior which is explained in detail in section 12.6 Addons.

### 5.7.2 Adding Dynamic Methods at Runtime

For Griffon managed classes like controllers, models and so forth you can add methods, constructors etc. using the ExpandoMetaClass mechanism by accessing each controller's MetaClass:

```
class ExampleAddon {
    def addonPostInit(app) {
        app.artifactManager.controllerClasses.each { controllerClass ->
             controllerClass.metaClass.myNewMethod = {-> println "hello world" }
        }
    }
}
```

In this case we use the `app.artifactManager` object to get a reference to all of the controller classes' MetaClass instances and then add a new method called `myNewMethod` to each controller. Alternatively, if you know before hand the class you wish add a method to you can simple reference that classes `metaClass` property:

```
class ExampleAddon {
    def addonPostInit(app) {
        String.metaClass.swapCase = {->
            def sb = new StringBuffer()
            delegate.each {
                sb << (Character.isUpperCase(it as char) ?
                       Character.toLowerCase(it as char) :
                       Character.toUpperCase(it as char))
            }
            sb.toString()
        }
        assert "UpAndDown" == "uPaNDdOWN".swapCase()
    }
}
```

In this example we add a new method `swapCase` to `java.lang.String` directly by accessing its `metaClass`.

### 5.7.3 Artifact Types

All Griffon artifacts share common behavior. This behavior is captured by an interface named griffon.core.GriffonArtifact. Additional interfaces with more explicit behavior exist per each artifact type. The following is a list of the basic types and their corresponding interface

- Model -> griffon.core.GriffonModel
- View -> griffon.core.GriffonView
- Controller -> griffon.core.GriffonController
- Service -> griffon.core.GriffonService

Starting with Griffon 0.9.1 the compiler will make sure that each artifact implements its corresponding interface via AST injection. This feature can be very useful when accessing artifacts from languages other than Groovy (see section 13.1 Dealing with Non-Groovy Artifacts to learn more about this kind of artifacts).

> AST injection is always enabled unless you disable it as explained in section 4.7.2 Disable AST Injection.

Additionally to each artifact type you will find a companion **GriffonClass** that is specialized for each type. These specialized classes can be used to discover metadata about a particular artifact. The following is a list of the companion GriffonClass for each of the basic artifacts found in core

- Model -> griffon.core.GriffonModelClass
- View -> griffon.core.GriffonViewClass
- Controller -> griffon.core.GriffonControllerClass
- Service -> griffon.core.GriffonServiceClass

Be aware that additional artifacts provided by plugins (such as Charts and Wizards) may provide their own interface and companion GriffonClass. These too will be available when querying the `ArtifactManager`.

## 5.8 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 application.
If no valid archetype is found then the default archetype will be used. Following is the default template for an application archetype

```
import griffon.util.Metadata
includeTargets << griffonScript('CreateMvc')
target(name: 'createApplicationProject',
        description: 'Creates a new application project',
        prehook: null, posthook: null) {
    createProjectWithDefaults()
    createMVC()
    // to install plugins do the following
    // Metadata md = Metadata.getInstance(new File("${basedir}/application.properties"))
    //
    // for a single plugin
    //     installPluginExternal md, pluginName, pluginVersion
    //        ** pluginVersion is optional **
    //
    // for multiple plugins where the latest version is preferred
    //     installPluginsLatest md, [pluginName1, pluginName2]
    //
    // for multiple plugins with an specific version
    //     installPlugins md, [pluginName1: pluginVersion1]
}
setDefaultTarget(createApplicationProject)
```

## 5.8.1 A Fancy Example

This section demonstrates how an archetype can be created and put to good use for building applications.

### #1 Create the archetype

The first step is to create the archetype project and its descriptor, which can be done by executing the following command

```
griffon create-archetype fancy
cd fancy
```

### #2 Tweak the archetype descriptor

Locate the archetype descriptor (`application.groovy`) and open it in your favorite editor, paste the following snippet

```
import griffon.util.Metadata
includeTargets << griffonScript('CreateMvc')
target(name: 'createApplicationProject',
       description: 'Creates a new application project',
       prehook: null, posthook: null) {
    createProjectWithDefaults()
    argsMap.model      = 'MainModel'
    argsMap.view       = 'MainView'
    argsMap.controller = 'MainController'
    createMVC()
    createArtifact(
        name:    mvcFullQualifiedClassName,
        suffix: 'Actions',
        type:   'MainActions',
        path:   'griffon-app/views')
    createArtifact(
        name:   mvcFullQualifiedClassName,
        suffix: 'MenuBar',
        type:   'MainMenuBar',
        path:   'griffon-app/views')
    createArtifact(
        name:   mvcFullQualifiedClassName,
        suffix: 'StatusBar',
        type:   'MainStatusBar',
        path:   'griffon-app/views')
    createArtifact(
        name:   mvcFullQualifiedClassName,
        suffix: 'Content',
        type:   'MainContent',
        path:   'griffon-app/views')
    Metadata md = Metadata.getInstance(new File("${basedir}/application.properties"))
    installPluginExternal md, 'miglayout'
}
setDefaultTarget(createApplicationProject)
```

This archetype overrides the default templates for Model, View and Controller that will be used for the initial MVC group. It also creates 4 additional files inside `griffon-app/views`. Finally it installs the latest version of the MigLayout plugin.

**#3 Create the artifact templates**

According to the conventions laid out in the archetype descriptor there must exist a file under `templates/artifacts` that matches each one of the specified artifact types, in other words we need the following files

**MainModel.groovy**

```
@artifact.package@import groovy.beans.Bindable
import griffon.util.GriffonNameUtils
class @artifact.name@ {
    @Bindable String status
    void mvcGroupInit(Map args) {
        status = "Welcome to ${GriffonNameUtils.capitalize(app.config.application.title)}"
    }
}
```

**MainController.groovy**

```
@artifact.package@class @artifact.name@ {
    def model
    def view
    // void mvcGroupInit(Map args) {
    //    // this method is called after model and view are injected
    // }
    // void mvcGroupDestroy() {
    //    // this method is called when the group is destroyed
    // }
    def newAction = { evt = null ->
        model.status = 'New action'
    }
    def openAction = { evt = null ->
        model.status = 'Open action'
    }
    def saveAction = { evt = null ->
        model.status = 'Save action'
    }
    def saveAsAction = { evt = null ->
        model.status = 'Save As action'
    }
    def aboutAction = { evt = null ->
        model.status = 'About action'
    }
    def quitAction = { evt = null ->
        model.status = 'Quit action'
    }
    def cutAction = { evt = null ->
        model.status = 'Cut action'
    }
    def copyAction = { evt = null ->
        model.status = 'Copy action'
    }
    def pasteAction = { evt = null ->
        model.status = 'Paste action'
    }
}
```

**MainView.groovy**

```
@artifact.package@build(@artifact.name.plain@Actions)
application(title: GriffonNameUtils.capitalize(app.config.application.title),
    pack: true,
    locationByPlatform:true,
    iconImage: imageIcon('/griffon-icon-48x48.png').image,
    iconImages: [imageIcon('/griffon-icon-48x48.png').image,
                 imageIcon('/griffon-icon-32x32.png').image,
                 imageIcon('/griffon-icon-16x16.png').image]) {
   menuBar(build(@artifact.name.plain@MenuBar))
   migLayout(layoutConstraints: 'fill')
   widget(build(@artifact.name.plain@Content), constraints: 'center, grow')
   widget(build(@artifact.name.plain@StatusBar), constraints: 'south, grow')
}
```

**MainActions.groovy**

```
@artifact.package@import groovy.ui.Console
actions {
   action( id: 'newAction',
      name: 'New',
      closure: controller.newAction,
      mnemonic: 'N',
      accelerator: shortcut('N'),
      smallIcon: imageIcon(resource:"icons/page.png", class: Console),
      shortDescription: 'New'
   )
   action( id: 'openAction',
      name: 'Open...',
      closure: controller.openAction,
      mnemonic: 'O',
      accelerator: shortcut('O'),
      smallIcon: imageIcon(resource:"icons/folder_page.png", class: Console),
      shortDescription: 'Open'
   )
   action( id: 'quitAction',
      name: 'Quit',
      closure: controller.quitAction,
      mnemonic: 'Q',
      accelerator: shortcut('Q'),
   )
   action( id: 'aboutAction',
      name: 'About',
      closure: controller.aboutAction,
      mnemonic: 'B',
      accelerator: shortcut('B')
   )
   action( id: 'saveAction',
      name: 'Save',
      closure: controller.saveAction,
      mnemonic: 'S',
      accelerator: shortcut('S'),
      smallIcon: imageIcon(resource:"icons/disk.png", class: Console),
      shortDescription: 'Save'
   )
   action( id: 'saveAsAction',
      name: 'Save as...',
      closure: controller.saveAsAction,
      accelerator: shortcut('shift S')
   )
   action(id: 'cutAction',
      name: 'Cut',
      closure: controller.cutAction,
      mnemonic: 'T',
      accelerator: shortcut('X'),
      smallIcon: imageIcon(resource:"icons/cut.png", class: Console),
      shortDescription: 'Cut'
   )
   action(id: 'copyAction',
      name: 'Copy',
      closure: controller.copyAction,
      mnemonic: 'C',
      accelerator: shortcut('C'),
      smallIcon: imageIcon(resource:"icons/page_copy.png", class: Console),
      shortDescription: 'Copy'
   )
   action(id: 'pasteAction',
      name: 'Paste',
      closure: controller.pasteAction,
      mnemonic: 'P',
      accelerator: shortcut('V'),
      smallIcon: imageIcon(resource:"icons/page_paste.png", class: Console),
      shortDescription: 'Paste'
   )
}
```

**MainMenuBar.groovy**

```
@artifact.package@import static griffon.util.GriffonApplicationUtils.*
menuBar = menuBar {
   menu(text: 'File', mnemonic: 'F') {
       menuItem(newAction)
       menuItem(openAction)
       separator()
       menuItem(saveAction)
       menuItem(saveAsAction)
       if( !isMacOSX ) {
           separator()
           menuItem(quitAction)
       }
   }
   menu(text: 'Edit', mnemonic: 'E') {
       menuItem(cutAction)
       menuItem(copyAction)
       menuItem(pasteAction)
   }
   if(!isMacOSX) {
       glue()
       menu(text: 'Help', mnemonic: 'H') {
           menuItem(aboutAction)
       }
   }
}
```

**MainContent.groovy**

```
@artifact.package@label('Main content')
```

**MainStatusBar.groovy**

```
@artifact.package@label(id: 'status', text: bind { model.status })
```

**#4 Package and install the archetype**

This step is easily done with a pair of command invocations

```
griffon package-archetype
griffon install-archetype target/package/griffon-fancy-0.1.zip
```

**#5 Use the archetype**

Now that the archetype has been installed all that is left is put it to good use, like this

```
griffon create-app sample --archetype=fancy
```

This command should generate the following files in the application directory

- griffon-app
    - controllers
        - sample
            - `SampleController`
    - models
        - sample
            - `sampleModel`
    - views
        - sample
            - `SampleActions`

- `SampleContent`
- `SampleMenuBar`
- `SampleStatusBar`
- `SampleView`

If you inspect the `application.properties` file you'll notice that the miglayout plugin has been installed too. Archetypes can be versioned, installed and released in the same way plugins are.

## 5.9 Platform Specific

The following sections outline specific tweaks and options available for a particular platform.

### 5.9.1 Tweaks for a Particular Platform

Griffon will automatically apply tweaks to the application depending on the current platform. However you have the option to specify a different set of tweaks. For example, the following configuration in `Config.groovy` specifies a different handler for `macosx`:

```
platform {
    handler = [
        macosx: 'com.acme.MyMacOSXPlatformHandler'
    ]
}
```

Now you only need to create such handler, like this:

```
package com.acme
import griffon.core.GriffonApplication
import griffon.util.PlatformHandler
class MyMacOSXPlatformHandler implements PlatformHandler {
    void handle(GriffonApplication app) {
        System.setProperty('apple.laf.useScreenMenuBar', 'true')
        …
    }
}
```

The following platform keys are recognized by the application in order to locate a particular handler: `linux`, `macosx`, `solaris` and `windows`.

### 5.9.2 MacOSX

Applications that run in Apple's MacOSX must adhere to an strict set of rules. We recommend you to have a look at Apple's (Human Interface Guidelines).
Griffon makes it easier to integrate with MacOSX by automatically registering a couple of System properties that make the applicaiton behave like a native one

- `apple.laf.useScreenMenuBar` - if set to true will force the application's menu bar to appear at the top. Griffon sets its value to true.
- `com.apple.mrj.application.apple.menu.about.name` - sets the name that will appear next to the `About` menu option.

Java applications running on MacOSX also have the option to register handlers for `About`, `Preferences` and `Quit` menu options. The default handlers will trigger an specific application event each. These events can be disabled with a command flag set in `griffon-app/conf/Config.groovy`. The following table outlines the events, flags and the default behavior when the flags are enabled

| Event | Fired when | Flag | Default behavior |
|---|---|---|---|
| OSXAbout | user activates About menu | osx.noabout | Default about dialog is displayed |
| OSXPrefs | user activates Preferences menu | osx.noprefs | No Preferences menu is available |
| OSXQuit | user activates Quit menu | osx.noquit | Application shutdowns immediately |

# 6. Models and Binding

This section describe models and all binding options.

## 6.1 Models

Models are very simple in nature. Their responsibility is to hold data that can be used by both Controller and View to communicate with each other. In other words, Models are **not** equivalent to domain classes.
Models can be observable by means of the **@Bindable** AST Transformation. This actually simplifies setting up bindings so that changes in the UI can automatically be sent to model properties and vice versa.
@Bindable will inject a `java.beans.PropertyChangeSupport` field and all methods required to make the model an observable class. It will also make sure that a `PropertyChangeEvent` is fired for each observable property whenever said property changes value.
The following is a list of all methods added by @Bindable

- `void addPropertyChangeListener(PropertyChangeListener listener)`
- `void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)`
- `void removePropertyChangeListener(PropertyChangeListener listener)`
- `void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)`
- `PropertyChangeListener[] getPropertyChangeListeners()`
- `PropertyChangeListener[] getPropertyChangeListeners(String propertyName)`
- `void firePropertyChange(String propertyName, Object oldValue, Object newValue)`

The following is a list of all methods added by @Vetoable

- `void addVetoableChangeListener(VetoableChangeListener listener)`
- `void addVetoableChangeListener(String propertyName, VetoableChangeListener listener)`
- `void removeVetoableChangeListener(VetoableChangeListener listener)`
- `void removeVetoableChangeListener(String propertyName, VetoableChangeListener listener)`
- `VetoableChangeListener[] getVetoableChangeListeners()`
- `VetoableChangeListener[] getVetoableChangeListeners(String propertyName)`
- `void fireVetoableChange(String propertyName, Object oldValue, Object newValue)`

Another annotation, @Listener, helps you register `PropertyChangeListeners` without so much effort. The following code

```
import griffon.transform.PropertyListener
import groovy.beans.Bindable
@PropertyListener(snoopAll)
class MyModel {
    def controller
    @Bindable String name
    @Bindable
    @PropertyListener({controller.someAction(it)})
    String lastname
    def snoopAll = { evt -> … }
}
```

is equivalent to this one

```
import groovy.beans.Bindable
import java.beans.PropertyChangeListener
class MyModel {
    def controller
    @Bindable String name
    @Bindable String lastname
    def snoopAll = { evt -> … }
    MyModel() {
        addPropertyChangeListener(snoopAll as PropertyChangeListener)
        addPropertyChangeListener('lastname', {
            controller.someAction(it)
        } as PropertyChangeListener)
    }
}
```

@PropertyListener accepts the following values

- in-place definition of a closure
- reference of a closure property defined in the same class
- a List of any of the previous two

### 6.2 Binding

Binding in Griffon is achieved by leveraging Java Beans' `PropertyChangeEvent` and their related classes, thus binding will work with any class that fires this type of event, regardless of its usage of @Bindable or not.

#### 6.2.1 Syntax

These are the three options for writing a binding using the `bind` node

- **Long**

The most complete of all three, you must specify both ends of the binding explicitly. The following snippet sets an unidirectional binding from `bean1.prop1` to `bean2.prop2`

```
bind(source: bean1, sourceProperty: 'prop1',
     target: bean2, targetProperty: 'prop2')
```

- **Contextual**

This type of binding can assume either the sources or the targets depending on the context. The following snippets set an unidirectional binding from `bean1.prop1` to `bean2.prop2`

  - Implicit source

```
bean(bean1, prop1: bind(target: bean2, targetProperty: 'prop2'))
```

  - Implicit target

```
bean(bean2, prop2: bind(source: bean1, sourceProperty: 'prop1'))
```

When used in this way, either `sourceProperty:` or `targetProperty:` can be omitted; the bind node's value will become the property name, in other words

```
bean(bean1, prop1: bind('prop2', target: bean2))
```

- **Short**

This type of binding is only useful for setting implicit targets. It expects a closure as the definition of the binding value

```
bean(bean2, prop2: bind{ bean1.prop1 })
```

### 6.2.2 Additional Properties

The following properties can be used with either the **long** or **contextual** binding syntax

- **mutual:**

Bindings are usually setup in one direction. If this property is specified with a value of `true` then a bidirectional binding will be created instead.

```
import groovy.beans.Bindable
import groovy.swing.SwingBuilder
class MyModel {
  @Bindable String value
}
def model = new MyModel()
def swing  = new SwingBuilder()
swing.edt {
  frame(title: 'Binding', pack: true, visible: true) {
    gridLayout(cols: 2, rows: 3)
    label 'Normal'
    textField(columns: 20, text: bind('value', target: model))
    label 'Bidirectional'
    textField(columns: 20, text: bind('value', target: model, mutual: true))
    label 'Model'
    textField(columns: 20, text: bind('value', source: model))
  }
}
```

Typing text on textfield #2 pushes the value to model, which in turns updates textfield #2 and #3, demonstrating that textfield #2 listens top model updates. Typing text on textfield #2 pushes the value to textfield #3 but not #1, demonstrating that textfield #1 is not a bidirectional binding.

- **converter:**

Transforms the value before it is sent to event listeners.

```
import groovy.beans.Bindable
import groovy.swing.SwingBuilder
class MyModel {
  @Bindable String value
}
def convertValue = { val ->
  '*' * val?.size()
}
def model = new MyModel()
def swing  = new SwingBuilder()
swing.edt {
  frame(title: 'Binding', pack: true, visible: true) {
    gridLayout(cols: 2, rows: 3)
    label 'Normal'
    textField(columns: 20, text: bind('value', target: model))
    label 'Converter'
    textField(columns: 20, text: bind('value', target: model, converter: convertValue))
    label 'Model'
    textField(columns: 20, text:  bind('value', source: model))
  }
}
```

Typing text on textfield #1 pushes the value to the model as expected, which you can inspect by looking at textfield #3. Typing text on textfield #2 however transform's every keystroke into an '*' character.

- **validator:**

Guards the trigger. Prevents the event from being sent if the return value is `false` or `null`.

```
import groovy.beans.Bindable
import groovy.swing.SwingBuilder
class MyModel {
  @Bindable String value
}
def isNumber = { val ->
  if(!val) return true
  try {
    Double.parseDouble(val)
  } catch(NumberFormatException e) {
    false
  }
}
def model = new MyModel()
def swing  = new SwingBuilder()
swing.edt {
  frame(title: 'Binding', pack: true, visible: true) {
    gridLayout(cols: 2, rows: 3)
    label 'Normal'
    textField(columns: 20, text: bind('value', target: model))
    label 'Converter'
    textField(columns: 20, text: bind('value', target: model, validator: isNumber))
    label 'Model'
    textField(columns: 20, text:  bind('value', source: model))
  }
}
```

You can type any characters on textfield #1 and see the result in textfield #3. You can only type numbers on textfield #2 and see the result in textfield #3

This type of validation is not suitable for semantic validation (a.k.a. constraints in domain classes). You would want to have a look at the Validation plugin.

- **sourceEvent:**

Maps a different event type, instead of `PropertyChangeEvent`.

- **sourceValue:**

Specify a value that may come from a different source. Usually found in partnership with `sourceevent`.

```
import groovy.beans.Bindable
import groovy.swing.SwingBuilder
class MyModel {
  @Bindable String value
}
def model = new MyModel()
def swing  = new SwingBuilder()
swing.edt {
  frame(title: 'Binding', pack: true, visible: true) {
    gridLayout(cols: 2, rows: 3)
    label 'Text'
    textField(columns: 20, id: 'tf1')
    label 'Trigger'
    button('Copy Text', id: 'bt1')
    bind(source: bt1,
         sourceEvent: 'actionPerformed',
         sourceValue: {tf1.text},
         target: model,
         targetProperty: 'value')
    label 'Model'
    textField(columns: 20, text:  bind('value', source: model))
  }
}
```

A contrived way to copy text from one textfield to another. The copy is performed by listening to `ActionEvents` pumped by the button.

# 7. Views

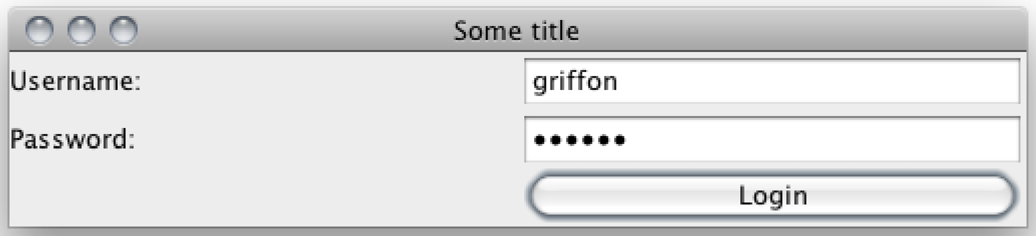
Views are responsible for defining how the application looks like. View scripts are always executed in the context of an UberBuilder, which means that Views have access to all nodes, properties and methods contributed by builders configured in `Builder.groovy`.
Views can reference directly both the `model` and `controller` instances that belong directly to their own MVC group.
View scripts are where you would usually setup bindings with their corresponding model instances.

## 7.1 Views and Swing

Views are usually written as Groovy scripts that create the UI by composing elements using builder nodes. Griffon supports all nodes provided by SwingBuilder by default. A typical View looks like this

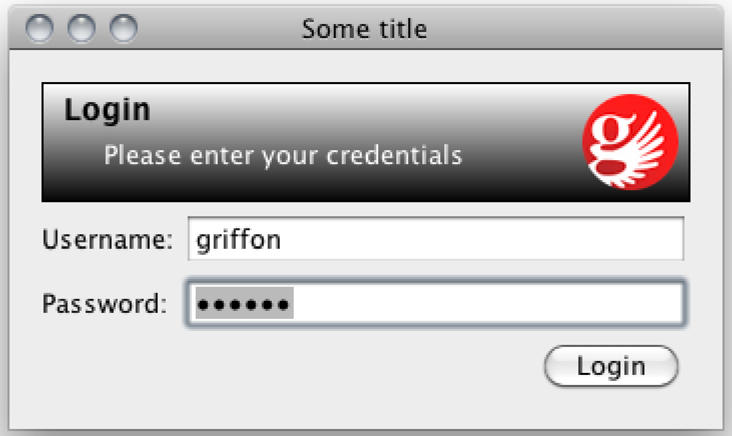
```
package login
actions {
    action(id: 'loginAction',
        name: 'Login',
        enabled: bind{ model.enabled },
        closure: controller.login)
}
application(title: 'Some title', pack:true,
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.png').image,
  iconImages: [imageIcon('/griffon-icon-48x48.png').image,
               imageIcon('/griffon-icon-32x32.png').image,
               imageIcon('/griffon-icon-16x16.png').image]) {
    gridLayout(cols: 2, rows: 3)
    label 'Username:'
    textField columns: 20, text: bind('username', target: model)
    label 'Password:'
    passwordField columns: 20, text: bind('password', target: model)
    label ''
    button loginAction
}
```

The resulting UI may look like this



It is pretty evident that changing layouts will greatly improve how this application looks. Additional nodes can be configured in `griffon-app/conf/Builder.groovy`, the Griffon runtime will make sure to setup the builder correctly. Here's an example with JideBuilder nodes used to setup a top banner. It also relies on MigLayout to arrange the components in a better way

```
package login
import java.awt.Color
actions {
    action(id: 'loginAction',
        name: 'Login',
        enabled: bind{ model.enabled },
        closure: controller.login)
}
application(title: 'Some title', pack:true,
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.png').image,
  iconImages: [imageIcon('/griffon-icon-48x48.png').image,
               imageIcon('/griffon-icon-32x32.png').image,
               imageIcon('/griffon-icon-16x16.png').image]) {
    migLayout(layoutConstraints: 'fill')
    bannerPanel(constraints: 'span 2, growx, wrap',
      title: 'Login',
      subtitle: 'Please enter your credentials',
      titleIcon: imageIcon('/griffon-icon-48x48.png'),
      border: lineBorder(color: Color.BLACK, thickness: 1),
      subTitleColor: Color.WHITE,
      background: new Color(0,0,0,1),
      startColor: Color.WHITE,
      endColor: Color.BLACK,
      vertical: true)
    label 'Username:', constraints: 'left'
    textField columns: 20, text: bind('username', target: model), constraints: 'wrap'
    label 'Password:', constraints: 'left'
    passwordField columns: 20, text: bind('password', target: model), constraints: 'wrap'
    button loginAction, constraints: 'span 2, right'
}
```



You'll need to install 2 plugins if you intend to run this application: jide-builder and miglayout. Here's the rest of the application, first the model

```
package login
import groovy.beans.Bindable
import griffon.transform.PropertyListener
@PropertyListener(enabler)
class LoginModel {
    @Bindable String username
    @Bindable String password
    @Bindable boolean enabled
    private enabler = { evt ->
        if(evt.propertyName == 'enabled') return
        enabled = username && password
    }
}
```

Then the controller

```
package login
import javax.swing.JOptionPane
class LoginController {
    def model
    def login = {
        JOptionPane.showMessageDialog(app.windowManager.windows[0],
            """
               username = $model.username
               password = $model.password
            """.stripIndent(14).toString())
    }
}
```

There are many plugins that will contribute additional nodes that can be used on Views.

### 7.2 Special Nodes

The rule of thumb to find out the node name of a Swing class is this:

- drop the first `J` from the class name
- uncapitalize the next character

Examples

- `JButton` => `button`
- `JLabel` => `label`

This rules apply to all Swing classes available in the JDK. There are a few additional nodes that provide a special function, which will be explained next.

#### 7.2.1 Application

Provided by: **Griffon**
This node defines a top level container depending on the current running mode. It it's `STANDALONE` or `WEBSTART` it will create a Window subclass according to the following rules:

- class name defined in `app.config.application.frameClass` (configured in `Application.groovy`)
- `JXFrame` if SwingX is available
- `JFrame` if all others fail

There's a slight change for the `APPLET` run mode, the container returned for the first invocation of the `application` node will be the applet itself, for all others the previous rules apply.
Of all the properties suggested by the default template you'll notice `iconImage` and `iconImages`. The first property is a standard property of `JFrame`. It's usually defines the icon to be displayed at the top of the frame (on platforms that support such setting). The second property (`iconImages`) is a Jdk6 addition to `java.awt.Window`. This property instructs the window to select the most appropriate icon according to platform preferences. Griffon ignores this setting if running in Jdk5. This property overrides the setting specified for `iconImage` if its supported in the current Jdk and platform.

#### 7.2.2 Container

Provided by: **SwingBuilder**
This is a pass through node that accepts any UI component as value. This node allows nesting of child content. It's quite useful when what you need is to embed a custom component for which a node is not available, for example

```
container(new MyCustomPanel()) {
    label 'Groovy is cool'
}
```

#### 7.2.3 Widget

Provided by: **SwingBuilder**
This is a pass through node that accepts any UI component as value. As opposed to `container`, this node **does not** allow nesting of child content. It's quite useful when what you need is to embed a custom component for which a node is not available, for example

```
widget(new MyCustomDisplay(), title: 'Groovy') {
```

### 7.2.4 Bean

Provided by: **SwingBuilder**
This is a catch-all node, it allows you to set properties on any object using the builder syntax, for example setting up bindings on a model

```
textField columns: 20, id: username
bean(model, value: bind{ username.text })
```

The previous code is equivalent to

```
textField columns: 20, text: bind('value', target: model)
```

### 7.2.5 Noparent

Provided by: **SwingBuilder**
Child nodes are always attached to their parents, there are times when you explicitly don't want that to happen. If that is the case then wrap those nodes with `noparent`

```
panel {
    gridLayout(cols: 2, rows: 2)
    button('Click 1', id: b1')
    button('Click 2', id: b2')
    button('Click 3', id: b2')
    button('Click 4', id: b4')
    // the following line will cause the buttons
    // to be reordered
    // bean(button1, text: 'Click 11')
    noparent {
        // this is safe, buttons do not change places
        bean(button1, text: 'Click 11')
    }
}
```

### 7.2.6 Root

Provided by: **Griffon**
Identifies the top level node of a secondary View script. View scripts are expected to return the top level node, however there may be times when further customizations prevent this from happening, for example wiring up a custom listener. When that happens the result has to be made explicit otherwise the script will return the wrong value. Using the `root()` node avoids forgetting this fact while also providing an alias for the node.
Secondary view script named "SampleSecondary"

```
root(
    tree(id: 'mytree')
)
mytree.addTreeSelectionModel(new DefaultTreeSelectionModel() {
    …
})
```

Primary view script named "SampleView"

```
build(SampleSecondary)
application(title: 'Sample') {
    borderLayout()
    label 'Options', constraints: NORTH
    widget root(SampleSecondary)
}
```

This node accepts an additional parameter `name` that can be used to override the default alias assigned to the node. If you specify a value for this parameter when the node is built then you'll need to use it again to retrieve the node.

# 8. Controllers and Services

This section describes the artifacts that hold the logic of a Griffon application.

## 8.1 Controllers

Controllers are the entry point for your application's logic. Each controller has access to their model and view instances from their respective MVC group.
Controller actions are usually defined using a closure property form, like the following one

```
class MyController {
    def someAction = { evt = null ->
        // do some stuff
    }
}
```

It is also possible to define actions as methods, however the closure property form is preferred (but not enforced). The caveat is that you would need to translate the method into a MethodClosure when referencing them form a View script. In the following example the action 'action1' is defined as a closure property, whereas the action 'action2' is defined as a method

```
application(title: 'Action sample', pack: true) {
    gridLayout(cols: 2, rows: 1) {
        button 'Action 1', actionPerformed: controller.action1
        button 'Action 2', actionPerformed: controller.&action2
    }
}
```

Actions must follow these rules in order to be considered as such:

- must have public (Java) or default (Groovy) visibility modifier.
- name does not match an event handler, i.e, it does not begin with `on`.
- must pass `GriffonClassUtils.isPlainMethod()` if it's a method.
- must have `void` as return type if it's a method.
- value must be a closure (including curried method pointers) if it's a property.

Controllers can perform other tasks:

- listen to application events.
- create and destroy MVC groups via a pair of methods (createMVCGroup, destroyMVCGroup).
- react to MVC initialization/destruction via a pair of methods (mvcGroupInit, mvcGroupDestroy).
- hold service references.

### 8.1.1 Threads and Actions

A key aspect that you must always keep in mind is proper threading. Often times controller actions will be bound in response to an event driven by the UI. Those actions will usually be invoked in the same thread that triggered the event, which would be the UI thread. When that happens you must make sure that the executed code is short and that it quickly returns control to the UI thread. Failure to do so may result in unresponsive applications.
The following example is the typical use case that must be avoided

```
class BadController {
    def badAction = {
        def sql = Sql.newInstance(
            app.config.datasource.url,
            model.username,
            model.password,
            app.config.datasource.driver
        )
        model.products.clear()
        sql.eachRow("select * from products") { product ->
            model.products << [product.id, product.name, product.price]
        }
        sql.close()
    }
}
```

There are two problems here. First the database connection is established inside the UI thread (which takes precious milliseconds or even longer), then a table (which could be arbitrarily large) is queried and each result sent to a List belonging to the model. Assuming that the list is bound to a Table Model then the UI will be updated constantly by each added row; which happens to be done all inside the UI thread. The application will certainly behave slow and sluggish, and to top it off the user won't be able to click on another button or select a menu item until this actions has been processed entirely.
Chapter 9 will discuss with further detail the options that you have at your disposal to make use of proper threading constructs. Here's a quick fix for the previous controller

```
class GoodController {
    def goodAction = {
        execOutsideUI {
            def sql = null
            try {
                sql = Sql.newInstance(
                    app.config.datasource.url,
                    model.username,
                    model.password,
                    app.config.datasource.driver
                )
                List results = []
                sql.eachRow("select * from products") { product ->
                    results << [product.id, product.name, product.price]
                }
                execInsideUIAsync {
                    model.products.clear()
                    model.addAll(results)
                }
            } finally {
                sql?.close()
            }
        }
    }
}
```

However starting with Griffon 0.9.2 you're no longer required to surround the action code with `execOutsideUI` as the compiler will do it for you. This feature breaks backward compatibility with previous releases so it's possible to disable it altogether. Please refer to section 4.7.5 Disable Threading Injection. This feature can be partially enabled/disabled too. You can specify with absolute precision which actions should have this feature enabled or disabled, by adding the following settings to `griffon-app/conf/BuildConfig.groovy`

```
compiler {
    threading {
        sample {
            SampleController {
                action1 = false
                action2 = true
            }
            FooController = false
        }
        bar = false
    }
}
```

The compiler will evaluate these settings as follows:

- the action identified by `sample.SampleController.action1` will not have automatic threading injected into its code, while `sample.SampleController.action2` (and any other found in the same controller) will have it.
- all actions belonging to `sample.FooController` will not have automatic threading injected.
- all actions belonging to all controllers in the `bar` package will not have threading injected.

> Automatic threading injection only works for Groovy based controllers. You must add appropriate threading code to controller actions that are written in languages other than Groovy.

### 8.2 Services

Services are responsible for the application logic that does not belong to a single controller. They are meant to be treated as singletons, injected to MVC members by following a naming convention. Services are optional artifacts, and as such there is no default folder created for them when a new application is created.
Services must be located inside the `griffon-app/services` directory with a `Service` suffix. The create-service command performs this job for you; also adding a unit test for the given service.
Let's say you need to create a Math service, the command to invoke would be

```
griffon create-service math
```

This results in the following files being created

- `griffon-app/services/MathService.groovy` - the service class.
- `test/unit/MathServiceTests.groovy` - service unit test.

A trivial implementation of an addition operation performed by the MathService would look like the following snippet

```
class MathService {
    def addition(a, b) { a + b }
}
```

Using this service from a Controller is a straight forward task. As mentioned earlier services will be injected by name, which means you only need to define a property whose name matches the uncapitalized service name, for example

```
class MyController {
    def mathService
    def action = {
        model.result = mathService.addition model.a, model.b
    }
}
```

The type of the service class is optional as basic injection is done by name alone.

> Service injection is trivial, it does not provide a full blown DI, in other words further service dependencies will not be resolved. You will need a richer DI solution in order to achieve this, fortunately there is a Spring plugin that does this and more.

Given that services are inherently treated as singletons they are also automatically registered as application event listeners. Be aware that services will be instantiated lazily which means that some events might not reach a particular

service if it has not been instantiated by the framework by the time of event publication. It also discouraged to use the @Singleton annotation on a Service class as it will cause trouble with the automatic singleton management Griffon has in place.
Lastly, all services instances will become available through an instance of type `griffon.core.ServiceManager`. This helper class exposes available services via a Map. You can query all currently available services in the following manner

```
app.serviceManager.services.each { name, instance ->
   // do something cool with services
}
```

You can also query for a particular service instance in the following way

```
def fooService = app.serviceManager.findService('foo')
```

It's worth mentioning that the previous method will instantiate the service if it wasn't available up to that point.
All services are instantiated lazily by default. You can change this behavior by adding a configuration flag to `Config.groovy`

```
griffon.services.eager.instantiation = true
```

# 9. Threading

Building a well-behaved multi-threaded desktop application has been a hard task for many years, however it does not have to be that way anymore. The following sections explain the threading facilities exposed by the Griffon framework.

> Prior to version 0.9.2 Controller actions were called in the same thread that published the event; most of the times this thread would be the UI thread. From 0.9.2 and onwards Controller actions will be executed outside of the UI thread. This feature can be disabled altogether or in a per case basis as explained in section 8.1.1.

## 9.1 Swing Threading

The Swing toolkit has a single golden rule: **all long computations must be performed outside of the Event Dispatch Thread** (or EDT for short). This rule also states that **all interaction with UI components must be done inside the EDT, including building a component and reading/writing component properties**. See Concurrency in Swing for more information.
Often times this rule can be broken easily as there is no compile time check for it. The Swing toolkit offers a helper class `SwingUtilities` that exposes a pair of method that let you run code inside the EDT, however there is no helper method for running code outside of the EDT
SwingBuilder provides a few methods that let you build multi-threaded applications the easy way. These methods are available in Views and Controllers.

### 9.1.1 Synchronous Calls

Synchronous calls inside the EDT can be achieved by calling the `edt{}` method. This method is smarter than plain `SwingUtilities.invokeAndWait` as it won't throw an exception if called inside the EDT, on the contrary, it will simply call the block of code it was given.
Example:

```
class MyController {
    def model
    def action1 = {
        // will be invoked inside the EDT by default (pre 0.9.2)
        def value = model.value
        Thread.start {
            // do some calculations
            edt {
                // back inside the EDT
                model.result = …
            }
        }
    }
    def action2 = {
        // will be invoked outside of the EDT by default (post 0.9.2)
        def value = model.value
        // do some calculations
        edt {
            // back inside the EDT
            model.result = …
        }
    }
}
```

### 9.1.2 Asynchronous Calls

Asynchronous calls inside the EDT can be made by calling the `doLater{}` method. This method simply posts a new event to the underlying EventQueue using `SwingUtilities.invokeLater`, meaning you spare a few characters and a class import.
Example:

```
class MyController {
    def model
    def action1 = {
        // will be invoked inside the EDT by default (pre 0.9.2)
        def value = model.value
        Thread.start {
            // do some calculations
            doLater {
                // back inside the EDT
                model.result = …
            }
        }
    }
    def action2 = {
        // will be invoked outside of the EDT by default (post 0.9.2)
        def value = model.value
        // do some calculations
        doLater {
            // back inside the EDT
            model.result = …
        }
    }
}
```

### 9.1.3 Outside Calls

The previous two examples showed a simple way to execute code outside of the EDT, simply put they spawn a new Thread. The problem with this approach is that creating new threads is an expensive operation, also you shouldn't need to create a new thread if the code is already being executed outside of the EDT.
The `doOutside{}` method takes these concerns into account, spawning a new thread if and only if the code is currently being executed inside the EDT. A rewrite of the previous example would be thus

```
class MyController {
    def model
    def action1 = {
        // will be invoked inside the EDT by default (pre 0.9.2)
        def value = model.value
        doOutside {
            // do some calculations
            doLater {
                // back inside the EDT
                model.result = …
            }
        }
    }
    def action2 = {
        // will be invoked outside of the EDT by default (post 0.9.2)
        def value = model.value
        // do some calculations
        doLater {
            // back inside the EDT
            model.result = …
            doOutside {
                // do more calculations
            }
        }
    }
}
```

## 9.2 Toolkit-agnostic Threading

Swing is not the only toolkit supported by Griffon. For those additional toolkits the three methods exposed in the previous sections (edt, doLater, doOutside) make no sense, however running code inside the UI thread in a synchronous/asynchronous way, as well as outside of it is something you must keep in mind.
The following sections outline toolkit-agnostic threading options, which can also be used with Swing in case you're wondering. These methods are available to all classes that implement the griffon.core.GriffonArtifact or griffon.core.GriffonApplication interfaces.

### 9.2.1 Synchronous Calls

Synchronous calls inside the UIThread are made by invoking the `execInsideUISync{}` method. This method is

equivalent to calling `edt{}` in Swing.
Example:

```
class MyController {
    def model
    def action1 = {
        // will be invoked inside the UI thread by default (pre 0.9.2)
        def value = model.value
        Thread.start {
            // do some calculations
            execInsideUISync {
                // back inside the UI thread
                model.result = …
            }
        }
    }
    def action2 = {
        // will be invoked outside of the UI thread by default (post 0.9.2)
        def value = model.value
        // do some calculations
        execInsideUISync {
            // back inside the UI thread
            model.result = …
        }
    }
}
```

### 9.2.2 Asynchronous Calls

Similarly to synchronous calls, asynchronous calls inside the UIThread are made by invoking the `execInsideUIAsync{}` method. This method is equivalent to calling `doLater{}` in Swing.
Example:

```
class MyController {
    def model
    def action1 = {
        // will be invoked inside the UI Thread by default (pre 0.9.2)
        def value = model.value
        Thread.start {
            // do some calculations
            execInsideUIAsync {
                // back inside the UI Thread
                model.result = …
            }
        }
    }
    def action2 = {
        // will be invoked outside of the UI Thread by default (post 0.9.2)
        def value = model.value
        // do some calculations
        execInsideUIAsync {
            // back inside the UI Thread
            model.result = …
        }
    }
}
```

### 9.2.3 Outside Calls

Making sure a block of code is executed outside the UIThread is made by invoking the `execOutsideUI{}` method. This method is equivalent to calling `doOutside{}` in Swing.
Example:

```
class MyController {
    def model
    def action1 = {
        // will be invoked inside the UI Thread by default (pre 0.9.2)
        def value = model.value
        execOutsideUI {
            // do some calculations
            execInsideUIAsync {
                // back inside the UI Thread
                model.result = …
            }
        }
    }
    def action2 = {
        // will be invoked outside of the UI Thread by default (post 0.9.2)
        def value = model.value
        // do some calculations
        execInsideUIAsync {
            // back inside the UI Thread
            model.result = …
            execOutsideUI {
                // do more calculations
            }
        }
    }
}
```

### 9.2.4 Additional Methods

There are two additional methods that complement the generic threading facilities that Griffon exposes to the application and its artifacts

- `isUIThread()` - returns true if the current thread is the UI Thread, false otherwise. Functionally equivalent to calling `SwingUtilities.isEventDispatchThread()` in Swing.
- `execFuture(ExecutorService s, Closure c)` - schedules a closure on the target ExecutorService. The executor service can be left unspecified, if so a default Thread pool executor (with 2 threads) will be used.
- `execFuture(ExecutorService s, Callable c)` - schedules a callable on the target ExecutorService. The executor service can be left unspecified, if so a default Thread pool executor (with 2 threads) will be used.

## 9.3 Annotation Based Threading

Starting with Griffon 0.9.2 there's also the possibility to define an specific thread execution policy for methods and properties via annotations

This feature is only available for Groovy code at the moment as it relies on the AST Transformation framework.

You must annotate a method or property with `@griffon.transform.Threading` and define a value of type `griffon.transform.Threading.Policy` (though the annotation uses `Threading.Policy.OUTSIDE_UITHREAD` by default). Annotated methods and properties must conform to these rules

- must be public.
- name does not match an event handler.
- must pass `GriffonClassUtils.isPlainMethod()` if it's a method.
- must have `void` as return type if it's a method.
- its value must be a closure (including curried method pointers) if it's a property.

Here's a trivial example

```
package sample
import griffon.transform.Threading
class Sample {
    @Threading
    void doStuff() {
        // executed outside of the UI thread
    }
    @Threading(Threading.Policy.INSIDE_UITHREAD_SYNC)
    void moreStuff() {
        // executed synchronously inside the UI thread
    }
    @Threading
    def work = {
        // executed outside of the UI thread
    }
    @Threading(Threading.Policy.INSIDE_UITHREAD_SYNC)
    def update = {
        // executed synchronously inside the UI thread
    }
}
```

It is worth noting that a `@Threading` annotation applied to a Controller's action/method will take precedence, this means you can force an specific threading policy on a Controller action other than the default one.

```
package sample
class SampleController {
    @Threading(Threading.Policy.INSIDE_UITHREAD_ASYNC)
    def popupDialog = {
        // build and show the dialog
    }
    def equivalentPopupDialog = {
        execInsideUIAsync {
            // build and show the dialog
        }
    }
}
```

# 10. Testing

Automated testing is seen as a key part of Griffon, implemented using Groovy Tests. Hence, Griffon provides many ways to making testing easier from low level unit testing to high level integration tests. This section details the different capabilities that Griffon offers in terms of testing.
The first thing to be aware of is that all of the `create-*` commands actually end up creating `unit` tests automatically for you. For example say you run the create-mvc command as follows:

```
griffon create-mvc com.yourcompany.yourapp.simple
```

Not only will Griffon create an MVC group with a controller at `griffon-app/controllers/com/yourcompany/yourapp/SimpleController.groovy`, but also an integration test at `test/integration/com/yourcompany/yourapp/SimpleControllerTests.groovy`. What Griffon won't do however is populate the logic inside the test! That is left up to you.

> As of Griffon 0.9, the suffix of `Test` is also supported for test cases.

**Running Tests**
Test are run with the test-app command:

```
griffon test-app
```

The above command will produce output such as:

```
-------------------------------------------------------
Running Unit Tests…
Running test FooTests...FAILURE
Unit Tests Completed in 464ms …
-------------------------------------------------------
Tests failed: 0 errors, 1 failures
```

Whilst reports will have been written out the `target/test-reports` directory.

> You can force a clean before running tests by passing `-clean` to the `test-app` command.

**Targeting Tests**

You can selectively target the test(s) to be run in different ways. To run all tests for a controller named `SimpleController` you would run:

```
griffon test-app SimpleController
```

This will run any tests for the class named `SimpleController`. Wildcards can be used...

```
griffon test-app *Controller
```

This will test all classes ending in `Controller`. Package names can optionally be specified...

```
griffon test-app some.org.*Controller
```

or to run all tests in a package...

```
griffon test-app some.org.*
```

or to run all tests in a package including subpackages...

```
griffon test-app some.org.**
```

You can also target particular test methods...

```
griffon test-app SimpleController.testLogin
```

This will run the `testLogin` test in the `SimpleController` tests. You can specify as many patterns in combination as you like...

```
griffon test-app some.org.* SimpleController.testLogin BookController
```

#### Targeting Test Types and/or Phases

In addition to targeting certain tests, you can also target test *types* and/or *phases* by using the `phase:type` syntax.

> Griffon organises tests by phase and by type. A test phase relates to the state of the Griffon application during the tests, and the type relates to the testing mechanism.
> Griffon comes with support for 3 test phases (`unit`, `integration`, and `other`) and JUnit test types for the `unit` and `integration` phases. These test types have the same name as the phase.
> Testing plugins may provide new test phases or new test types for existing phases. Refer to the plugin documentation.

To execute the JUnit `integration` tests you can run:

```
griffon test-app integration:integration
```

Both `phase` and `type` are optional. Their absence acts as a wildcard. The following command will run all test types in the `unit` phase:

```
griffon test-app unit:
```

The Griffon Spock Plugin is one plugin that adds new test types to Griffon. It adds a `spock` test type to the `unit` and`integration` phases. To run all spock tests in all phases you would run the following:

```
griffon test-app :spock
```

To run the all of the spock tests in the `integration` phase you would run...

```
griffon test-app integration:spock
```

More than one pattern can be specified...

```
griffon test-app unit:spock integration:spock
```

#### Targeting Tests in Types and/or Phases

Test and type/phase targetting can be applied at the same time:

```
griffon test-app integration: unit: some.org.**
```

This would run all tests in the `integration` and `unit` phases that are in the page `some.org` or a subpackage of.

### 10.1 Unit Testing

Unit testing are tests at the "unit" level. In other words you are testing individual methods or blocks of code without considering for surrounding infrastructure. The following is an unit test created using the default template

```
import griffon.test.*
class SomeUnitTests extends GriffonUnitTestCase {
    protected void setUp() {
        super.setUp()
    }
    protected void tearDown() {
        super.tearDown()
    }
    void testSomething() {
    }
}
```

You have access to all mocking facilities exposed by GriffonUnitTestCase within this test.

### 10.2 Integration Testing

Integration tests differ from unit tests in that you have full access to the Griffon application within the test. The following is an integration test created using the default template

```
import griffon.core.GriffonApplication
import griffon.test.*
class SomeControllerTests extends GriffonUnitTestCase {
    GriffonApplication app
    protected void setUp() {
        super.setUp()
    }
    protected void tearDown() {
        super.tearDown()
    }
    void testSomething() {
    }
}
```

As with unit tests, you have access to all mocking facilities exposed by GriffonUnitTestCase within this test, but you also have access to a full running Griffon application. By default this application is bootstrapped to the INITIALIZE phase. It's up to you to instruct the application to move to another phase depending on what you want to test (refer to `startup()`, `ready()`, `realize()` and `show()` methods).
The type of application to be run depends on the type of project and/or a configuration flag as explained next:

- if a configuration flag `griffon.application.mainClass` exists then its value will be used (assumes the value is a literal full qualified class).
- if the project is an addon then it will use `griffon.test.mock.MockApplication`
- finally it will fall back to `griffon.swing.SwingApplication`

### 10.3 Mocking

Mocking is but one of the many alternatives you have at your disposal to reduce complexity while setting up a test that requires a good number of components to be setup before actually testing the real class under test. Griffon provides a few mocking helper methods and classes, which will be discussed next.

#### 10.3.1 MockGriffonApplication

`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.

# 11. Packaging and Deployment

Griffon can package applications in several modes. There are 4 modes supported by default: zip, jar, webstart and applet.
To package an application use the package command. All modes will be used when calling the package command with no arguments. You can specify one or more packaging modes when executing the command. Packages will be place in their respective directory inside the `dist` directory located at the root of the application. You can configure a different default set of deployment targets that will be used when invoking this command without arguments. Simply add a configuration flag to `BuildConfig.groovy` like this

```
griffon.packaging = 'zip'
```

Now, any time you call the package command without arguments only the `zip` target will be executed.
It is possible to specify files that can be shared across packaging modes, like a README or a LICENSE file. Make sure to place them under `griffon-app/conf/dist/shared`.
Files that should be packed inside the application's jar META-INF directory must be placed in `griffon-app/conf/metainf`. This setting works for addons too.
Packaging an application will be executed in the **production** environment by default. You may specify a different environment as you would with other command. This setting impacts directly how webstart and applet modes are executed, as they will sign and pack all jars by default when in production mode. Please review and update your configuration if you desire a different behavior.
Each packaging target triggers a `PackageStart` and `PackageEnd` events, with their type as the single event parameter.

## 11.1 Zip

Packages the application using a conventional directory layout as found typically in Un*x packages. The directory layout is as follows:

- [root] - contains all files available at `griffon-app/conf/dist/shared`
  - bin - binary launchers (Windows and Un*x)
  - lib - application jars

The application launcher will bear the name of the application.
Run it with the following command

```
griffon package zip
```

Arguments: None
Configuration options: None

## 11.2 Jar

This is the simplest packaging mode available. It will package the application in a single jar file, by unpacking all dependencies and packing them once more in a sole file, so place close attention to potential duplicate entries, especially those found inside META-INF.

```
griffon package jar
```

Arguments:

- `name` - override the name of the generated jar file.

Configuration Options:

- `griffon.jars.jarName` - name of the application's main jar file.
- `griffon.dist.jar.nozip` - skip zipping the distribution if set to true.

There's a high chance of some files to have duplicates, e.g. griffon-artifacts.properties if you have installed a plugin that provides MVC groups. It's possible to instruct the build to merge duplicate files by specifying a regular expression and a merging strategy. The following table explains the different merging strategies available

| Strategy | Description |
|---|---|
| Skip | Do not perform any merge. Duplicate is discarded. |
| Replace | Duplicate is preferred and overwrites previous. |
| Append | Duplicate is appended at the end of previous. |
| Merge | Common lines found in duplicate are discarded. New lines found in duplicate are appended at the end. |
| MergeManifest | Duplicate keys override the previous ones. New keys are added to the merged result. |
| MergeProperties | Duplicate keys override the previous ones. New keys are added to the merged result. |
| MergeGriffonArtifacts | Merges artifact definitions per type. |

You can specify merging preferences in `BuildConfig.groovy` like this

```
griffon {
    jars {
        merge = [
            '.*.xml': org.codehaus.griffon.ant.taskdefs.FileMergeTask.Replace
        ]
    }
}
```

This setting will overwrite any XML file found in the path with the last version encountered as jars are processed. The griffon build defines a set of default mappings, which are the ones found in the next table

| Regexp | MergeStrategy |
|---|---|
| META-INF/griffon-artifacts.properties | MergeGriffonArtifacts |
| META-INF/MANIFEST.MF | MergeManifest |
| META-INF/services/.* | Merge |
| .*.properties | MergeProperties |

Merging preferences must be defined from the most specific to the least. Your preferences will override any default settings.

### 11.3 Webstart

Packages the application to be used in webstart mode. Will generate an appropriate JNLP file similarly as it's done when running the application in webstart mode.

```
griffon package webstart
```

Arguments:

- `codebase` - specify the codebase to be written in the JNLP file.

Configuration Options:

- `griffon.dist.webstart.nozip` - skip zipping the distribution if set to true.
- same configuration options used when running in webstart mode.

### 11.4 Applet

Packages the application to be used in applet mode. Will generate an appropriate JNLP and Html files similarly as it's done when running the application in applet mode.

```
griffon package applet
```

Arguments:

- `codebase` - specify the codebase to be written in the JNLP file.

Configuration Options:

- `griffon.dist.applet.nozip` - skip zipping the distribution if set to true.
- same configuration options used when running in applet mode.

## 11.5 Additional modes

If any of the afore mentioned packaging modes does not suite your needs you may use the Installer plugin to craft a better packaging option. This plugin supports the following additional modes:

- `izpack` - universal installer using Izpack.
- `mac` - for MacOSX.
- `rpm` - for rpm based Linux distributions.
- `deb` - for .deb based Linux distributions.
- `jsmooth` - for Windows.

You may call any of these modes as you would with the standard ones when the installer plugin is available, in other words

```
griffon package izpack
```

Many of these modes support additional configuration before generating the final package. It is a good idea to follow a two-step process

```
griffon prepare-izpack
// edit target/installer/izpack/resources/installer.xml
// and/or add more files to that directory
griffon create-izpack
```

Each additional packaging mode triggers 4 events with their type as the single event parameter: `PreparePackageStart`, `PreparePackageEnd`, `CreatePackageStart` and `CreatePackageEnd`.

## 11.6 Custom Manifest Entries

Griffon will automatically create the following entries in the application's manifest

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.1
Created-By: ${jvm.version} (${jvm.vendor})
Main-Class: ${griffonApplicationClass} | ${griffonAppletClass}
Built-By: ${user.name}
Build-Date: dd-MM-yyyy HH:mm:ss
Griffon-Version: ${griffonVersion}
Implementation-Title: capitalize(${griffonAppName})
Implementation-Version: ${appVersion}
Implementation-Vendor: capitalize(${griffonAppName})
```

There might be times when you must specify additional attributes or override existing ones. You can do this by adding a new block of configuration to `BuildConfig.groovy`, for example

```
griffon {
    jars {
        manifest = [
            'Foo': 'Bar'
            'Built-By': 'Acme'
        ]
    }
}
```

# 12. Plug-ins

Griffon provides a number of extension points that allow you to extend anything from the command line interface to the runtime configuration engine. The following sections detail how to go about it.

## 12.1 Creating and Installing Plug-ins

**Creating Plug-ins**
Creating a Griffon plugin is a simple matter of running the command:

```
griffon create-plugin [PLUGIN NAME]
```

This will create a plugin project for the name you specify. Say for example you run `griffon create-plugin example`. This would create a new plugin project called `example`.
The structure of a Griffon plugin is exactly the same as a regular Griffon project's directory structure, except that in the root of the plugin directory you will find a plugin Groovy file called the "plugin descriptor".
The plugin descriptor itself ends with the convention `GriffonPlugin` and is found in the root of the plugin project. For example:

```
class ExampleGriffonPlugin {
   def version = 0.1
   …
}
```

All plugins must have this class in the root of their directory structure to be valid. The plugin class defines the version of the plugin and optionally various hooks into plugin extension points (covered shortly).
You can also provide additional information about your plugin using several special properties:

- `title` - short one sentence description of your plugin
- `version` - the version of your plugin. Valid versions are for example "0.1", "0.2-SNAPSHOT", "0.1.4" etc.
- `griffonVersion` - The version of version range of Griffon that the plugin supports. eg. "1.1 > *"
- `license` - the plugin's license name in one sentence
- `pluginIncludes` - additional resources that should be included in the plugin zip
- `toolkits` - a list of supported toolkits [swing, javafx, swt, pivot, gtk]
- `platforms` - a list of supported platforms [linux, linux64, windows, windows64, macosx, macosx64, solaris, solaris64]
- `authosr` - a list of plugin author names/emails
- `description` - full multi-line description of plugin's features
- `documentation` - URL where plugin's documentation can be found
- `source` - URL where plugin's source can be found

Here is an example from Swing plugin :

```
class SwingGriffonPlugin {
    String version = '0.9.5'
    String griffonVersion = '0.9.5 > *'
    Map dependsOn = [:]
    List pluginIncludes = []
    String license = 'Apache Software License 2.0'
    List toolkits = ['swing']
    List platforms = []
    String documentation = ''
    String source = 'https://github.com/griffon/griffon-swing-plugin'
    List authors = [
        [
            name: 'Andres Almiray',
            email: 'aalmiray@yahoo.com'
        ]
    ]
    String title = 'Enables Swing support'
    String description = '''
Enables the usage of Swing based components in Views.
Usage
----
This plugin enables the usage of the following nodes inside a View.
...
Configuration
-------------
There's no special configuration for this plugin.
[1]: http://groovy.codehaus.org/Swing+Builder
'''
}
```

**Installing & Distributing Plugins**
To distribute a plugin you need to navigate to its root directory in a terminal window and then type:

```
griffon package-plugin
```

This will create a zip file of the plugin starting with `griffon-` then the plugin name and version. For example with the example plugin created earlier this would be `griffon-example-0.1.zip`. The `package-plugin` command will also generate `plugin.json` file which contains machine-readable information about plugin's name, version, author, and so on.
Once you have a plugin distribution file you can navigate to a Griffon project and type:

```
griffon install-plugin /path/to/plugin/griffon-example-0.1.zip
```

If the plugin is hosted on a remote HTTP server you can also do:

```
griffon install-plugin http://myserver.com/plugins/griffon-example-0.1.zip
```

**Releasing Plugins into a Griffon Artifact Repository**
To release a plugin call the `release-plugin` command while inside the plugin project. If no `repository` flag is specified then the default artifact repository (`griffon-central`) will be used. For quick testing purposes you can publish a release to `griffon-local` (which is always available) by issuing the following command

```
griffon install-plugin --repository=griffon-local
```

The aforementioned steps can be applied to archetypes too, you just need to change the command names from `package-plugin` to `package-archetype`; from `install-plugin` to `install-archetype`; from `release-plugin` to `release-archetype`.
Should you decide to become a plugin/archetype author and wish to publish your artifacts to the Griffon Central

repository then you must follow these steps:

- Create an account at http://artifacts.griffon-framework.org
- After confirming your email, log into your profile and click the button for membership request.
- Ping us at the developer mailing list or at @theaviary
- Once approved configure your credentials in `$USER_HOME/.griffon/settings.groovy` like this

```
griffon.artifact.repositories = [
    'griffon-central': [
        username: 'yourUsername',
        password: 'yourPassword'
    ]
]
```

## 12.2 Artifact Repositories

There are 3 types of plugin repositories: `local`, `remote` and `legacy`. Artifact repositories can be either configured locally to a project (inside `griffon-app/conf/BuildConfig`) or globally to all projects (inside `$USER_HOME/.griffon/settings.groovy`),

**Local Artifact Repositories**
This type of repository is file based and can be hosted anywhere in the file system, even on shared folders over the network. Local repositories makes it easier to share snapshot releases among team mates as the network latency should be smaller. Their configuration requires but one parameter to be specified: the path where the artifacts will be placed. Here's a sample configuration for a local repository named '`my-local-repo`'.

```
griffon.artifact.repositories = [
    'my-local-repo': [
        type: 'local',
        path: '/usr/local/share/griffon/repository'
    ]
]
```

There's a local repository available to you at all times. It's name is '`griffon-local`' and it's default path is `$USER_HOME/.griffon/repository`. This repository is the default place where downloaded plugins will be installed for speeding up retrievals at a later time.

**Remote Artifact Repositories**
This type of repository allows developers to publish releases via SCP or web. The repository is handled by a Grails application whose code is freely available at https://github.com/griffon/griffon-artifact-portal .
This code has been released under Apache Software License 2.0. Follow the instructions found in the README to run your own artifact portal. Configuring a remote repository requires a different set of properties than those exposed by local repositories. For example, if your organization is running a remote artifact repository located at `http://acme.com:8080/portal` then use the following configuration

```
griffon.artifact.repositories = [
    'acme': [
        type: 'remote',
        url: 'http://acme.com:8080/portal'
    ]
]
```

You may specify additional properties such as

```
griffon.artifact.repositories = [
    'acme': [
        type: 'remote',
        url: 'http://acme.com:8080/portal',
        username: 'wallace',
        password: 'gromit',
        port: 2345,
        timeout: 60
    ]
]
```

Where the following defaults apply

- port = 2222
- timeout = 30 (in seconds)

You may leave both `username` and `password` out however you will be asked for this credentials when publishing a release to this particular repository. Adding your credentials in the configuration avoids typing them when releasing artifacts.

**Legacy Artifact Repository**
This is a very special type of repository that exists only for backward compatibility during the migration of the old Griffon plugin repository to the new infrastructure in http://artifacts.griffon-framework.org .
There are no configuration options for this repository, neither you can publish a release to it; it's effectively read-only.

## 12.3 Understanding a Plugins Structure

As mentioned previously, a plugin is merely a project with an structure similar to a Griffon application with the addition of a contained plugin descriptor. However when installed, the structure of a plugin differs slightly. For example, take a look at this plugin directory structure:

```
+ griffon-app
     + controllers
     + models
     + views
     …
 + lib
 + src
     + main
     + cli
     + doc
```

Essentially when a plugin is installed into a project, the contents of the zip file will go into a directory such as `plugins/example-1.0/`. Plugin contents **will not** be copied into the main source tree. A plugin never interferes with a project's primary source tree.

## 12.4 Providing Basic Artefacts

**Adding a new Script**
A plugin can add a new script simply by providing the relevant Gant script within the scripts directory of the plugin:

```
+ MyPlugin.groovy
   + scripts     <-- additional scripts here
   + griffon-app
        + controllers
        + models
        + etc.
    + lib
```

**Adding a new Controller, Model, View or Service**
A plugin can add a new MVC Group, service or whatever by simply creating the relevant file within the `griffon-app` tree. However you'll need to create an Addon in order to package them properly.

```
+ ExamplePlugin.groovy
   + scripts
   + griffon-app
        + controllers  <-- additional controllers here
        + services <-- additional services here
        + etc.  <-- additional XXX here
    + lib
```

## 12.5 Hooking into Build Events

**Post-Install Configuration and Participating in Upgrades**
Griffon plugins can do post-install configuration and participate in application upgrade process (the upgrade command). This is achieved via two specially named scripts under `scripts` directory of the plugin - `_Install.groovy` and `_Upgrade.groovy`.
`_Install.groovy` is executed after the plugin has been installed and `_Upgrade.groovy` is executed each time the user upgrades his application with upgrade command.
These scripts are normal Gant scripts so you can use the full power of Gant. An addition to the standard Gant variables is the `pluginBasedir` variable which points at the plugin installation basedir.
As an example the below `_Install.groovy` script will create a new directory type under the `griffon-app` directory and install a configuration template:

```
ant.mkdir(dir:"${basedir}/griffon-app/jobs")
ant.copy(file:"${pluginBasedir}/src/samples/SamplePluginConfiguration.groovy",
         todir:"${basedir}/griffon-app/conf")
// To access Griffon home you can use following code:
// ant.property(environment:"env")
// griffonHome = ant.antProject.properties."env.GRIFFON_HOME"
```

**Scripting events**
It is also possible to hook into command line scripting events through plugins. These are events triggered during execution of Griffon target and plugin scripts.
For example, you can hook into status update output (i.e. "Tests passed", "Server running") and the creation of files or artifacts.
A plugin merely has to provide an `_Events.groovy` script to listen to the required events. Refer the documentation on Hooking into Events for further information.

## 12.6 Addons

**Understanding Addons**
Addons are a plugin's best friend. While plugins can only contribute build-time artifacts (such as scripts) and participate on build events, addons may contribute runtime artifacts (such as MVC Groups or services) and participate on application events.
Often times whenever you'd like to package a reusable runtime artifact you'd have to create an Addon as well.

**Addon responsibilities**
Addons may contribute any of the following to your application: MVC Groups and application event handlers. They can also contribute the following to the CompositeBuilder: factories, methods, properties and FactoryBuilderSupport delegates (attribute, preInstantiate, postInstantiate, postNodeCompletion).
Addons are created using a template that suggests all of the properties and methods you can use configure. The complete list follows:

- `addonInit` - called right after the addon has been loaded but before contributions are taken into account
- `addonPostInit` - called after all contributions haven been made
- `addonBuilderInit` - called before contributions to the CompositeBuilder are taken into account
- `addonBuilderPostInit` - called after all CompositeBuilder contributions haven been made
- `events` - Map of additional application event handlers
- `factories` - Map of additional node factories, added to CompositeBuilder
- `methods` - Map of additional methods, added to CompositeBuilder
- `props` - Map of additional methods, added to CompositeBuilder
- `attributeDelegates` - List of attributeDelegates (as Closures), added to CompositeBuilder
- `preInstantiateDelegates` - List of preInstantiateDelegates (as Closures), added to CompositeBuilder

- `postInstantiateDelegates` - List of postInstantiateDelegates (as Closures), added to CompositeBuilder
- `postNodeCompletionDelegates` - List of postNodeCompletionDelegates (as Closures), added to CompositeBuilder

**Configuring Addons**
This task is done automatically for you when you package an addon inside a plugin. The plugin's `_Install` and `_Uninstall` scripts will make sure that `griffon-app/conf/Builder.groovy` stays up to date. When you install a plugin that contains an addon you'll notice that `Builder.groovy` may get updated with a line similar to the next one

```
root.'CustomGriffonAddon'.addon=true
```

This means that all factories, methods and props defined on the Addon will be available to View scripts. However you need to explicitly specify which contributions should be made to other MVC members. You can list them one by one, or use a special group qualified by '*'. In recent releases of Griffon the default configuration is assumed meaning you won't see any changes in the `Builder.groovy` file. You can still apply modifications as explained below.
The following snippet shows how to configure an Addon to contribute all of its methods to Controllers, and all of its contributions to Models.

```
root.'CustomGriffonAddon'.controller='*:methods'
root.'CustomGriffonAddon'.model='*'
```

The special groups are: '*', '*:factories', '*:methods', '*:props'
Should you encounter a problem with duplicate node names you can change the default prefix (`root`) of the addon to something more suitable to your needs. All nodes contributed by the addon will now be accessible using that prefix. Here's an example

```
nx.'CustomGriffonAddon'.addon=true
```

Assuming `CustomGriffonAddon` is defined as follows

```
class CustomGriffonAddon {
    def factories = [
        button: com.acme.CustomButton
    ]
}
```

Then instances of `CustomButton` can be obtained by using `nxbutton`, whereas regular instances of `JButton` will be accessible with `button`.

## 12.7 Understanding Plugin Order

**Controlling Plugin Dependencies**
Plugins often depend on the presence of other plugins and can also adapt depending on the presence of others. To cover this, a plugin can define a `dependsOn` property. For example, take a look at this snippet from the Griffon Clojure plugin:

```
class ClojureGriffonPlugin {
   def version = 0.3
   def dependsOn = ["lang-bridge": "0.2.1"]
}
```

As the above example demonstrates the Clojure plugin is dependent on the presence of a single plugin: the `lang-bridge` plugin.
Essentially the dependencies will be loaded first and then the Clojure plugin. If all dependencies do not load, then the plugin will not load.
The `dependsOn` property also supports a mini expression language for specifying version ranges. A few examples of the syntax can be seen below:

```
def dependsOn = [foo:"* > 1.0"]
def dependsOn = [foo:"1.0 > 1.1"]
def dependsOn = [foo:"1.0 > *"]
```

When the wildcard * character is used it denotes "any" version. The expression syntax also excludes any suffixes such as -BETA, -ALPHA etc. so for example the expression "1.0 > 1.1" would match any of the following versions:

- 1.1
- 1.0
- 1.0.1
- 1.0.3-SNAPSHOT
- 1.1-BETA2

**Controlling Addon Load Order**
Addons will be loaded in the order determined by the dependencies set forth in their containing plugins. Using `dependsOn` establishes a "hard" dependency. Any addons provided by the dependencies will be added first to the builder configuration file when installed.

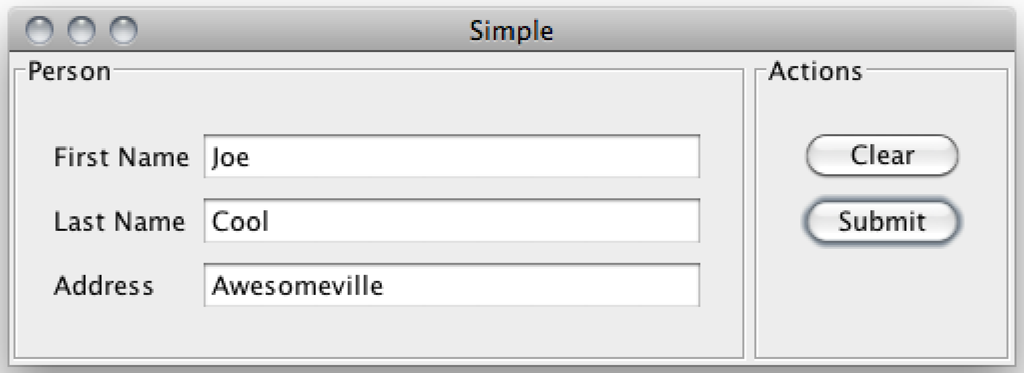
## 12.8 CLI Dependencies

Plugins can provide compile time classes that should not be bundled with runtime classes (i.e, addon sources). Sources and resources placed under `$basedir/src/cli` will be automatically compiled and packaged into a jar whose name matches `griffon-${plugin.name}-${plugin.version}-compile.jar`. A typical use case for these type of classes is a custom AST transformation that should be run during compile time but not at runtime.

# 13. Tips and Tricks

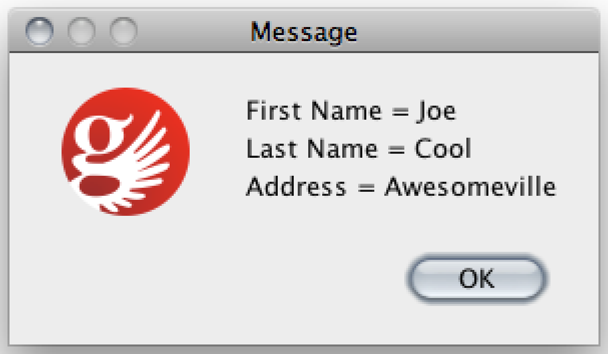
## 13.1 Using Artifact Conventions to your Advantage

The Artifact API can be a very powerful ally. Not only it's useful for adding new methods (explained in section 5.7.2 Adding Dynamic Methods at Runtime) but also comes in handy to finding out what application specific attributes an artifact has, for example Controller actions or Model properties. The following screenshot shows a simple application that presents a form based View.



First Name Joe
Last Name Cool
Address Awesomeville

When the user clicks the `Submit` button a dialog appears



First Name = Joe
Last Name = Cool
Address = Awesomeville

Believe it or not both the View and the Controller know nothing about the specific property names found in the Model. Let's have a look at the Model first

```
package simple
@Bindable
class SimpleModel {
    String firstName
    String lastName
    String address
}
```

There are 3 observable properties defined in the Model. Note the usage of default imports to avoid an extra line for importing groovy.beans.Bindable. Now, the Controller on the other hand defines two actions

```
package simple
import griffon.util.GriffonNameUtils as GNU
class SimpleController {
    def model
    def clear = {
        model.griffonClass.propertyNames.each { name ->
            model[name] = ''
        }
    }
    @Threading(Threading.Policy.SKIP)
    def submit = {
        javax.swing.JOptionPane.showMessageDialog(
            app.windowManager.windows.find{it.focused},
            model.griffonClass.propertyNames.collect([]) { name ->
                "${GNU.getNaturalName(name)} = ${model[name]}"
            }.join('n')
        )
    }
}
```

The `clear()` action is responsible for reseting the values of each Model property. It does so by iterating over the names of the properties found in the Model. The `submit()` action constructs a list fo model property names and their corresponding values, then presents it in a dialog. Notice that the Controller never refers to a Model property directly by its name, i.e, the Controller doesn't really know that the Model has a `firstName` property. Finally the View

```
package simple
import griffon.util.GriffonNameUtils as GNU
application(title: 'Simple',
  pack: true,
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.png').image,
  iconImages: [imageIcon('/griffon-icon-48x48.png').image,
               imageIcon('/griffon-icon-32x32.png').image,
               imageIcon('/griffon-icon-16x16.png').image]) {
    borderLayout()
    panel(constraints: CENTER,
          border: titledBorder(title: 'Person')) {
        migLayout()
        model.griffonClass.propertyNames.each { name ->
            label(GNU.getNaturalName(name), constraints: 'left')
            textField(columns: 20, constraints: 'growx, wrap',
                text: bind(name, target: model, mutual: true))
        }
    }
    panel(constraints: EAST,
          border: titledBorder(title: 'Actions')) {
        migLayout()
        controller.griffonClass.actionNames.each { name ->
            button(GNU.getNaturalName(name),
                actionPerformed: controller."$name",
                constraints: 'growx, wrap')
        }
    }
}
```

The View also iterates over the Model's property names in order to construct the form. It follows a similar approach to dynamically discover the actions that the Controller exposes.

You must install the MigLayout Plugin before running this application.

## 13.2 Dealing with Non-Groovy Artifacts

Since version 0.9.1 Griffon supports writing artifacts in JVM languages other than Groovy. The first of such languages is Java and it's supported in core by default. Additional languague support will be provided by plugins.

**Creating a Non-Groovy Artifact**

Many of the `create-*` scripts that come bundled with Griffon support an additional parameter that can be used to specify the language or filetype of the artifact. Non-Groovy artifacts must extend a particular class in order to receive all the benefits of a typical artifact. The default artifact templates can handle both Groovy and Java types. The following command will create an application that uses Java as the default language for the the initial MVC group

```
griffon create-app simple -fileType=java
```

The fileType switch indicates that the templates must pick a Java based template first. If no suitable template is found then a Groovy based template will be used. The setting of this flag is saved in the application's build configuration, this way you don't need to specific the fileType switch again if your intention is to create another artifact of the same type. Of course you can specify the flag at any time with a different value. It's worth mentioning that the default Groovy based template will be used if a suitable template for the specified fileType cannot be found. Peeking into each member of the `simple` MVC group we find the following code. First the Model

```
package simple;
import org.codehaus.griffon.runtime.core.AbstractGriffonModel;
public class SimpleModel extends AbstractGriffonModel {
    // an observable property
    // private String input;
    // public String getInput() {
    //     return input;
    // }
    // public void setInput(String input) {
    //     firePropertyChange("input", this.input, this.input = input);
    // }
}
```

Next is the Controller

```
package simple;
import java.awt.event.ActionEvent;
import org.codehaus.griffon.runtime.core.AbstractGriffonController;
public class SimpleController extends AbstractGriffonController {
    private SimpleModel model;
    public void setModel(SimpleModel model) {
        this.model = model;
    }
    /*
    public void action(ActionEvent e) {
    }
    */
}
```

And finally the View

```
package simple;
import java.awt.*;
import javax.swing.*;
import java.util.Map;
import griffon.swing.SwingGriffonApplication;
import griffon.swing.WindowManager;
import org.codehaus.griffon.runtime.core.AbstractGriffonView;
public class SimpleView extends AbstractGriffonView {
    private SimpleController controller;
    private SimpleModel model;
    public void setController(SimpleController controller) {
        this.controller = controller;
    }
    public void setModel(SimpleModel model) {
        this.model = model;
    }
    // build the UI
    private JComponent init() {
        JPanel panel = new JPanel(new BorderLayout());
        panel.add(new JLabel("Content Goes Here"), BorderLayout.CENTER);
        return panel;
    }
    @Override
    public void mvcGroupInit(Map<String, Object> args) {
        execInsideUISync(new Runnable() {
            public void run() {
                Container container = (Container) getApp().createApplicationContainer();
                if(container instanceof Window) {
                   containerPreInit((Window) container);
                }
                container.add(init());
                if(container instanceof Window) {
                   containerPostInit((Window) container);
                }
            }
        });
    }
    private void containerPreInit(Window window) {
        if(window instanceof Frame) ((Frame) window).setTitle("simple");
        window.setIconImage(getImage("/griffon-icon-48x48.png"));
        // uncomment the following lines if targeting +JDK6
        // window.setIconImages(java.util.Arrays.asList(
        //     getImage("/griffon-icon-48x48.png"),
        //     getImage("/griffon-icon-32x32.png"),
        //     getImage("/griffon-icon-16x16.png")
        // ));
        window.setLocationByPlatform(true);
        window.setPreferredSize(new Dimension(320, 240));
    }
    private void containerPostInit(Window window) {
        window.pack();
        ((SwingGriffonApplication) getApp()).getWindowManager().attach(window);
    }
    private Image getImage(String path) {
        return Toolkit.getDefaultToolkit().getImage(SimpleView.class.getResource(path));
    }
}
```

## 13.3 Externalizing Views

Groovy is the default language/format for writing Views, however there might be times where you would rather use a different format for describing a View. It might be the case that you have a legacy View (plain Java code) that you would like to plugin into Griffon. Here are a few tips to get the job done.

### 13.3.1 NetBeans Matisse

NetBeans comes with a visual designer named Matisse which is quite popular among a good number of developers. Matisse views are usually defined by a Java class. Most of the times all UI widgets are exposed as fields on the Java class. Based with this information Griffon can generate a View script that is backed by this particular Java class. Follow these steps to reuse a Matisse view.

#### #1 Place the Matisse View in your application

If you have access to the View's source code then please it somewhere in the application's source tree. A matching package to the traget MVC group in `src/main` is what is preferred. However, if the View is distributed in byte code form the make sure to place the jar that contains the View inside the application's `lib` directory. Alternatively you can use the Dependency DSL if the jar is available from a jar file repository (such as Maven or Ivy). Lastly, make sure that you have added the jar that contains `GroupLayout`, Matisse's work horse. this is easily accomplished by adding the following confuration in `griffon-app/conf/BuildConfig.groovy`

```
griffon.project.dependency.resolution = {
    repositories {
        // enable this option in an existing 'repositories' block
        mavenCentral()
    }
    dependencies {
        // add this to an existing 'dependencies' block
        compile 'org.swinglabs:swing-layout:1.0.3'
    }
}
```

**#2 Convert the View into a Script**

Griffon includes a script commmand target that can read a Matisse View and generate a Groovy View Script from it: `generate-view-script`. Execute the command by specifying the name of the Java class that defines the Matisse View, like this

```
griffon generate-view-script sample.LoginDialog
```

This command should generate the file `griffon-app/views/sample/LoginDialogView.groovy` with the following contents

```
// create instance of view object
widget(new LoginDialog(), id:'loginDialog')
noparent {
    // javax.swing.JTextField usernameField declared in LoginDialog
    bean(loginDialog.usernameField, id:'usernameField')
    // javax.swing.JPasswordField passwordField declared in LoginDialog
    bean(loginDialog.passwordField, id:'passwordField')
    // javax.swing.JButton okButton declared in LoginDialog
    bean(loginDialog.okButton, id:'okButton')
    // javax.swing.JButton cancelButton declared in LoginDialog
    bean(loginDialog.cancelButton, id:'cancelButton')
}
return loginDialog
```

**#3 Tweak the generated View**

From here on you can update the generated View as you see fit, for example by adding bindings to each field and actions to the buttons

```
widget(new LoginDialog(mainFrame, true), id:'loginDialog')
noparent {
    bean(loginDialog.usernameField, id:'usernameField',
         text: bind(target: model, 'username'))
    bean(loginDialog.passwordField, id:'passwordField',
         text: bind(target: model, 'password'))
    bean(loginDialog.okButton, id:'okButton',
         actionPerformed: controller.loginOk)
    bean(loginDialog.cancelButton, id:'cancelButton',
         actionPerformed: controller.loginCancel)
}
return loginDialog
```

### 13.3.2 Abeille Forms Designer

Another interesting choice is Abeille Forms, which is supported via a Builder and a plugin. Abeille Forms includes a visual designer that arranges the widgets with either JGoodies FormLayout or the JDK's GridBagLayout. Integrating these kind of views is a bit easier than the previous ones, as Abeille Forms views are usually distributed in either XML or a binary format. The plugin provides a View node that is capable of reading both formats. Follow these steps to setup a View of this type.

**#1 Install the Abeille Forms plugin**

As with any oher plugin, just call the `install-plugin` command with the name of the plugin

```
griffon install-plugin abeilleform-builder
```

**#2 Place the form definition in your source code**

If you have direct access to the files generated by Abeille's designer then place them somewhere under `griffon-app/resources`. Otherwise if the files are packaged in a jar, place the jar in your application's `lib` directory. Alternatively you can use the Dependency DSL if the jar is available from a jar file repository (such as Maven or Ivy).

**#3 Use the formPanel node**

As a final step you just need to use the `formPanel` node in a regular Groovy View script. All of the form's elements will be exposed to the Script, which means you can tweak their bindings and actions too, like this

```
dialog(owner: mainFrame,
  id: "loginDialog",
  resizable: false,
  pack: true,
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.png').image,
  iconImages: [imageIcon('/griffon-icon-48x48.png').image,
               imageIcon('/griffon-icon-32x32.png').image,
               imageIcon('/griffon-icon-16x16.png').image]) {
    formPanel('login.xml')
    noparent {
        bean(model, username: bind{ usernameField.text })
        bean(model, password: bind{ passwordField.text })
        bean(okButton, actionPerformed: controller.loginOk)
        bean(cancelButton, actionPerformed: controller.loginCancel)
    }
}
```

## 13.3.3 XML

Yet another option to externalize a View is a custom XML format that closely ressembles the code that you can find in a Groovy View script. Why XML you ask? Well because it is a ver popular format choice still, some developers prefer writing declarative programming with it. This option is recommended to be paired with Java views, just because if you're writing a Groovy View it makes more sense to use Groovy to write the whole instead. Follow these steps to get it done.

**#1 Change the Java View class**

A typical Java View class will extend from AbstractGriffonView. This super class defines a method named `buildViewFromXml()` that takes a Map as its sole argument. This map should contain all variables that the builder may require to wire the View, such as 'app', 'controller' and 'model' for example.

```
package sample;
import java.util.Map;
import org.codehaus.griffon.runtime.core.AbstractGriffonView;
public class SampleView extends AbstractGriffonView {
    private SampleController controller;
    private SampleModel model;
    public void setController(SampleController controller) {
        this.controller = controller;
    }
    public void setModel(SampleModel model) {
        this.model = model;
    }
    public void mvcGroupInit(Map<String, Object> args) {
        buildViewFromXml(args);
    }
}
```

**#2 Define the XML view**

The `buildViewFromXml()` method expects an XML file whose name matches the name of the class from where it's called, in this case it should be `SampleViw.xml`. Make sure to place the following contents in `griffon-app/resources/sample/SampleView.xml`

```
<application title="app.config.application.title"
             pack="true">
    <actions>
        <action id="'clickAction'"
                name="'Click'"
                closure="{controller.click(it)}"/>
    </actions>
    <gridLayout cols="1" rows="3"/>
    <textField id="'input'" columns="20"
        text="bind('value', target: model)"/>
    <textField id="'output'" columns="20"
        text="bind{model.value}" editable="false"/>
    <button action="clickAction"/>
</application>
```

Notice that every string literal value must be escaped with additional quotes otherwise the builder will have trouble setting the appropriate values. When the time comes for the builder to parse the view it will translate the XML to a Groovy scritpt then interpret it. This is the generated Groovy script that matches the previous XML definition

```
application(title: app.config.application.title, pack: true) {
  actions {
    action(id: 'clickAction', name: 'Click', closure: {controller.click(it)})
  }
  gridLayout(cols: 1, rows: 3)
  textField(id: 'input', text: bind('value', target: model), columns: 20)
  textField(id: 'output', text: bind{model.value}, columns: 20, editable: false)
  button(action: clickAction)
}
```

## 13.4 Creating Bindings in Java

Bindings are an effective way to keep two properties in sync. Unfortunately Java does not provide a mechanism nor an API to make bindings, but Griffon does.
As shown in section 6.2 Binding, Griffon relies on `PropertyChangeEvent` and `PropertyChangeListener` to keep track of property changes and notify observers. Swing components are already observable by default. You can build your own observable classes by following a convention, or implement the Observable interface (there's a handy partial implementation in AbstractObservable that you can subclass).
Bindings can be created by using BindUtils.binding(), like the following example shows

```
package sample;
import java.util.Map;
import groovy.util.FactoryBuilderSupport;
import griffon.swing.BindUtils;
import org.codehaus.griffon.runtime.core.AbstractGriffonView;
public class SampleView extends AbstractGriffonView {
    private SampleController controller;
    private SampleModel model;
    public void setController(SampleController controller) {
        this.controller = controller;
    }
    public void setModel(SampleModel model) {
        this.model = model;
    }
    public void mvcGroupInit(Map<String, Object> args) {
        buildViewFromXml(args);
        FactoryBuilderSupport builder = getBuilder();
        /*
         * Equivalent Groovy code
         * bind(source: input, sourceProperty: 'text',
         *      target: model, targetProperty: 'value')
         */
        BindUtils.binding()
                 .withSource(builder.getVariable("input"))
                 .withSourceProperty("text")
                 .withTarget(model))
                 .withTargetProperty("value")
                 .make(builder);
        /*
         * Equivalent Groovy code
         * bind(source: model, sourceProperty: 'value',
         *      target: input, targetProperty: 'text')
         */
        BindUtils.binding()
                 .withSource(model)
                 .withSourceProperty("value")
                 .withTarget(builder.getVariable("output"))
                 .withTargetProperty("text")
                 .make(builder);
    }
}
```

The following rules apply:

- both `source` and `target` values must be specified. An `IllegalArgumentException` will be thrown if that's not the case.
- both `source` and `target` instances must be observable. This does not imply that both must implement Observable per se, as Swing components do not.
- either `sourceProperty` or `targetProperty` can be omitted but not both. The missing value will be taken from the other property.
- the `builder` instance must be able to resolve the `bind()` node. This is typically the case for the default builder supplied to Views (because Swingbuilder is included).

Bindings created in this way also support the following attributes: `mutual`, `converter` and `validator`. The next snippet improves on the previous example by setting a converter and a validator, only numeric values will be accepted.

```
package sample;
import java.util.Map;
import groovy.util.FactoryBuilderSupport;
import griffon.swing.BindUtils;
import griffon.util.CallableWithArgs;
import org.codehaus.griffon.runtime.core.AbstractGriffonView;
public class SampleView extends AbstractGriffonView {
    private SampleController controller;
    private SampleModel model;
    public void setController(SampleController controller) {
        this.controller = controller;
    }
    public void setModel(SampleModel model) {
        this.model = model;
    }
    public void mvcGroupInit(Map<String, Object> args) {
        buildViewFromXml(args);
        FactoryBuilderSupport builder = getBuilder();
        /*
         * Equivalent Groovy code
         * bind(source: input, sourceProperty: 'text',
         *      target: model, targetProperty: 'value',
         *      converter: {v -> v? "FOO $v" : 'BAR'},
         *      validator: {v ->
         *          if(v == null) true
         *          try { Integer.parseInt(String.valueOf(v)); true }
         *          catch(NumberFormatException e) { false }
         *      })
         */
        BindUtils.binding()
                 .withSource(builder.getVariable("input"))
                 .withSourceProperty("text")
                 .withTarget(model)
                 .withTargetProperty("value")
                 .withConverter(new CallableWithArgs<String>() {
                      public String call(Object[] args) {
                          return args.length > 0 ? "FOO "+ args[0] : "BAR";
                      }
                  })
                 .withValidator(new CallableWithArgs<Boolean>() {
                      public Boolean call(Object[] args) {
                          if(args.length == 0) return Boolean.TRUE;
                          try {
                              Integer.parseInt(String.valueOf(args[0]));
                              return Boolean.TRUE;
                          } catch(NumberFormatException e) {
                              return Boolean.FALSE;
                          }
                      }
                  })
                 .make(builder);
        /*
         * Equivalent Groovy code
         * bind(source: model, sourceProperty: 'value',
         *      target: input, targetProperty: 'text')
         */
        BindUtils.binding()
                 .withSource(model)
                 .withSourceProperty("value")
                 .withTarget(builder.getVariable("output"))
                 .withTargetProperty("text")
                 .make(builder);
    }
}
```

The View for these examples is defined in XML format (as described in the previous section)

```
<application title="app.config.application.title"
             pack="true">
    <actions>
        <action id="'clickAction'"
                name="'Click'"
                closure="{controller.click(it)}"/>
    </actions>
    <gridLayout cols="1" rows="3"/>
    <textField id="'input'" columns="20"/>
    <textField id="'output'" columns="20" editable="false"/>
    <button action="clickAction"
</application>
```

However you can build the View in any way, bindings do not require an specific View construction mechanism in order to work.