CONTENTS INCLUDE:

Getting Started with Griffon

By Hamlet D'Arcy

WHAT IS GRIFFON?

Griffon is a Grails like application framework for rich desktop applications and is built on top of Groovy, Java, and Swing. Griffon embraces convention over configuration, automates many common development tasks, and features a large and growing plugin system. Griffon also features property binding for widgets and a broad and extensible event system. These combine to make Griffon an excellent choice for rich Internet applications. You can write maintainable and well designed applications quickly without spending time on builds, deployment, or configuration tweaking. You also get a rock-solid, secure, and well-known platform in the JVM, and along with the productivity boost of a modern, dynamic language while retaining and using all your Java expertise.

FROM 0 TO DEPLOYED

Griffon is a full life-cycle framework: it automates not just creation and maintenance of applications, but also build and deployment tasks. We'll cover Model-View-Controller next, but let's start with creating, running, packaging, and deploying an app with Java WebStart, going from nothing to deployed in about 5 minutes time. We could deploy as an applet, Jar, or desktop installation as well, but WebStart is the most interesting for RIAs.

Prerequisites – You should have the Java Development Kit (JDK) version 1.5+ (Version 6 recommended).

Installing Griffon – Download the latest release from http://griffon.codehaus.org/. Simply extract the .zip file somewhere on your machine. Next, create an environment variable called GRIFFON_HOME, pointing to the directory you unzipped the package, and add GRIFFON_HOME/bin to your path. The Griffon website contains instructions on how to set environment variables if you need more specific guidance. If you've done everything correctly then you should be able to open a command prompt, enter "griffon help", and see a help message from Griffon.

```
$ griffon help
Welcome to Griffon 0.3.1 - http://griffon.codehaus.org/
...
```

Griffon provides many commands to help you create and manage an application, and plugins may add more commands. "griffon help" will display all the available commands for your system.

griffon help	Displays all the available commands for the Griffon installation
griffon <target> help	Displays help for the specified target

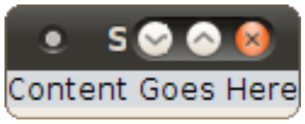
Creating an Application – A full application stack is only a command away. Simply run "griffon create-app" and enter the name for the app when you are prompted.

```
$ griffon create-app
...
Application name not specified. Please enter:
Starter
...
Created Griffon Application at /home/hdarcy/dev/Starter
$ cd Starter
```

This step creates an entire project on your disk: standard directory layout, MVC groups, internationalization bundles, build scripts, IDE files, and more. The application is little more than a Hello World app at this point, but you can run it.

griffon create-app	Creates a new Griffon application

Running the Application – Griffon apps are desktop applications that a user can install, Applet applications that run in a browser, and WebStart applications that run as Internet apps. Griffon hides the platform differences from you so any Griffon app you write can be deployed in any of these forms. To run the application on your desktop enter "griffon run-app", and your Hello World style window will pop up shortly after the compile and packaging automatically finishes.



It's not much to look at yet, but pretty good considering we have written no code. You can also run the application using "griffon run-applet" or "griffon run-webstart".

griffon run-app	Compiles and runs the application as a desktop app
griffon run-applet	Compiles and runs the application a browser Applet
griffon run-webstart	Compiles and runs the application as a JNLP Webstart project

Signing the Application – The Java platform offers excellent security options, but the in past configuring these options was complex. This is an optional step, you do not need to sign applications, but you should, and Griffon handles all the hard work of managing digital signatures and signing Jar files for you with just two easy steps. First create your digital signature using the JDK's keytool program (unless your company already has done this). We'll call ours MyKey.

```
$ keytool -genkey -alias MyKey
```

You'll be prompted to enter a password, as well as your name, company, and location. This creates the required Java key files in your home directory. Now we just need to tell our application about this key file. In a text editor, open the `./griffon-app/conf/Config.groovy` configuration file. There are 5 entries that need to be changed in order to properly sign a WebStart application. In the `environments.production.signingkey.params` properties, add the following settings modified with your home directory and key name:

```
sigfile = 'MyKey'
keystore = "/home/hdarcy/.keystore"
alias = 'MyKey'
```

Then, under `environments.production.griffon` turn Jar packing off and specify your Internet deployment location.

```
jars {
    pack = false
}
webstart {
    codebase = 'http://www.canoo.com/griffon/starter'
}
```

Finished. The Jars will all be signed the next time you package the app.

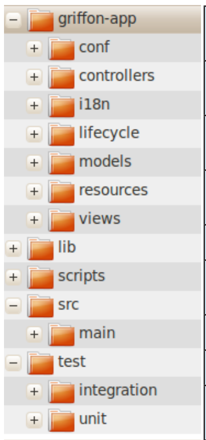
Hot Tip Groovy property files are an improved version of Java property files. The old .properties file syntax of "key=value" can still be used if you want. However, properties are hierarchical and grouped, and the Groovy syntax makes these groupings more apparent. Plus, you can put any code you want in a property file and have it execute.

Packaging and Deployment – To package your application, run the "griffon package" target. This creates deployable files for Jar, Applet, and WebStart deployments. You'll be prompted for your keystore password during the signature process. From here, just copy all the contents of your `./dist/webstart` folder onto any webserver and your application is ready to launch over the Internet. You can see this sample deployed at http://canoo.com/griffon/starter. There is no need for a Java Servlet container, just copy the files and access the URL.

griffon package	Packages the application into deployable bundles

SWING DONE RIGHT

Consistent Project Structure - Griffon projects follow a convention; all Griffon projects are meant to look the same. This consistency eases maintenance costs and promotes application best practices. The following table shows the project layout for a default project, and lists the contents of each folder.



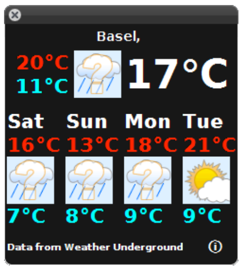
./griffon-app/conf	Build and runtime configuration data
./griffon-app/controllers	Controller classes, orchestrate views and models
./griffon-app/i18n	Message bundles for internationalization
./griffon-app/lifecycle	Scripts to run on application events
./griffon-app/models	Model classes
./griffon-app/resources	Images, properties, and other resources
./griffon-app/views	User interface view classes
./lib	Jar files and libraries
./scripts	Gant scripts, a Groovy wrapper around the Ant build tool
./src/main	Other source files, with many JVM languages supported
./test	Testing files, at both the integration and unit level

Hot Tip When configuring version control for your project, do not check in these files: dist, staging, stacktrace.log, and the .iws file. Instead, add them to your VC's ignore list.

Pervasive MVC – Model-View-Controller (MVC) is an often used design pattern that separates concerns in applications. An MVC pattern contains three essential elements: A **View** defines how your application looks. Buttons, Frames, and Widgets are all part of the view layer. A **Controller** defines how your application behaves. Querying the database, managing data, and coordinating user events are all part of the controller layer. The **Model** holds data and state required by both the controller and view. The state of buttons, the contents of text boxes, and dirty field tracking are all part of the model layer.

Hot Tip A Griffon model is not a domain model, but an application model. As such, the Griffon model makes it easy for the controller and view to exchange data in a toolkit agnostic way. A domain model describes the conceptual entities in your software system. Consider the difference between an Employee object (a domain model) and an EmployeeTableModel (an application model).

Our starter application contains exactly one MVC triad: the main frame. Larger Griffon applications are made of many MVC triads, which are all defined in `./griffon-app/conf/Application.groovy`. An MVC triad can be based around an entire window, a panel, or simply a widget. Beginners often create one MVC triad for each window in their application, but this is a mistake. MVC is a pattern you apply to components or bundles of components. The WeatherWidget sample application from the Griffon distribution illustrates this:

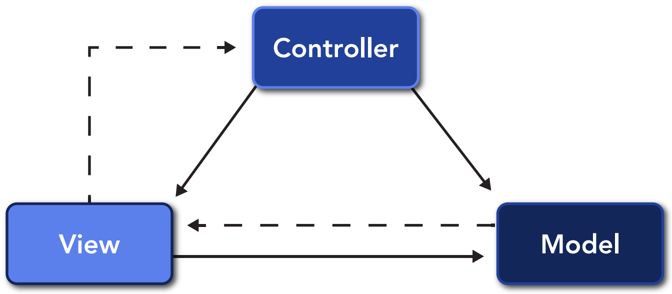


In this window there are two visible MVC triads: The main frame WeatherWidget, which contains four SmallForecast MVC triads (one for each day of the week shown). Creating and reusing MVC triads is a key design and decomposition technique for building Griffon applications. Strive to build many, small, and reusable MVC triads within your application. This Griffon command will help you:

griffon create-mvc	Creates a new MVC group: model, view, controller, and test

SIMPLE MVC

Griffon supports many view layer GUI toolkits through plugins: Swing (the default), Eclipse Standard Widget Toolkit (SWT), Pivot, Gtk, and even JavaFX. Using these toolkits is as easy as installing the appropriate plugin. For now, we'll stay in Swing and start with an example. Following is a modified MVC Group from the first example, we're adding a button and a simple counter to the screen. The view script adds a button and a label to the form, and the Controller simply updates a counter. The Model is used to communicate between the two:



ClickerView.groovy

```
application(title:'Clicker') {
    gridLayout(rows: 2, cols: 1)
    label(text:bind {model.message})
    button(text: 'Click Me', actionPerformed: controller.&action)
}
```

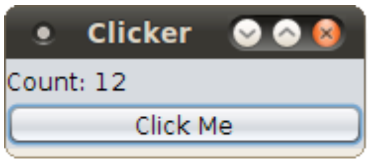
ClickerController.groovy

```
class ClickerController {
    def model
    def counter = 0
    def action = {
        model.message = "Count: ${counter++}"
    }
}
```

ClickerModel.groovy

```
class ClickerModel {
   @groovy.beans.Bindable String message
}
```

The launched application looks like this once we click the button a few times:



View - The view script is Groovy and relies heavily on SwingBuilder, a Groovy convenience class for building Swing based UIs. As you can see, SwingBuilder is a terse and DSL-like script for writing widgets and configuration, which maps closely to the Swing API. In case you are not too familiar with Groovy, here is a quick summary of how this View script relates back to the JDK:

gridLayout(rows: 2, cols: 1)	Applies the Java GridLayout to the current container. Parameters are named explicitly and map to the Java constructor or setter methods.
label(text:bind {model.message})	label() is a dynamic method and creates a JLabel in the current container. The text of the label is bound to a model property called message. Any updates to the model.message field will result in the label being updated. All of the event notification and listeners are handled for you.
button(text:'Click Me', ...)	button() is a dynamic method and creates a JButton in the current container. Setters may be invoked as named argument parameters.
actionPerformed: controller&action	Creates and adds a Swing ActionListener to the component. This is the Groovy alternative to an anonymous inner class. When the button is clicked the controller is invoked.

Hot Tip Laying out components in Swing forms is notoriously difficult using the GridBagLayout, which is exactly why MigLayout was created. It's goal is to make complex layouts easy and normal layouts one-liners, and may of the Griffon team swears by it. Just install the miglayout plugin to get started.

Groovy adds a simplifying layer over the standard GUI toolkits, like Swing; however, GUI toolkits are usually large and complex. It is worthwhile to invest in one of the many Swing books available. Groovy in Action contains a thorough treatment of SwingBuilder, and the Oracle online documentation for Swing is excellent.

Controller – The controller simply updates a counter and tells the model that an update occurred. The model field is automatically injected into controllers by Griffon (as is the view, if you desire). The action field is just a closure (think Runnable) that writes a new message back to the model.

Model – The communication hub holds a simple String field marked @Bindable. This lets Griffon know that objects may bind to the property, and any time the property is updated then the correct PropertyChangeListeners will fire and observers will be notified. Those with a Swing background will understand that this simple annotation removes about 50 lines of boilerplate Swing code!

Advanced Data Binding – Binding widgets to a model is a requirement for modern GUI toolkits. But if you've worked with other tools you may be worried about too many events being posted and your application slowing down under a load of bound requests, a situation known as a "Bind Storm". Griffon offers a flexible API to shelter you from this condition:

bind()	Adds automatic event listeners and ties a component to a model
unbind()	Removes the automatic listeners from a component
rebind()	Reinstates the automatic listeners to a component

You don't need to understand how binding works to write an application fast, but you may need to in order to write a fast application.

Hot Tip Many Look & Feels exist to skin Swing apps into something more eye-pleasing. One of the best collections is Substance, available from http://substance.dev.java.net/. To install the Substance Look & Feels, run the following command: *griffon install-plugin lookandfeel-substance*

SERVICES

Many fields are automatically injected into your components by Griffon: Views receive an application instance, a model, and a controller, and Controllers receive a model and view. You can also create your own helper objects that will automatically be injected into your controllers by using Griffon services. A service is an object with a no-arg constructor, and they are discovered and injected based on naming conventions. To create a service use this Griffon command:

griffon create-service	Creates a service class, prompting you to enter a package and a name. Also creates a unit test.

Here is our previous Clicker example refactored to use a service:

CounterService.groovy

```
class CounterService {
    def counter = 0
    def getNext() {
        counter++
    }
}
```

ClickerController.groovy

```
class ClickerController {

    def model
    def counterService

    def action = {
           model.message = “Count: $counterService.next”
    }
}
```

Services are a key tool in decomposing large applications into manageable, independent pieces. Strive to move logic out of controllers and into reusable services. If you need more control over construction, then use the Spring or Guice plugins to provide full dependency injection frameworks.

Hot Tip Mac and Linux users can easily chain commands together using the && operator. For example, to run a clean and package, run the command: “griffon clean && griffon package”. The 2nd command only runs if the 1st succeeds. This also works under Windows with Cygwin.

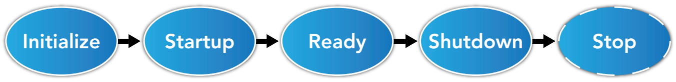
EVENTS

Custom Event Handing - Griffon has a rich, lightweight event system. For the most part, components in your application communicate via events and message passing rather than direct method calls. This means your controllers stay decoupled from one another but still communicate. Custom events and event handlers are wired together based on naming convention. Here is our controller that posts an event by interacting with the implicit “app” object and handles the event itself.

```
class ClickerController {

    def model
    def counterService

    def action = {
        app.event(“Click”, [counterService.next])
    }

    def onClick = { value ->
        doLater { model.message = “Count: $value” }
    }
}
```

Raise events using the GriffonApplication object named “app” and handle them by declaring an “on<EventName>” closure. Simple.

Application Life-Cycle Events - All Griffon applications have the same life-cycle, regardless of whether it is deployed as an applet, application, or webstart. As an application moves through the phases, you have the opportunity to cleanly execute any sort of acquiring or releasing resources you wish. Griffon provides an event system for you to both post and handle the events with custom code. Here’s the basics of the life-cycle:



Initialize	Application is created and configuration read. Good place to apply a new Look and Feel
Startup	All MVC Groups from ./griffon-app/conf/Application.groovy marked as startup groups are instantiated
Ready	All pending UI events have been processed and the main frame is about to be displayed
Shutdown	The application is about to close
Stop	Only available in Applet mode, called when destroy() is invoked by the container.

To respond and handle any of the lifecycle events, such as BootstrapEnd, LoadPluginStart, NewInstance, or any of the many others, then add a handler in the file `./griffon-app/conf/Events.groovy`. The name of the method should be “on<EventName>”. All of the available events are documented in the Griffon User Guide, and you may add your own custom events as well.

Extensible Build Scripting – The build of Griffon is completely scriptable; there is even a Griffon command to help you write build event extensions:

griffon create-script	Creates a build script in the ./script directory, prompting you to enter a script name

The Griffon build is built on the Gant framework, a Groovy extension to Ant. The content of a build script is an event handler. There are build events for CompileStart, PackagingStart, RunAppEnd, and many more. To add pre or post processing to a build event then simply declare a closure in the script named “event<EventName>”. You can even add new build events if you need to, and the Griffon User Guide contains a complete reference of the built-in build events.

THREADING

The last example contained a method call to something called “doLater” with a closure as a parameter. This hints at one of the major concerns for the Griffon or Swing developer: threading. Swing is a single-threaded GUI toolkit, meaning there is a single thread called the Event Dispatch Thread (EDT) dedicated to refreshing and repainting the UI. If you perform a long computation on the EDT then your application will not repaint properly or may become sluggish. If you update the state of a UI widget from a thread other than the EDT then your widget may not paint or be updated correctly. As a Swing programmer you must remember two rules: all interactions with widgets must occur on the EDT and any other processing should occur off the EDT, and this rule holds true for other GUI toolkits as well. Groovy’s SwingBuilder object gives you three convenience methods to help you with this, and Griffon automatically imports these methods into your MVC groups.

doOutside { ... }	Executes a block of code off the EDT.
edt { ... }	Executes a block of code on the EDT. Similar to the JDK's SwingUtilities.invokeAndWait.
doLater { ... }	Executes a block of code on the EDT. Similar to the JDK's SwingUtilities.invokeLater.

It is common for controller actions to start by reading a widget on the EDT, perform work off the EDT, and finally update the UI on the EDT. Here is a properly threaded version of our ClickerController.

```
class ClickerController {

    def model
    def counterService

    def action = {
        model.busy = true
        doOutside {
            try {
           model.message = "Count: $counterService.next"
            } finally {
                edt { model.busy = false }
            }
        }
    }
}
```

The SwingBuilder methods are Swing specific, but Griffon offers a platform agnostic way to invoke them as well. If you are targeting SWT or some other toolkit then use the execOutside, execSync, and execAsync methods instead.

Hot Tip It is worthwile to invest time understanding how Swing threading works by using the plenty of high-quality documentation that exists on the Internet. Oracle's Java Tutorial contains a section called "Concurrency in Swing" and other articles appear on the Sun Developer Network.

USING PLUGINS

The Griffon Plugin system is a key part of the framework. There are currently over 100 plugins listed in the public plugin repository, ranging from persistence providers like CouchDB and GSQL, rich components like Coverflow and GlazedLists, testing support like easyb and Spock, language support like Clojure and Scala, and many, many more. Plugins strive to make third party library integration a one or two line of code affair, and they are a great way to add features with a minimum of effort. As an example, consider how simple it is to generate Mac, Windows, and Linux installers with the Installer Plugin:

```
$ griffon install-plugin installer
$ griffon prepare-all-launchers
$ griffon create-all-launchers
```

Installing the plugin downloads the package from the public repository and installs it. The plugin adds several Griffon targets, and invoking these targets builds the packages. After running these commands you can see a Mac .app application, and Windows .exe installer, and several other formats all sitting in the ./installer folder.

Working effectively with plugins only requires mastering four Griffon targets:

griffon list-plugins	Lists all the plugins available to install
griffon plugin-info <plugin name>	Lists the documentation of the specified plugin
griffon install-plugin <plugin name>	Downloads and installs the specified plugin. Also accepts a file or URL as an argument.
griffon uninstall-plugin <plugin name>	Uninstalls the specified plugin.

Hot Tip If you'd like to publish your own plugin to the repository then send an email to the Griffon mailing list: you'll quickly be given the version control rights to execute a "griffon release-plugin". Also, the next version of Griffon allows you to create your own plugin repository so your organization can have a private repo for non-open source plugins.

TESTING

Testing is a first class concern in Griffon. Many of the built in targets create unit test stubs for you and the quality related plugins are particularly rich. The core targets for testing are:

griffon test-app griffon test-app -unit	Executes the tests in the project. When the -unit option is present, runs only the unit tests.
griffon create-unit-test	Creates a new unit test.
griffon create-integration-test	Creates a new integration test, in which the GriffonApplication object is available.

For additional testing options, install the easyb, FEST, and Spock plugins. Easyb is a Jolt award winning Behavior Driven Development library for Groovy, FEST is a UI testing library for Swing, and Spock is a rapidly growing testing tool in the Groovy community. There are many other code quality related plugins as well. The Clover and Cobertura plugins make code coverage statistics available, FindBugs and CodeNarc provide static code analysis, and JDepend and GMetrics offer structural and dependency analysis. Keeping the code clean is only a plugin install away.

Hot Tip Command completion using the Tab key is available for Bash based command shells, like Linux, Mac, or Windows' Cygwin. To enable completion, run the command "source $GRIFFON_HOME/bash/griffon-auto-scripts". You might just add this to execute as part of your login scripts.

IDE SUPPORT

Groovy enjoys very good IDE support; IntelliJ IDEA, NetBeans, and Eclipse all offer some level of Groovy support, and Griffon makes tooling easy by generating both Eclipse and IntelliJ IDEA projects files for every application. Currently, IntelliJ IDEA offers the best Groovy and Griffon support. Refactoring, Java-aware find usages, code completion, and many code intentions are supported. As for Griffon specific features, IDEA provides a project builder for new projects, a customized view that knows about MVC layouts, a Griffon target window to replace the need for the command line, support for unit and integration tests, and a UI to manage Griffon Plugins. Plus, you can generate an IDEA project from your Griffon sources in case you generated the project from the command line.

ON THE ROAD TO GRIFFON 1.0

The next version of Griffon will be 0.9, an API stable release on the road to 1.0. However, Griffon handles upgrades automatically and upgrades have always been painless. If you install version 0.3 and later upgrade to 0.9, then Griffon prompts you to run the upgrade scripts and convert the project. Upgrading should be seamless, any required changes are handled by the scripts supplied by the Griffon team or the plugin authors. If you've written your own plugins, take a look at the user mailing list to see if there are any required upgrade steps you need to provide to your users for a new release.

Hot Tip In version 0.9, Griffon supports camelCase for targets. This means typing "griffon cApp" matches "create-app" and executes that target. If the camelCase input is ambiguous then Griffon will prompt you to select the intended target from a list.

MORE INFO

The best documentation for Griffon is the Griffon in Action book, currently available in early access form from the publisher's website, if it hasn't already been published. It extensively covers the material here, but also takes in depth looks at writing your own plugins, installing new look and feels, and managing the development versus production environments. Otherwise, the Griffon User Guide and mailing list are free to use, and the mailing list in particular is quite responsive. For more info on rich clients in Swing, I recommend the book Filthy Rich Swing Clients.

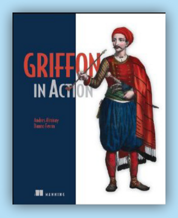
For learning by example, the Griffon download package contains several examples, and the code for all examples in this Refcard are available at: http://github.com/HamletDRC/GriffonRefcard. Lastly, you can follow all the latest Griffon news by following @theaviary on Twitter.
Happy Developing!

ABOUT THE AUTHOR

Hamlet D'Arcy

Hamlet D'Arcy has been writing software for over a decade, and has spent considerable time coding in Groovy, Java, and C++. He's passionate about learning new languages and different ways to think about problems, and recently he's been discovering the joys of both F# and Scheme. He's a committer on several open source projects including Groovy and JConch, and is a contributor on a few others (including Griffon and the IDEA Groovy Plugin). He blogs regularly at http://hamletdarcy.blogspot.com, tweets as HamletDRC, and can be contacted at hamletdrc@gmail.com.

RECOMMENDED BOOKS



Griffon in Action is a comprehensive tutorial written for Java developers who want a more productive approach to UI development. In this book, readers will immediately dive into Griffon. After a Griffon orientation and a quick Groovy tutorial, they'll start building examples that explore Griffon's high productivity approach to Swing development. The book covers declarative view development, like the one provided by JavaFX Script, as well as the structure, architecture and life cycle of Java application development.

BUY NOW
books.dzone.com/books/griffon-in-action

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-75-2
ISBN-10: 1-934238-75-9
50795
9 781934 238752
$7.95

Version 1.0