GRIFFON IN ACTION

Andres Almiray
Danno Ferrin
James Shingler

FOREWORD BY Dierk König

MANNING

Griffon in Action

Griffon in Action

ANDRES ALMIRAY
DANNO FERRIN
JAMES SHINGLER

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Cynthia Kane
Technical proofreader: Al Scherer
Copyeditors: Tiffany Taylor, Andy Carroll
Proofreader: Melody Dolab
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781935182238
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 18 17 16 15 14 13 12

brief contents

contents

foreword

As soon as I heard about *Griffon in Action*, I was eager to get it into my hands. What I expected was a typical Manning *In Action* book: providing an easy jump start, working from actionable examples, and providing lots of insight about the technology at hand. It turned out that this book not only lived up to my expectations, it exceeded them in many ways.

First, the authors' knowledge is indisputable. This is obvious for the technology, because we're talking about main Griffon contributors. But beyond that comes experience about all aspects of developing desktop applications based on Swing, ranging from how to set up your project, through proper separation of concerns, threading, building, testing, visual composition, and code metrics, down to how to deliver the final application to the customer.

Second, the book goes beyond giving simple recipes. It explains the underlying constraints and considerations that enable readers to make informed decisions about their projects.

Third, *Griffon in Action* is a great reference. I have it open whenever I write Griffon applications so I can quickly look up an example or a list of available goodies. It is such a thorough source of information that I consider it the definitive guide.

Writing such a book is a huge effort—especially when aiming for approachability and completeness at the same time. Additionally, the authors pushed the Griffon project forward while writing this book, and one or the other may even have an additional day job.

A big "thank you" to the authors of this book; and to you, readers, a warm-hearted "Keep groovin'."

DIERK KÖNIG
Author of *Groovy in Action*
First and Second Editions

preface

The book you're holding in your hands went through a lot of iterations before it reached its final form. We're not referring to the editorial process, but rather to the deep relationship it has with the topic it discusses: the Griffon framework. Both evolved at the same time almost from the beginning.

On a peaceful October afternoon back in 2007, Danno Ferrin, James Williams, and I (all members of the Groovy development team) had a very productive chat over Skype about the future of Groovy's SwingBuilder—an enabler for writing desktop GUIs using Swing as a DSL. We recognized the potential of mixing and matching different builders to write richer UIs, but the current syntax wasn't pleasant to use. We drafted a plan and got to work on our respective areas.

Fast-forward to JavaOne 2008, where the three of us got to meet face to face for the first time. Joined by Guillaume Laforge, we hatched the idea of what was to become the Griffon framework. We knew that Grails was making waves in the web space, and we felt the need for a similar outcome in the desktop space. Cue the light-bulb moment: we agreed that creating a desktop framework that stuck as closely as possible to Grails would be the way to go—although we didn't have a name for it yet.

Danno went back to his batcave after the conference and in a matter of weeks bootstrapped the framework by forking Grails and removing all the webby stuff that was not needed. Then he grafted in the most important pieces of Griffon's architecture: the UberBuilder, the MVC group conventions, and the application life cycle.

We finally had something tangible. James picked the name and we went public with the project on September 2008. The initial reaction from the community was so

positive than in a matter of months work on the book began. And this is where both projects got intertwined.

Together with Danno and Geertjan Wielenga, we wrote the first part of the book. We went to work on the framework, and then we came back to the book when we stopped to rethink where we were going with the framework. This kept going for months: hacking some code, writing a few pages. In the meantime, we received plenty of feedback about both projects. A particular advantage of this setup was that we were able to address the needs of users and readers and thus save time, the most precious resource for an open source effort.

Eventually Geertjan and Danno reduced their contributions, and my coauthor and good friend Jim Shingler joined the project. Being an early adopter of the technology plus a seasoned Swing developer meant he was the right person for the job. And he didn't disappoint. Thank you, Jim!

All this leads to where we are now, with you reading these pages. During the time it took to get the book into your hands, we painstakingly revised its goals and the framework, making sure both were kept as accurate and fresh as possible. Despite what the naysayers have said for years—that Java on the desktop is no longer relevant—the current situation couldn't be further from the truth. Griffon has been used to write applications that manage patient data, process the data required to manage the railroad schedule of an entire country, and even talk to a satellite in space!

It's our hope that you'll find the book to be the best resource for starting to work with this technology. Keep it close as a reference when you're in doubt about how to use a particular feature.

Enjoy!

ANDRES ALMIRAY

acknowledgments

Griffon in Action is the culmination of the efforts of a lot of people, without whom we would not have been able to accomplish its publication. We would like to begin by thanking Josh A. Reed for pitching the book during an autumn conversation as well as Christina Rudloff at Manning for getting the ball rolling. We need to express our appreciation to our development editors, Tara McGoldrick Walsh, Lianna Wlasiuk, and Cynthia Kane. Associate publisher Michael Stephens organized the project and got us on track to get the book finished in a timely and organized manner. Thanks to our editorial director Maureen Spencer, and to our copy editors, Tiffany Taylor and Andy Carroll, for making our writing readable. And thanks to the rest of the Manning staff, including Melody Dolab, Karen Tegtmeyer, Steven Hong, and Candace Gillhoolley.

It's important that a technical book be accurate, so we would like to thank our formal technical reviewers, Dean Iverson, Dierk König, and Al Scherer. We also thank those who read the book and provided feedback during various stages of the book's development: Geertjan Wielenga, Venkat Subramanian, Ken Kousen, Scott Davis, Michael Kimsal, Peter Niederwiser, Alex Ruiz, Guillaume Laforge, Dierk König, Hamlet D'Arcy, Gerrit Grünwald, Carl Dea, Dave Klein, Santosh D. Shanbhag, Edward Gibbs, Bob Brown, Doug Warren, Shawn Hartsock, Jean-Francois Poilpret, Amos Bannister, Gordon Dickens, Glen Smith, Jonas Bandi, Mykel Alvis, Eitan Suez, Sven Haiges, Jonathan Giles, Robby O'Connor, Josh Reed, and James Williams. We also thank Dierk for contributing the foreword to our book.

Thanks to all those who have contributed to the Groovy, Griffon, and Grails projects, especially Guillaume Laforge, Graeme Rocher, Jochen Theodoru, Alex Tkachman, Paul

King, Hans Dockter, Peter Niederwiser, Luke Daley, Spring Source, and VMWare. We would also like to thank other Groovy, Griffon, and Grails community contributors, including James Williams for SwingXBuilder, Alexander Klein for bringing new ideas to the framework, and René Gröschke and his build-bending Gradle powers. They have created some great stuff and should be proud of themselves. Thanks to Sven Haiges, Glen Smith, and Peter Ledbrook for their informative Grails podcast, where Griffon was present on several occasions. Other special mentions go to Peter for the countless exchanges we had regarding Grails and Griffon; Dick "I loooove the Groovy" Wall, Tor Norbye, Carl Quinn, and Joe Nuxoll for the Java Posse podcast; and Michael Kimsal for Groovy Mag.

ANDRES ALMIRAY

First and foremost, I would like to thank my wife, Ix-chel, for being my rock, anchor, companion, and soul mate. You wouldn't be holding this book in your hands without her patience, understanding, and driving force. I'd like to thank my parents for bringing me into this world and for all their love through the years. Patricia and Astrud: where would I be without all your help? A very special and warm thank you to Christianne, Joseph, and Didier Muelemans, dear mentors and beacons of hope. We had a group of professors back in college who shaped our professional lives and led us to where we are. Bruno Guardia, Enrique Espinoza, Carlos Guerra, Angel Kuri, and Barbaro Ferro, I'm grateful for all your lessons and your words of encouragement.

Danno Ferrin is the man with the plan. He wrote the initial pieces that eventually led us to bring forth the Griffon framework. You rock!

Geertjan Wielenga started the book with us; sadly, he had to let it go after a while. Still, his contributions in the early stages are deeply engrained in the book. Thank you for keeping the light of desktop Java shining bright (and the NetBeans Griffon plugin too!).

Thank you to the members of the Groovy community at large: Guillaume Laforge, Graeme Rocher, Jochen Theodoru, Alex Tkachman, Paul King, Hans Dockter, Peter Niederwiser, Luke Daley, Adam Murdoch, Dierk König, Hamlet D'Arcy, Roshan Dawrani, Cédric Champeau, Stéphane Maldini, Dave Klein, Zachary Klein, Ben Klein, Michael Kimsal, Jim Shingler, Chris Judd, Joseph Nusairat, Ken Kousen, Ken Sipe, Andrew Glover, Venkat Subramanian, Scott Davis, Tim Berglund, Matthew McCullough, Erik Wendelin, Burth Beckwith, Jeff Brown, Peter Ledbrook, Glen Smith, Sven Haiges, Tim Yates, Marc Palmer, Robert Fletcher, Tomas Lin, Andre Steingress, Andrew Eisenberg, Andy Clement, Peter Gromov, Colin Harrington, Shawn Hartsock, Søren Berg Glasius, Hubbert Klein Ikkink, Sébastien Blanc, Vaclav Pech, Russel Winder, Bernardo Gomez Palacios, Domingo Suarez, Jose Juan Reyes, and Alberto Vilches.

Java on the desktop has evolved a lot since the platform's inception back in 1995. The following people have carried it on their shoulders and sent it forward: Amy Fowler, Richard Bair, Jasper Potts, Joshua Marinacci, Hans Muller, Chet Haase, Scott Violet, Chris Campbell, Shannon Hickey, Romain Guy, Kirill Grouchnikov, Mikael

Grev, Jean-Francois Poilpret, Karsten Lentzsch, Gerrit Grünwald, Jim Weaver, Stephen Chin, Dean Iverson, Jim Clarke, Jonathan Giles, Carl Dea, Jeanette Winzenburg, and Rémy Rakic.

Thanks to the friends and colleagues I've met across the years: el equipazo! (Artemio Urbina, Jose Luis Balderas, Pedro Iniestra, and Francisco Macias), Ignacio Molina, Agustin Ramos, Kevin Nilson, Mike van Riper, Alex Ruiz, Yvonne Price, Stoyan Vassilev, Jay Zimmerman, Ben Ellison, Deepak Alur, Etienne Studder, Johannes Bühler, Sven Herke, Alberto Mijares, Detlef Brendle, Sibylle Peter, Dieter Holz, and Hans-Dirk Walter.

Last but not least, thanks to Mac Liaw, the evil genius behind it all.

JIM SHINGLER

I would like to thank my wife, Wendy, and son, Tyler, for their support and patience during the writing of the book and in our journey together through life. I would like to thank all those who have contributed to my personal and professional growth over the years: Wendy Shingler, Tyler Shingler, James L. Shingler Sr., Linda Shingler, George Ramsayer, Chris Judd, Andres Almiray, Danno Ferrin, Tom Posival, Ken Heintz, Bryce Kerlin, Rick Burchfield, David Lucas, Chris Nicholas, Tim Resch, BJ Allmon, Kevin Smith, Jeff Brown, Dave Klein, Paul King, Soren Berg Glasius, Michael Kimsal, Joseph Nusairat, Brian Sam-Bodden. Steve Swing, Brian Campbell, Greg Wilmer, Rick Fannin, Kunal Bajaj, Mukund Chandrasekar, Seth Flory, Frank Neugebauer, David Duhl, Jason Gilmore, Teresa Whitt, Jay Johnson, Gerry Wright, and the many other people who have touched my life. I'd also like to thank Jay Zimmerman, Andrew Glover, Dave Thomas, Venkat Subramaniam, Scott Davis, Neal Ford, Ted Neward, and the other great speakers and influencers on the "No Fluff Just Stuff" tour.

DANNO FERRIN

I would like to thank K.D., S.R., C.B, J.C., H.G., and H.F. for their support and patience.

about this book

Griffon in Action is a comprehensive introduction to the Griffon framework that covers the basic building blocks such as MVC groups, binding, threading, services, plugins, and addons. But don't let this quick summary fool you into thinking the topics are covered lightly. The book provides deep dives into the topics at hand, following a practical approach to get you started as quickly as possible.

Who should read this book

This book is for anyone interested in writing desktop applications for the Java virtual machine (JVM). Whether you're a seasoned Java developer or just starting on your way, *Griffon in Action* will give you the knowledge to get started writing desktop applications in a productive manner and—why not?—have some fun while you're at it.

Some experience with Java Swing is assumed. Previous experience with Grails is an advantage, but we take the time to explain the crucial concepts where we think a common base should be explicitly stated. If you're coming from another language background (such as Ruby or Python), you should find that using the Groovy language comes naturally.

Roadmap

Griffon in Action gives a quick, accessible, no-fluff introduction to writing desktop applications in the Java universe.

The book is divided into four parts:

- Part 1 Getting started
- Part 2 Essential Griffon

- Part 3 Advanced Griffon
- Part 4 Extending Griffon's reach

We cover what Griffon is in chapter 1: where did it come from, and why was such a development platform needed in the first place? This chapter presents theory along with a good deal of practical advice and code—we want you to get a quick start right off the bat.

In chapter 2, we explain the configuration options for an application both at compile time and runtime. The command-line tools are discussed extensively.

In part 2 of the book, we go deep into the Griffon's lair and explore the MVC components found in every Griffon application. Our first stop is modeling data and establishing automatic updates via binding. We hope that by the end of chapter 3, you'll agree that binding makes life much easier that manually wiring up triggers; and event listeners will be a task you cross off your list permanently.

Walking further into the den of the beast in chapter 4, we'll discuss several techniques for building a UI. Declarative programming is certainly within your reach, and the fact that Griffon uses Groovy—a real programming language—makes things much sweeter. You'll find that the relationships between the different components emerge naturally as you progress.

Closer to the nest, in chapter 5, are the components that form the logic of an application: controllers and services. They're responsible for routing events and data, as well as responding to user events.

All the pieces will have fallen into place at this point, but you may have some unanswered questions regarding the relationships between components. Chapter 6 covers in great detail how the platform manages its components and the facilities it puts at your disposal to make the most out of them.

In part 3, we progress to more advanced topics. Building a responsive application can be a daunting task, but in chapter 7 we'll show you a few options that will help you sort out multithreading obstacles with ease. Dealing with highly coupled components is equally intimidating; but, fortunately, Griffon lets you react to well-timed events depending on the application's life cycle. You can even trigger your own events. And did we mention that the event system is also useful for the command line? Events are essential to building an application, and we'll show you how to use them.

Chapter 8 offers complete coverage of notifications. Then, we'll move to an often-neglected aspect of desktop applications: proper testing, involving the UI. Griffon simplifies that task as well, as we'll explain in chapter 9.

Finally, we get into the subject of deployment in chapter 10. We cannot stress enough how important it is to package the application in a way that customers can start using it immediately. Griffon provides highly configurable options to gift-wrap that application, and you need only concern yourself with how you'll ship it to your customers. The beast should be tamed by now and comfortably accepting your commands.

We'll begin part 4 by flying the friendly skies of plugins and extensions. We'll bank left to chart our way through chapter 11. Plugins, a key Griffon feature, let you as a

developer customize further how applications are built and packaged, for example. In chapter 12, we'll climb up to the highest clouds, close to the stars, where the imagination roams freely through the vast expanse of customized views.

Before we complete our journey and shoot for the stars, you'll put all your newfound knowledge and training to the test in chapter 13. We'll show you how to build a prototype application that spans both desktop and web spaces, thanks to friendly cooperation between Griffon and Grails.

You'll want to keep your flying steed well nourished and in excellent condition. In chapter 14, we'll look at the most common tools, such as editors and build tools, that you can use to maximize Griffon's performance.

Code conventions

This book provides examples that demonstrate in a hands-on fashion how to use Griffon features. Source code in listings or in text appears in a `fixed-width font like this` to separate it from the ordinary text. In addition, class and method names, object properties, and other code-related terms and content in text are presented using the same `fixed-width font`.

Code and command-line input/output can be verbose. In some cases, the original source code (available online) has been reformatted; we've added line breaks and reworked indentation to accommodate the page space available in the book. In rare cases, when even this was not enough, line-continuation markers were added to show where longer lines had to be broken.

Code annotations accompany many of the listings, highlighting important concepts. In some cases, numbered cueballs link to additional explanations that follow the listing.

Source code downloads

You can access the source code for all examples in the book from the publisher's website: www.manning.com/GriffoninAction. All source code for the book is hosted at GitHub (github.com), a commercial Git hosting firm. We'll maintain the current URL via the publisher's website, also mirrored at https://github.com/aalmiray/griffoninaction. To simplify finding your way, the source code is maintained by chapter.

Software requirements

All you need to get started is a working version of Oracle's JDK6 (available from http://java.oracle.com) that matches your platform and operating system plus the latest stable Griffon release (from http://griffon.codehaus.org/download). Additional software may be required, such as plugins or tools; we'll provide download instructions when applicable.

Staying up to date

We wrote the book as Griffon evolved, targeting 0.9.5 specifically, however subsequent Griffon versions may have been released by the time you read this. New Griffon versions bring new functionality, and although Griffon reached 1.0 status right about the time this book was finished, the Griffon team made sure to keep away from introducing breaking changes after 0.9.5 was released. This means all the knowledge you learn here is valid for future releases.

If portions of source code require modification for a future release, you'll be able to find information on the *Griffon in Action* Author Online forum (www.manning.com/GriffoninAction).

You can also use the Author Online forum to make comments about the book, point out any errors that may have been missed, ask technical questions, and receive help from the authors and from other users.

About the authors

ANDRES ALMIRAY is a Java/Groovy developer and Java Champion, with more than a decade of experience in software design and development. He has been involved in web and desktop application developments since the early days of Java. His current interests include Groovy and Swing. He is a true believer in open source and has participated in popular projects like Groovy, Grails, JMatter, and DbUnit, as well as starting his own projects. Andres is a founding member and current project lead of the Griffon framework. He blogs periodically at http://jroller.com/aalmiray and is a regular speaker at international conferences. You can find him on twitter as @aalmiray.

DANNO FERRIN is a component lead engineer with experience in Java, Groovy, and Swing. He's the cofounder of Griffon, an active committer to the Groovy language, and a former committer to both Tomcat and Ant.

JAMES SHINGLER is the lead technical architect for Big Lots (a nationwide retailer base in Columbus, Ohio), a conference speaker, an open source advocate, and coauthor of *Beginning Groovy and Grails* (2008). The focus of his career has been using cutting-edge technology to develop IT solutions for the retail, insurance, financial services, and manufacturing industries. He has 14 years of large-scale Java experience and significant experience in distributed and relational technologies.

about the cover illustration

The figure on the cover of *Griffon in Action* is captioned "An inhabitant of Breno." The illustration is taken from a reproduction of the travel logs of Francesco Carrara (1812–1854), a historian and archaeologist, who traveled extensively through Dalmatia, Northern Italy, and Austria, recording his impressions of the history, politics, and customs of the places he visited. The travel logs, accompanied by finely colored illustrations, give a rare and detailed account of regional life in that part of Europe in the mid-nineteenth century. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian's retirement palace from around AD 304.

Breno is a small town in the province of Brescia in the Lombardy region of Italy. The town is the historical capital of the Valcamonica, the valley formed by the river Oglio as it flows through the surrounding Alps. The area is famous for its petroglyphs dating from around 20,000 BC, which are listed among UNESCO's World Heritage Sites.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It's now hard to tell apart the inhabitants of different continents, let alone of different hamlets or towns separated by only a few miles. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought to life by illustrations from old books and collections like this one.

Part 1

Getting started

Our goal in part 1 is to get you up to speed on what Griffon offers to the desktop application development experience by diving directly into code. Part 1 is all about hitting the ground running.

We'll introduce you to Griffon by guiding you through building your first Griffon application: a simple multitabbed file viewer. You'll experience most of the tasks required to design, build, package, and deploy an application; and we'll take a quick look at the building blocks of the framework, its conventions, and the application's life cycle.

Taking inspiration from mythology, a Griffon (or Griffin) is a mystical beast that's half eagle, half lion. In antiquity, the lion was considered the king of beasts, while the eagle held the same title for birds. Thus an amalgam of both creatures results in the king of all creatures. The Griffon framework is an amalgam between the web world (thanks to its Grails heritage) and the desktop world. Griffons were thought to guard treasures and riches; in our case, Griffon is the key to a productive experience when writing desktop applications.

Let's begin our journey by looking the Griffon directly in the eye.

1 Welcome to the Griffon revolution

This chapter covers

- What Griffon is all about
- Installing Griffon
- Building your first Griffon application
- Understanding how Griffon simplifies desktop development

Welcome to a revolution in how desktop applications are designed, developed, and maintained. You may be wondering, a revolution against what exactly? Let's begin with how you pick the application's source layout, or how you organize build time versus runtime dependencies. What about keeping the code clean? How do you deal with multithreading concerns? Can you extend an application's capabilities with plugins? These are but a few of the most common obstacles that must be sorted out in order to get an application out the door. Many hurdles and obstacles lurk in your path, waiting their turn to make you slip that important deadline or drive you to frustration.

Griffon is a revolutionary solution that can make your job easier while bringing back the fun of being programmer. Griffon is a Model-View-Controller (MVC) based, convention-over-configuration, Groovy-powered desktop application development framework. Using Griffon to build your desktop applications will result in

organized code and less of it. But why would you build a desktop application in the first place? There are times when being close to the metal pays off really well: for example, how would you access a local device like a scanner or a printer from a web page? Via some other domain-specific device, perhaps? This is a valid use case scenario in both financial and health industries. We believe that once you use Griffon, you'll enjoy it as much as we do.

This chapter will get you started building Griffon applications. It lays out the core concepts and underlying designs behind the framework. You'll start by getting your development environment set up and building your first Griffon application. You'll build on your first application and create a simple tab-based editor with a menu and actions to open and save files. We'll review some of the challenges with Java-based desktop development and see how Griffon approaches it. Along the way, we'll discuss some of the core Griffon constructs, components, and philosophy.

Are you ready to become truly productive building applications for the desktop? Let's begin!

1.1 Introducing Griffon

Griffon's goal is to bring the simplicity and productivity of modern web application frameworks like Grails and Rails to desktop development. Griffon leverages years of experience and lessons learned by Grails, Groovy, Rails, Ruby, Java Desktop, and Java developers and their communities. Griffon has adopted many of those languages' and frameworks' best practices, including Model-View-Controller, convention-over-configuration, a modern dynamic language (Groovy), domain-specific languages (DSLs), and the builder pattern.

Web application development as we knew it suddenly changed in 2004, when a framework named Ruby on Rails (RoR; http://rubyonrails.org) was released in the wild. It showed that a dynamic language like Ruby could make you highly productive when teamed with a well-thought-out set of conventions. Add the convention-over-configuration paradigm and the viral reception from disheartened Java developers longing for something better than JEE, and RoR suddenly stepped into the spotlight.

A year later, another web framework appeared: its name was Grails, and Groovy was its game. It followed RoR's ideals, but its founders decided to base the framework on well-known Java technologies such as the Spring framework, Hibernate, SiteMesh, and Quartz. Grails included a default database and a full stack to develop JEE applications without the hassle that comes with a regular JEE application.

Grails grew in popularity and a community was created around it, to the point that it's now the most successful and biggest project at the Codehaus (www.codehaus.org), an organization that hosts open source projects; that's where Grails was born and Griffon is hosted.

Grails is a convention-over-configuration, MVC-based, Groovy-powered web application development framework. Does that definition sound familiar? Just exchange *desktop* for *web*, and you get Griffon.

Both frameworks share a lot of traits, and it's no surprise that Griffon's MVC design and plugin facility were based on those provided by Grails, or that the command-line tools and scripts found in one framework can also be found in the other. The decision to use Grails as the foundation of Griffon empowers developers to switch between web and desktop development: the knowledge gathered in one environment can easily be translated to the other.

> **NOTE** If you're in a hurry to understand how to use plugins, take a quick peek at chapter 11.

Let's get started by setting up the development environment and building your first simple Griffon application.

1.1.1 Setting up your development environment

In order to get started with Griffon, you'll need the following three items in your toolbox: a working JDK installation, a binary distribution of the Griffon framework, and your favorite text editor or IDE.

First, make sure you have the JDK installed. The version should be 1.6 or later: to check, type `javac -version` from your command prompt.

Next, download the latest IzPack-based Griffon distribution from http://griffon.codehaus.org/download. The file link looks like this one:

```
griffon-0.9.5-installer.jar
```

Note that the version number may differ. The important thing is that you pick the IzPack link. IzPack provides a cross-platform installer that should take care of installing the software and configuring the environment variables for you. It will even unpack the source distribution of the framework, where you can find sample applications that are useful for learning cool tricks. You can run the installer by locating the file and double-clicking it. Alternatively, you can run the following command in a console prompt:

```
java –jar griffon-0.9.5-installer.jar
```

If for some reason the installer doesn't work for you, or if you'd rather configure everything by yourself, download the latest Griffon binary distribution from the same page in either zip or tar.gz format. Uncompress the downloaded file into a folder of your choosing (preferably one whose name doesn't contain whitespace characters). A standard Griffon distribution contains all the files and tools you need to get going, including libraries, executables, and documentation.

> **CAUTION** If you're working on a Windows platform, avoid installing Griffon in the special Program Files directory, because the operating system may impose special restrictions that hinder Griffon's setup.

Next, set an environment variable called `GRIFFON_HOME`, pointing to your Griffon installation folder. Finally, add GRIFFON_HOME/bin (or %GRIFFON_HOME%\bin on Windows) to your path:

- *OS X and Linux*—This is normally done by editing your shell configuration file (such as ~/.profile) by adding the following lines:

```
export GRIFFON_HOME=/opt/griffon
export PATH=$PATH:$GRIFFON_HOME/bin
```

- *Windows*—Go to the Environment Variables dialog to define a `GRIFFON_HOME` variable and update your path settings (see figure 1.1).

Verify that Griffon has been installed correctly by typing `griffon help` at your command prompt. This should display a list of available Griffon commands, confirming that `GRIFFON_HOME` has been set as expected and that the `griffon` command is available on your path. The output should be similar to this:

```
$ griffon help
Welcome to Griffon 0.9.5 - http://griffon.codehaus.org/
Licensed under Apache Standard License 2.0
Griffon home is set to: /opt/griffon
```

Be aware that Griffon may produce output in addition to this—particularly when run for the first time, Griffon will make sure it can locate all the appropriate dependencies it requires, which should be available in the folder where Griffon was installed.

Griffon commands

The `griffon` command is the entry point for other commands, such as the `help` command you just used. It's a good idea to familiarize yourself with the additional commands because they're useful when you're developing Griffon applications. Using Griffon's command line will be explored further in chapter 2.

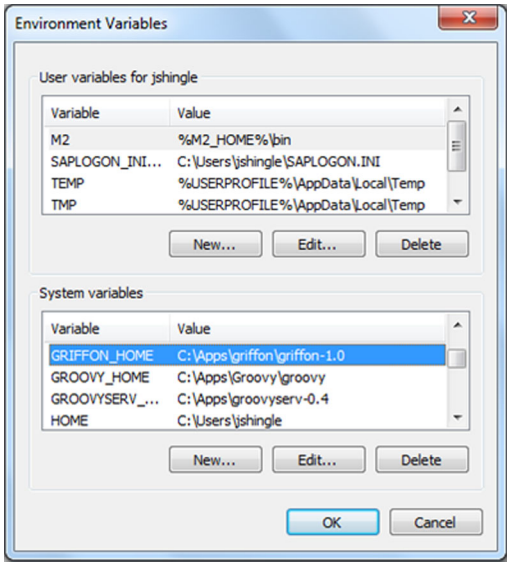


Figure 1.1 Updating variable settings on Windows

Now you're ready to start building your first application. You'll start with a default Griffon application and evolve it into a simple tab-based editor with a menu and actions to open and save files. The application is small enough that you don't need to use an IDE. The goal is to learn how Griffon works, and using an IDE right now would just add an extra layer for you to figure out.

The first order of business in developing an application is setting up the directory layout and defining references to the Griffon framework.

1.1.2 Your first Griffon application

Fortunately, you can do all the bootstrapping plus a bit more with a simple command. All Griffon applications use the `create-app` command to bootstrap themselves. Enter the following in your command-line window:

```
$ griffon create-app groovyEdit
```

That's it! You can give yourself a pat on the back, because you've already done a lot of the work that would have taken you considerably longer in regular Java/Swing. The `create-app` command created the appropriate directory structure, the application, and even skeleton code that can be used to launch the application.

> **A quick peek at a simple Swing application**
>
> If you're new to Swing, listing 1.9 in section 1.3 is a simple Java Swing application. It will give you an idea of how Java desktop development was done before Griffon.

But don't take our word for it; take it for a spin. Make sure you're in the main folder of your new application structure:

```
$ cd groovyEdit
```

Then type the following command at the command prompt:

```
$ griffon run-app
```

On issuing that command, you should see Griffon compiling and packaging your sources. After a few seconds, you'll see a screen similar to figure 1.2.

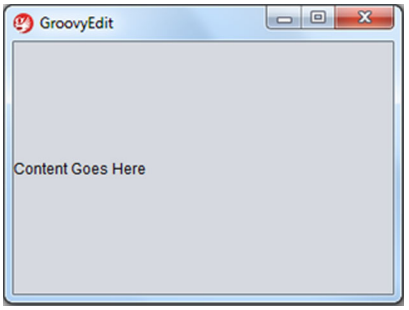


Figure 1.2 Your first application is up and running in standalone mode.

Granted, it doesn't look like much yet, but remember that although you haven't touched the code, the application is up and running in literally seconds.[1] You ran the application from the command line, but that isn't your only option.

[1] That is one of the advantages of the convention-over-configuration paradigm.

Java became famous in 1995 because it was possible to create little applications called *applets*[2] that run in a browser. Java also provides a mechanism for delivering desktop applications across the network: Java Web Start. Although powerful, these options carry with them the burden of configuration, which can get tricky in some situations. Wouldn't it be great if Griffon applications could run in those two modes as well, without the configuration hassle?

As you'll quickly discover, Griffon is all about productivity and having fun while developing applications. That means it's possible to provide these deployment options in a typical Griffon way. Close the GroovyEdit application if it's still running. Now, type the following command, and you'll launch the current application in Web Start mode:

```
$ griffon run-webstart
```

You should see Griffon compiling and packaging your sources. After a few seconds, you'll see a screen similar to figure 1.3.

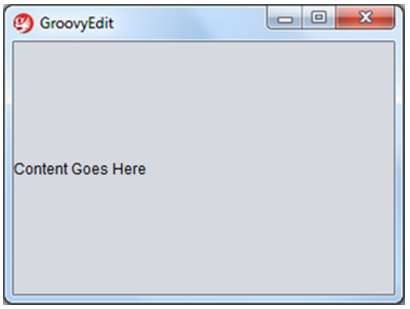


Figure 1.3 Your first application running in Web Start mode

Notice that Griffon performs some additional tasks, such as signing the Java archives (jars). You'll also see the Java Web Start splash screen and a security dialog asking you to accept the self-signed certificate. After you accept the certificate—which is OK because the application isn't malicious in any way—you should again see a screen similar to figure 1.2.

Finally, you can run the application in applet mode with the following command:

```
$ griffon run-applet
```

You should see Griffon compiling and packaging your sources. After a few seconds, you'll see a screen similar to figure 1.4.

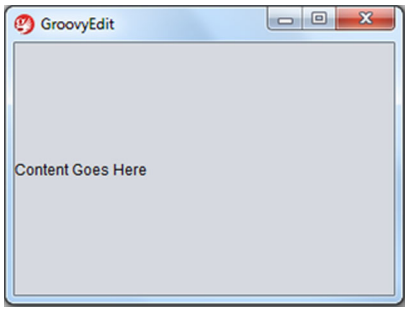


Figure 1.4 The GroovyEdit application running in applet mode

This command signs the application's jars as well, if they're not up to date. But if you launched the applet mode after the previous step, you won't see the jars being signed. You're asked again to accept the certificate if you didn't do so previously. Then, after a moment, you should see the application running again, using Java's applet viewer.

Bearing in mind that you can deploy the application in any of these three modes, we'll continue with the standalone mode for the rest of the chapter, because it's the

[2] Who could forget the Dancing Duke and Nervous Text applets?

fastest (it doesn't require signing the jars that have been updated when you compile the sources repeatedly). We'll cover deployment options in greater detail in chapter 10, where you'll even learn to create a cross-platform installer with minimal configuration from your side.

We hope you're getting excited about the painless configuration: so far, you haven't done any! In the next section, you'll build on this great start and create an editor.

1.2 Building the GroovyEdit text editor in minutes

Many consider Swing application development painful. "Aaargh, Swing!" sums up this attitude. Swing development isn't easy, and time to market suffers because of all the tweaking required. There's truth in these complaints, at least partly because the Swing toolkit is more than 10 years old. It's powerful, but it requires too much work for a new developer to come to terms with quickly. Add to that the perils of Java's multithreaded environment and the verbosity of Swing's syntax, and the life of a Swing developer, especially a newbie, isn't easy.

Given these hurdles, is it even possible to build a meaningful Swing application in minutes? The answer is, of course, "Yes!" One of the core features of Griffon is a powerful domain-specific language (DSL) that overcomes the issues we just mentioned. SwingBuilder is a core Griffon component that allows you to easily create an application using Swing. You're about to find out how easy using Swing can be.

In this section, you'll expand your GroovyEdit application by adding tabs, a menu structure, and the ability to open, save, and close files. At the end, you should have a working application that looks similar to figure 1.5.

At the next stop in your journey, you'll add a bit of spice to the application by changing the way it looks. To do so, you'll modify your application's view.

1.2.1 Giving GroovyEdit a view

The goal we've set for this chapter is to create an application that looks like figure 1.5, which clearly doesn't resemble figure 1.3. A quick glance at figure 1.5 reveals the following elements:

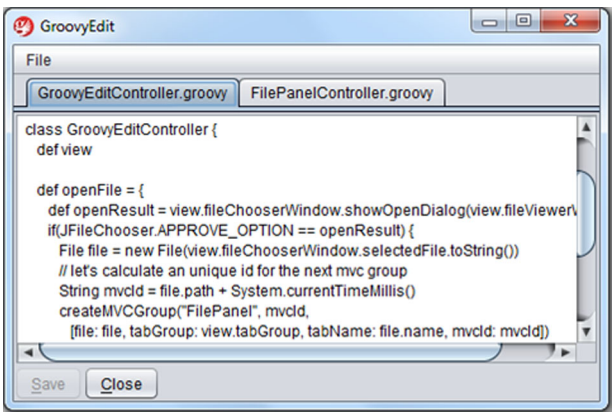


Figure 1.5 Finished GroovyEdit application displaying two tabs with its own source code

- The menu bar has a single visible menu item (File).
- Each tab displays the file name as its title.
- The contents area has both vertical and horizontal scrollbars.
- Each tab includes a Save button and a Close button. Those buttons have a mnemonic set on their label.
- The Save button is disabled.

You're ready to roll up your sleeves and start coding! You'll start by editing the application's view.

UNDERSTANDING THE ROLE OF THE VIEW

Griffon follows the MVC pattern (Model-View-Controller). This means the smallest unit of structure in the application is an *MVC group*. An MVC group is a set of three components, one for each member of the MVC pattern: model, view, and controller. Each member follows a naming convention that's easy to follow. We'll look more closely at the MVC paradigm in section 1.4.

Griffon created an initial MVC group for the application when you issued the `create-app` command. Equipped with this information, let's look at the *view*: the part of the application the user sees and interacts with (see the following listing). This file is located at `griffon-app/view/groovyedit/GroovyEditView.groovy`.

Listing 1.1 Default GroovyEditView

```
package groovyedit
application(title: 'groovyEdit', size: [320,340], 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
}
```

Griffon uses a declarative programming style to reduce the amount of work required to build an application. From this code, you can see that the `create-app` command defines an application titled GroovyEdit with a default size of 480 by 320, some icons, and a Content Goes Here label.

Let's take a closer look. One of the goals of Griffon is to simplify and shield you from implementation details. Java can be a bit of a hassle: desktop applications extend `javax.swing.JFrame`, but applets extend `javax.swing.JApplet`. Griffon takes care of this for you. In listing 1.1, the `application` node resolves to a `javax.swing.JFrame` instance when run in standalone mode and a `javax.swing.JApplet` instance when run in applet mode. After the code sets some basic properties, such as the title and the location, in the application node, the `label` component resolves to `javax.swing.JLabel`.

Next you'll move forward with the application by adding a file chooser (`JFileChooser`), a menu structure (`JMenuBar`), and a tab structure (`JTabbedPane`).

SwingBuilder naming conventions

Swing components in Groovy follow naming conventions. Let's take `JLabel`, for example. Its corresponding Griffon component is `label`. Can you guess what the corresponding component is for `JButton`? If you guessed `button`, you're correct!

The naming convention is roughly this: remove the prefixing *J* from the Swing class name, and lowercase the next character. We'll discuss declarative UI programming with Groovy thoroughly in chapter 5, but for now this tip can save you from some head-scratching as you read this chapter.

ADDING UI ELEMENTS

Following the preferred convention-over-configuration approach laid out by Griffon, the GroovyEditView.groovy file should contain all the view components this MVC group will work with. Replace the contents of the entire file (listing 1.1) with the code in the following listing.

Listing 1.2 Adding menus and a tabbed pane to GroovyEditView.groovy

```
package groovyedit
fileChooserWindow = fileChooser()                                  ❶ Declare FileChooser
fileViewerWindow = application(title:'GroovyEdit', size: [480,320],
     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 {                                                      Declare menuBar and
        menu('File') {                                             menu structure
            menuItem 'Open'
            separator()
            menuItem 'Quit'
        }
    }
    borderLayout()
    tabbedPane(id: 'tabGroup', constraints: CENTER)                ❷ Declare reference via id
}
```

By now, you can begin to appreciate a few advantages of using a general programming language like Groovy instead of a markup language like XML for declarative UI programming. The code is close to what you would have written in Java, yet the verbosity is kept to a minimum; there's hardly a trace of visual clutter.

NOTE If you're not that familiar with Groovy, please refer to *Groovy in Action* (www.manning.com/koenig2/). For now, think of Groovy as a superset of Java with shorthand notations to make your programming life easier.

In order to refer to these components from other files, you need to declare references for the file chooser and the tabbed pane. The return value of the first node call (`fileChooser`) is kept as an explicit variable ❶ as you would in regular Groovy code.

The second way to define a reference is by setting an `id` property ❷ on the target node. In this case, a variable named `tabGroup` is created that can be referenced from the view script. The advantage of the second approach, as you'll see later in the book, is that you can create variable names in a dynamic way.

Having done this, you can refer back to these components from the other files in your application, while at the same time ensuring that all the view components are in the same place. Imagine how useful that will be for someone maintaining the application. They'll know exactly where to go to find the application's view components.

Run the application by typing the following Griffon command at the command prompt:

```
$ griffon run-app
```

When you do so, you should see a screen similar to figure 1.6.

ADDING THE MENU ITEMS

Next, you'll spend some time working with the menu items. You've hard-coded the names of the actions into the view of your application. Griffon lets you separate your action code from the rest of your application. Defining an action also leads to code reuse, because many Swing components can use the action definition to configure themselves—for example, their label and icon—and also to handle the job they're supposed to do.

You'll define two actions in `GroovyEditView.groovy`. Note the id of each action:

```
actions {
   action(id: 'openAction',
      name: 'Open',
      mnemonic: 'O',
      accelerator: shortcut('O'))
   action(id: 'quitAction',
      name: 'Quit',
      mnemonic: 'Q',
      accelerator: shortcut('Q'))
}
```

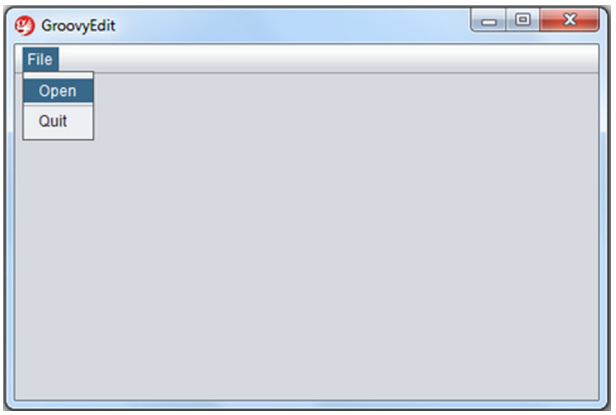


Figure 1.6 The GroovyEdit application now has a menu.

This code must precede the code that uses the actions. For example, you could insert it before the application node or just before the application node is defined.

In the definition of your menu items, change `menuItem 'Open'` to `menuItem openAction`. Do the same for the Quit action:

```
menuBar {
   menu('File') {
      menuItem openAction
      separator()
      menuItem quitAction
   }
```

Your `GroovyEditView.groovy` file should now look like the following listing.

Listing 1.3 Full source of GroovyEditView.groovy

```
package groovyedit
actions {
   action(id: 'openAction',
      name: 'Open',
      mnemonic: 'O',
      accelerator: shortcut('O'))
   action(id: 'quitAction',
      name: 'Quit',
      mnemonic: 'Q',
      accelerator: shortcut('Q'))
}

fileChooserWindow = fileChooser()
fileViewerWindow = application(title:'GroovyEdit', size:[480,320],
     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 {
      menu('File') {
         menuItem openAction
         separator()
         menuItem quitAction
      }
   }
   borderLayout()
   tabbedPane id: 'tabGroup', constraints: CENTER
}
```

Set open action

Set quit action

Run the application again, and you'll see the newly defined properties of the menu items as shown in figure 1.7. They now include mnemonics and accelerators, honoring the current operating system.

What about adding some behavior to the menu items? A controller is the appropriate location for behaviors.

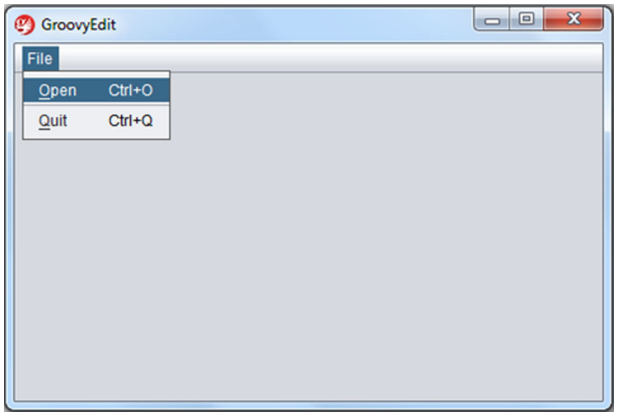


Figure 1.7 Native menu accelerators (Windows)

1.2.2 Making the menu items behave: the controller

Remember the MVC group that was created by default along with the skeleton code? One of the files that was created has the controller responsibility; its name should be `GroovyEditController.groovy`, per the naming convention, and it should be located in the `griffon-app/controllers/groovyedit` folder.

> **Griffon naming conventions**
>
> By now you'll probably have discovered the basics of the Griffon naming conventions. It should follow that if the view and controller for the current MVC group are called GroovyEditView and GroovyEditController, respectively, the model should be GroovyEditModel. It should also be evident that if the view's location is griffon-app/views and the controller's is griffon-app/controllers, the model's location should be griffon-app/models. We'll begin looking at models in section 1.2.4 and cover the directory layout in more detail in the next chapter.

Now you'll edit the controller file and define the behavior of your actions, as shown in the next listing.

Listing 1.4 GroovyEditController with two actions

```
package groovyedit
import javax.swing.JFileChooser
class GroovyEditController {
    def view

    def openFile = {
        def openResult = view.fileChooserWindow.showOpenDialog()
        if( JFileChooser.APPROVE_OPTION == openResult ) {
        }
    }

    def quit = {
```

To be defined shortly

Define file action

1 Define quit action

```
            app.shutdown()
        }
}
```

You start by declaring a view field, which, when the application is running, will point to an instance of the view script. The framework will make sure to supply the correct instance. That's great, because now you can refer to the view components you've defined there (through the references you created, remember?).

Let's move on with the implementation of the behavior for the Quit and Open File actions. In the case of the Quit definition ❶, notice that you also have access to the application as a whole, via the app reference—another handy reference automatically injected at runtime.

To declare a variable or not

You may have noticed that a `view` property is defined for the controller whereas the `app` variable is not. Still, the code will compile and work as expected. What's going on? How can you tell which variable needs to be declared and which doesn't? In a nutshell, every model, view, and controller class has direct access to the application's instance via the `app` variable that's always injected. In contrast, other variables must be declared explicitly. Don't worry: we'll spell out all the various properties you may declare in each artifact type as we continue the book. Chapter 6 discusses all things MVC.

Because Griffon handles the life cycle of the application for you, you can call the `shutdown()` method on the application reference. It should then shut down gracefully when called. If you're interested in the application's life cycle and how the framework manages it, be sure to read section 2.4 in the next chapter.

With this simple action completed, let's focus on the business logic of your application, which in this case is the opening of a file. That's where the `openFile()` method comes into play. The code is similar to what you can find in a regular Swing application, except that you refer to the components in your view.

You refer to the `fileChooser` that you defined in the view, after which you need to add handling code for dealing with the file that has been opened.

Closure support in Griffon

Both `openFile()` and `quit()` look like methods, but they're actually something different. Groovy has support for *closures*—reusable blocks of code. Closures are far more versatile than methods in many situations, as you're about to see.

You're almost ready to try it again, but first hook up the previous behavior to each action. Return to the view file, and change the action as follows:

```
action(id: 'openAction',
        name: 'Open',
        mnemonic: 'O',
        accelerator: shortcut('O'),
        closure: controller.openFile)
```

Note the `closure` attribute: it links back to the controller file where your behavior is neatly organized in a closure. Also notice the symmetry in the available field names: the controller has access to a view instance, and the view has access to a controller instance. Again, this way, a maintainer of your code will know where each piece of the application is found: views in the view file and behavior in the controller file.

Make sure you also change the definition of the Quit action in the view file, so it links back to the `quit` closure in the controller file.

If you run the application at this point and click the Open menu, a `fileChooser` should appear, but nothing else happens if you select a file and click the Open button. You haven't worked on that part of the application yet! You'll do so in the next section via a second MVC group, which will provide the view, the controller, and the model for every tab.

1.2.3 How about a tab per file?

In many modern editors, when you open a new file, it's opened in a new tab with the file name displayed on the tab. To implement this UI functionality, you'll create a MVC group to display the file that you've opened via the Open action. The new MVC group is where you'll provide the view, controller, and model for that tab.

Return to the command prompt in the GroovyEdit application's root folder, and run this command:

```
$ griffon create-mvc filePanel
```

You named the group `filePanel` because you'll use a panel container, but you could have chosen `FileTab` or some other name that indicates this group is related to files and tabs. Again you end up with three files that follow the MVC pattern, each organized in a specific folder.

CREATING THE VIEW FOR THE FILEPANEL MVC GROUP

You'll begin by adding some content to the view, whose name you should be able to figure out by now: `FilePanelView.groovy`. See the following listing.

Listing 1.5 FilePanelView with a tab, a text area, and buttons

```
package groovyedit
tabbedPane(tabGroup, selectedIndex: tabGroup.tabCount) {
   panel(title: tabName, id: 'tab') {
      borderLayout()
      scrollPane(constraints: CENTER) {
         textArea(id: 'editor')
      }
      hbox(constraints: SOUTH) {
         button 'Save'
```

1 Define tab

```
            button 'Close'
        }
    }
}
```

Your view defines a tabbed pane this time, instead of an application. Because you'll embed this particular view in a tabbed pane, there's no need for a top-level window. Although at this moment it may appear that you'll create a new tabbed pane each time a view of this type is instantiated, nothing could be further from the truth. Review the code again and notice that the `tabbedPane` references a `tabGroup` variable ❶, which you'll probably remember as a variable from listing 1.2. The tabbedpane component is one of the many smart Groovy Swing components that know when they should create a new instance and when they should reuse a previous instance, as is the case here.

Tabbed panes accept any Swing node as content. Provided you set a title for them (and a few additional properties), that title value will be used as the tab's title. Because you want the tab's title to be the current file being edited, you can't set it to a particular value; but you can set it to a variable with a value that will be determined at a later point in your application, in a controller.

CONNECTING TWO MVC GROUPS

You may wonder how you connect the `filePanel` MVC group to the `groovyEdit` MVC group. Back in the `GroovyEditController`, fill out the `if` clause of the `openFile()` action as follows:

```
if( JFileChooser.APPROVE_OPTION == openResult ) {
    File file = view.fileChooserWindow.selectedFile
    String mvcId = file.path + System.currentTimeMillis()
    createMVCGroup('filePanel', mvcId,
        [file: file, tabGroup: view.tabGroup, tabName: file.name, mvcId:
        mvcId])
}
```

Every member of an MVC group is able to instantiate another MVC group, via `createMVCGroup()`. Although the particulars of MVC groups will be covered in chapter 6, know that this method requires three arguments: the type of group to be created, a unique identifier, and additional values that can be useful when setting up each member.

Take careful note of the values you're passing to the `filePanel` MVC group. For example, the file that has been opened is one of these values, as well as its name. Within the controller of the `filePanel` MVC group, you'll use these values to initialize the model you'll create there. That will expose these values to the rest of the MVC group in a neat and consistent manner. Remember the `tabGroup` and `tabName` variables in FilePanelView.groovy? Now you know where their values come from.

Before going further, run the application again. You should see a window like the one shown in figure 1.8 (you've opened a few files).

Nothing is shown in the tabs yet, because you haven't added the necessary code. But the name of the tab is the name of the file you instructed the code to open. In the

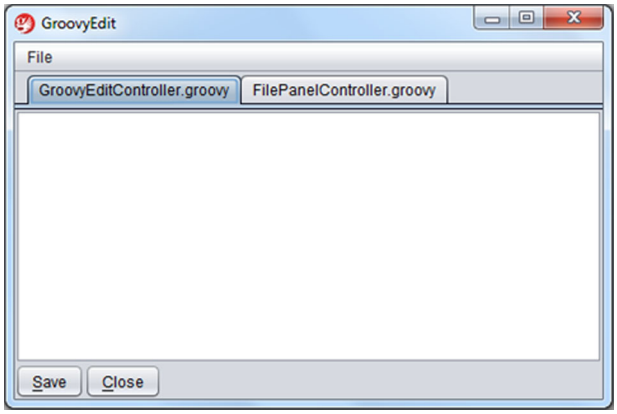


Figure 1.8 GroovyEdit displaying two tabs. But where is the content?

next section, you'll add the missing pieces: reading the content of the file and enabling the Save and Close buttons.

1.2.4 Making GroovyEdit functional: the FilePanel model

A few variables should be shared consistently between the script providing the view and the script providing the behavior. Models fit perfectly for that responsibility, mediating data between controllers and views. To that end, edit the FilePanelModel.groovy file, which is in the griffon-app/models/groovyedit folder, and add the following definition.

Listing 1.6 `FilePanelModel` with required model properties

```
package groovyedit
import groovy.beans.Bindable

class FilePanelModel {
   File loadedFile
   @Bindable String fileText
   @Bindable boolean dirty
   String mvcId
}
```

You define four properties: one for the file being edited, another to specify its content as text, a property indicating whether the file's content is changed, and a unique id for the tab.

As you can see, each property in a Groovy class is formed by defining its type and its name. As opposed to what happens in Java, where these properties would be scoped as package-protected fields, these fields will be mapped to their correspondent properties following the Java Beans convention. The Groovy compiler will generate the appropriate bytecode instructions for a pair of methods (the getter and setter) and a private field. Also, notice the `@Bindable` attribute, which sets up Griffon data binding. We'll look at binding in the next section.

Next, as you may have already guessed, you'll configure the controller.

Behind the scenes

In a Java application, you would have to create the getters and setters yourself or use an IDE to generate them. Griffon takes care of this for you. Talk about savings in lines of code! And as an added bonus, the `@Bindable` annotation generates the required code to make each annotated property observable. The property will fire up `PropertyChangeEvent`s whenever the value is modified. Say farewell to boilerplate code.

The particulars of bindings and `@Bindable` will be covered in chapter 3, where we explain the main responsibilities of models. You'll see later in chapters 4 and 5 how controllers and views communicate with each other thanks to bindings set on model properties.

1.2.5 Configuring the FilePanel controller

You have a model, you have a view, and now it's time to finish the MVC group by addressing the controller, as shown in the next listing. The controller brings it all together: it contains the logic to manage loading the file, saving the file, closing the file, and making sure the MVC group is properly initialized.

Listing 1.7 `FilePanelController`'s full implementation

```
package groovyedit
class FilePanelController {
   def model
   def view

   void mvcGroupInit(Map args) {
      model.loadedFile = args.file
      model.mvcId = args.mvcId
      execOutsideUI {
         String text = model.loadedFile.text
         execInsideUIAsync { model.fileText = text }
      }
   }

   def saveFile = {
      execOutsideUI {
         model.loadedFile.text = view.editor.text
         execInsideUIAsync { model.fileText = view.editor.text }
      }
   }

   def closeFile = {
      view.tabGroup.remove(view.tab)
      destroyMVCGroup(model.mvcId)
   }
}
```

1 Inject model and view properties

2 Run outside EDT

Read file

3 Run in EDT

Write file

Remember from the previous controller that a view property was auto-injected at runtime? Well, in this case you also need a model. The framework again figures out the correct type of model it should inject 1, thanks to you defining a model field. Without

going into too much detail at this point, the `init` method initializes the entire MVC group with the values received when the user opens a file. Calls to `execOutsideUI` ❷ and `execInsideUIAsync` ❸ handle the threading for your application. For now, remember the following rule: when doing a computation or an operation that isn't related to the UI, perform it outside the event dispatch thread (EDT), but come back to it if you do need to perform a UI update.

> **Multithreaded Swing applications**
>
> As you're probably aware, the Java platform provides a multithreaded environment for running applications. There's no doubt that concurrent programming is a hard task—add the fact that Swing is a single-threaded UI toolkit, and you get a recipe for disaster. But don't worry: you've only caught a glimpse of what Griffon has to offer to aid you in creating high-performing, multithreaded Swing applications. We'll cover threading in more detail in chapter 7.

Similarly, you interact with your model from the `saveFile` closure, as well as from the `closeFile` closure, while also interacting with your view. This is Griffon's solution to connecting the separate parts that make up your application. It might take some getting used to at first, but it's intuitive.

DEFINING ACTIONS

Next, as you did for the GroovyEdit MVC group, you need to hook the Save and Close behaviors into the view. You do this by using the action's `id` parameter. First, in the `FilePanelView.groovy` class, define these actions before the `tabbedPane` node:

```
actions {
   action(id: 'saveAction',
      enabled: bind {model.dirty},
      name: 'Save',
      mnemonic: 'S',
      accelerator: shortcut('S'),
      closure: controller.saveFile)
   action(id: 'closeAction',
      name: 'Close',
      mnemonic: 'C',
      accelerator: shortcut('C'),
      closure: controller.closeFile)
}
```

Then, you change the buttons so the ids of these actions are hooked into them, as in this case for the Save action:

```
button saveAction
```

Do the same for the Close action.

Finally, let's look at how the most important functionality in your application is implemented.

DISPLAYING THE OPEN FILE'S CONTENTS

How do you set the content of the opened file to the text value of the text area? As you may recall, the file's contents are read when the controller is initialized. Those contents are then saved into a property in the model, which means you should edit the definition of your text area in `FilePanelView.groovy` and bind its `text` property to the file's contents in the model:

```
textArea(id: 'editor', text: bind {model.fileText})
```

IMPLEMENTING THE SAVE BUTTON

You want to make sure that when there's a change to the text in the text area, the Save button is enabled. But it should be disabled if the contents return to their original state (for example, if you edit the text so it's what it previously was).

Add the following bean definition to the end of the view file:

```
bean(model, dirty: bind {editor.text != model.fileText})
```

You've introduced a number of nodes for this particular view. The particulars for working with nodes and views will be discussed in chapter 4. Now, whenever the text in the text area doesn't match the text in the model, the `dirty` boolean's value switches, which enables or disables the Save button.

And that's it. The code for this view is shown in the following listing.

Listing 1.8 Full source of FilePanelView.groovy

```
package groovyedit
actions {
   action(id: 'saveAction',
      enabled: bind {model.dirty},
      name: 'Save',
      mnemonic: 'S',
      accelerator: shortcut('S'),
      closure: controller.saveFile)
   action(id: 'closeAction',
      name: 'Close',
      mnemonic: 'C',
      accelerator: shortcut('C'),
      closure: controller.closeFile)
}

tabbedPane(tabGroup, selectedIndex: tabGroup.tabCount) {
   panel(title: tabName, id: 'tab') {
      borderLayout()
      scrollPane(constraints: CENTER) {
         textArea(id: 'editor', text: bind {model.fileText})
      }
      hbox(constraints: SOUTH) {
         button saveAction
         button closeAction
      }
   }
}

bean(model, dirty: bind {editor.text != model.fileText})
```

Action set

Content area

Button area

Bind model dirty property

Run the application, and it will function as it should. Open a file or two, make some changes, and then save.

We've covered a lot of ground. You have a functional application, and it required no configuration at all. You're also able to launch the application in three different modes, again with no configuration changes from your side. Let's look at some statistics. At the command prompt, type `griffon stats`, and you should see output similar to the following:

```
$ griffon stats
+----------------------+-------+-------+
| Name                 | Files |  LOC  |
+----------------------+-------+-------+
| Models               |     2 |    12 |
| Views                |     2 |    56 |
| Controllers          |     2 |    40 |
| Lifecycle            |     5 |     5 |
| Integration Tests    |     2 |    10 |
+----------------------+-------+-------+
| Totals               |    13 |   123 |
+----------------------+-------+-------+
```

Amazing! The application took 123 lines of code and 13 files, of which you only needed to edit 5. We won't ask you to do mind gymnastics to figure out how much code it would take you to accomplish the same feat with regular Java/Swing—it's too painful and tiresome.

You may argue that although the application is functional, it's lacking in some areas: for example, each tab has Close and Save buttons instead of the application providing Close/Save menu items. There's no Help menu, and the application doesn't confirm that it's saving edits to disk before you quit it. As the book continues, you'll see how to add functionality to the application. The take-away here is that you've built a functional multitabbed editor with relative ease.

In order to gain a deeper understanding of Griffon and its driving goals, let's take a couple of minutes to look at some of the challenges of traditional Java desktop development and follow up with Griffon's approach to the challenges.

1.3 Java desktop development: welcome to the jungle

If you've developed Java desktop applications, take a few moments to reflect on your past experiences. What things prevented you from reaching a goal on time? What practices would you have applied instead? Did the language get in the way instead of helping you? Chances are, you've encountered one or more of the following pain points:

- Lots of boilerplate code (ceremony versus essence)
- UI definition complexity
- Inconsistent application structure
- Lack of application life cycle management
- No built-in build management

Let's review each of these.

1.3.1 Lots of boilerplate code (ceremony vs. essence)

The Java platform is a great place to develop applications, as witnessed by the myriad libraries, frameworks, and enterprise solutions that rely on it. It's also a wonderful host to several programming languages, of which the Java programming language is the first and the most widely used so far. Unfortunately, the language is showing its age (it was introduced in 1995).

The Java programming language was a refreshing change when it was first introduced. Developers around the world were able to pick up the language and jump ship, so to speak, in a matter of weeks. The language's syntax and features were similar to what developers were used to programming with, while at the same time Java included new and desired features baked right into the language, such as threading concerns and the notion that the network should be a first-class citizen. Everybody marveled at it. That is perhaps why there were few complaints about the amount of code it took to perform a simple task, such as printing a sequence of characters to the console. The following code is a descriptive example (the often-used `HelloWorld`):

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Notice how much code has to be written just to print "Hello World!" This is referred to as *ceremony*—stuff you just have to do. Don't get us wrong: this example is much better than what was previously available, but there's still room for improvement. Imagine for a moment that you know little about the Java language (if that's the case, don't worry—we'll explain what's going on with that snippet of code). The *essence* of the program is pretty much described by `System.out.println("Hello World!");`, but as you can see you must type a few additional things to please the compiler.

Every piece of code you write in the Java language must be tied to a class, because Java is an object-oriented language that uses classes to define what an object can do.[3] Thus you must define a `HelloWorld` class. A class may define a method with a special signature (the `main` method) that's used as the entry point of your program, so you define it as well. In that method, you place the code that fulfills the task. How would you explain to a person new to the Java language that they must know all these things (and a few more, including access modifiers and static versus instance class members) just to print a message to the console? All of this just to please the compiler and create a simple example. Wouldn't it be easier if the compiler accepted something like the following?

```
println "Hello World!"
```

It might surprise you to learn that this Groovy example is equivalent to the Java example. If this code makes more sense than the first version, it's because the *essence* of the

[3] As opposed to JavaScript, which uses prototypes to accomplish the same feat.

task has been made explicit: no additional keywords or syntax constructs distract you from understanding what the code does. This is exactly what we mean by *essence versus ceremony*, a term Neal Ford[4] mentions regularly.

Java is a good language to develop applications, but it requires a steeper learning curve than other languages that can run on the JVM—languages that provide the same behavior in many respects, but in a more expressive manner. Griffon addresses this issue through the use of the Groovy language, builders, and plugins.

Strongly related to this point, defining a Java-based UI can be overly complex and verbose.

1.3.2 UI definition complexity

The Java Standard Library, which comes bundled with the Java language when you download the Java Development Kit (JDK) or Java Runtime Environment (JRE), delivers two windowing toolkits: the Abstract Windowing Toolkit (AWT) and Swing. Of the two, Swing is the more widely used, because it's highly configurable and extensible. But those benefits come with a price: you have to write a lot of code to get a decent-looking UI to work; you have to deal with many collaborators and helpers to react to events and provide feedback; and on top of all that, you have to please the compiler again.

Let's review the following example of a basic straight Java application that copies the value of a text widget into another when you click a button.

Listing 1.9 Basic Java Swing application

```
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.SwingUtilities;

public class JavaFrame {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable(){
      public void run() {
        JFrame frame = buildUI();
        frame.setVisible(true);
      }
    });
  }

  private static JFrame buildUI() {
    JFrame frame = new JFrame("JavaFrame");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(
      new GridLayout(3,1)
    );
```

Swing components must be created in EDT

Instantiate frame and set properties on it

[4] You can read Neal's thoughts at http://memeagora.blogspot.com.

```
    final JTextField input = new JTextField(20);
    final JTextField output = new JTextField(20);
    output.setEditable(false);
    JButton button = new JButton("Click me!");
    button.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event) {
        output.setText(input.getText());
      }
    });
    frame.getContentPane().add(input);
    frame.getContentPane().add(button);
    frame.getContentPane().add(output);
    frame.pack();
    return frame;
  }
}
```

Add event listener on button

There have been several attempts to simplify how UIs are created in Java, to the point of externalizing them in a different format; most of the time, the chosen format is XML. For some reason, Java developers have had a strong love/hate relationship with XML since the early days. When a configuration challenge appears or the need to externalize an aspect of an application arises, XML is the choice 99% of the time. The problem with XML is that it's so easy to hurt yourself with it.[5] Once you go down the path of declarative UI programming with XML, you'll eventually find yourself in a world of pain.

In listing 1.9, you can appreciate that the widgets are instantiated, configured with additional properties, and added to a parent container according to the rules of a predefined layout. Behavior is wired in a convenient way—well, as convenient as it can be to define an anonymous inner class as the event handler on the button, which is a common pattern in Swing applications.[6] If you were to define the UI in XML format, you would need to perform at least the following tasks:

- Read the XML definition by means of SAX, DOM, or your own parser.
- Translate the declarative definitions into widget instances.
- Wire the behavior, perhaps by following a particular configuration path or a predefined convention.

Of course, you'll have to deal with exceptions, classpath configuration, and additional setup: in other words, make sure your favorite debugger is close by. We guarantee you'll be reaching for it before the job is done.

Now compare the Java program (listing 1.9) to a Griffon program that does the same thing (in the next listing). The Griffon program uses Groovy and SwingBuilder to remove most of the visual clutter and verbosity while retaining the same essence.

---

[5] Hey, it has these sharp, pointy things < > or didn't you notice? Stay away from sharp edges!

[6] Anonymous inner classes, along with the other three types of inner classes, are advanced concepts that people new to Java tend to stay away from. Go figure.

Listing 1.10 Simplified Swing application

```
import groovy.swing.SwingBuilder
import static javax.swing.JFrame.EXIT_ON_CLOSE

new SwingBuilder().edt {
  frame(title: "GroovyFrame", pack: true, visible: true,
        defaultCloseOperation: EXIT_ON_CLOSE) {
    gridLayout cols: 1, rows: 3
    textField id: "input", columns: 20
    button("Click me!", actionPerformed: {
      output.text = input.text
    })
    textField id: "output", columns: 20, editable: false
  }
}
```

It may be hard to believe at first, but both listings produce the same behavior. The advantages of listing 1.10 should be apparent to the naked eye: you've removed more than half the original lines of code, the relationship between widgets and container is more explicit (they're all contained in a block that is defined in the container), the event handler's code has been reduced to its minimal essence, and some values (like `rows:` and `columns:`) now have a sensible meaning. Griffon's use of Groovy, builders, and plugins makes creating UIs much easier than in the past. And Griffon has the additional benefit of making the code easy to read.

But what about application structure and life cycle issues?

1.3.3 Lack of application life cycle management

Every application requires some structure; otherwise it would be a maintenance nightmare, to say the least. Applications have been built since the Java platform was born, and for a while everybody followed Java best practices to build them. Then Struts came into the web application development scene (http://struts.apache.org/). All of a sudden people realized that a predefined structure that everyone agreed on was a good idea: not only was it possible to recognize the function of a particular component by its place and naming convention, but you could switch from one Struts application to another and reap the benefits of knowing the basic structure. You could be productive from the get-go.

Tied to a particular structure, an application should be able to manage all aspects of its life cycle: what to do to bootstrap itself, initialize its components (perhaps by type, layer, or responsibility), allocate resources, and bind all event handlers, just to name a few common tasks. Take for example LimeWire, a popular Java Swing application used to share files on peer-to-peer networks. Can you imagine the phases the application has to undertake to work properly? What about another popular Java Swing application, NetBeans? Surely the people behind its design have given a lot of thought to how the application should control its life cycle. What would it take for you to achieve the same thing?

It's easy to get lost in the details. It should be more convenient to let a framework resolve those matters for you. Fortunately, the Griffon framework addresses these issues by providing a consistent application structure (see chapter 2, section 2.1) and extensible life cycle event management (see section 2.4).

Finally, there's the matter of building the application in a reliable way while taking care of proper dependency management, version control, and infrastructure upgrades.

1.3.4 No built-in build management

The Java platform offers a number of tools for project management, dependency management, IDE integration, and so on. The problem is choosing the ones that solve a particular problem while also being able to integrate with other tools and being extensible enough. The question isn't whether the tool is extensible but how soon you'll extend the tool. Sooner than later, you'll face that decision.

Choosing a build tool can lead to heated discussions among Java developers, just as picking the best editor[7] does. Many favor the simplicity of Ant; others preach the advances pioneered by Maven, whereas its detractors complain that it requires downloading the whole internet just to execute a simple goal like cleaning the project's build directory. Some have gone outside of the bounds of XML-based tools and have chosen Buildr or Rake/Raven, build tools that rely on expressive languages, or even build-oriented DSLs.

The point is that you might be stuck with a solution that may be unreliable or may not work as expected with your next project due to its particular requirements. Having to manage build configuration and artifacts can be taxing, considering you still have to worry about production and testing artifacts in the first place. Like Grails, Griffon comes with a build facility; you used it when you ran the Griffon command `run-app`.

Now that we've examined the obstacles that lie in the path of building a typical application on the JVM, let's discuss what Griffon does to sort them out and let you reach the goal line.

1.4 The Griffon approach

Having experienced the issues described in the previous section, the creators of the Grails and Griffon frameworks worked to make sure Grails and Griffon minimized if not eliminated them. This section will give you more insight into Griffon and how it works.

First and foremost, every application must have a well-defined kernel as its structure: something that ties together all components given their responsibilities and relationships with one another. The Griffon developers decided to pick the popular Model-View-Controller pattern (MVC) as a basis. This is the framework's shaping element.

The next step was finding the proper balance between *configuration* and expected conditions or *conventions*. We're sure you agree that you'd like to spend more time

[7] Everybody knows that vim is the one and only editor used by real programmers, right?

pushing code to production than figuring out the proper configuration flags and properties to get a particular piece of the application working. Along with common components and tools, this is the framework's constituent element.

Finally, there's the matter of the verbosity the Java language imposes. You want to be productive without needing to learn a new language or leave all your Java knowledge behind. This is where the Groovy language comes in: it's a binding agent between the shaping and constituent elements of the framework.

Let's see how these elements come together to offer you a better experience when you're developing a desktop application.

1.4.1 At the core: the MVC pattern

Nearly every developer who has created a professional GUI has had to work with or use the Model-View-Controller pattern. And because practically every developer has to write a GUI at some point, it's a well-known pattern name. But its details aren't as widely known for several reasons. One is that when people say *MVC*, they can be referring to one of several different closely related patterns. To understand how this came to be, you need to understand where the MVC pattern came from.

A BRIEF HISTORY OF MVC

When web applications started growing in popularity, the MVC paradigm was recast against a physical structure that closely matched its triad: the model was the database tier of the application, the view was the user's web browser, and the controller was the web application. The downside of this arrangement is that the view and the model don't directly communicate with each other. Instead, the controller mediates all interactions.

> **The original MVC code**
>
> The very first code ever to use the Model-View-Controller pattern was written circa 1978 by Trygve Reenskaug, a professor from the University of Oslo, while he was a visiting scientist at ParcPlace. He wrote a couple of research notes on the subject and, after consulting with Adele Goldberg, settled on the nomenclature of model, view, and controller. What is interesting to note is that his original pattern contained a fourth component, the editor, whose exclusion from almost all variations of the MVC pattern heralded the mutable beginnings of the MVC pattern.
>
> The original MVC code led a low-key life in the UI code of Smalltalk-80 for nearly a decade before the first serious academic journal article was published by Glenn Krasner and Stephen Pope in the August/September 1988 issue of the *Journal of Object-Oriented Programming*. This article described MVC as more of a paradigm than a concrete pattern. Many later variations demonstrated the value of this paradigm, notably the Presentation-Abstraction-Controller (PAC) pattern and the Model-View-Presenter pattern from Taligent.

The MVC pattern as followed by modern web applications doesn't literally place the view and the model directly on the browser and the database; instead, all interactions

by the application server between these two pieces pass through code in the application server. The pieces that correspond to communicating with the database and browser are handled in the context of a model and view, respectively, all seen through the glasses of a fixed request/response life cycle.

This model 2 view of MVC is web centric, and it works well when you're mediating systems that are physically or network separated. But that is the exact opposite of what a Rich Internet Application (RIA) is. Not only are all the pieces of the MVC triad local to the same machine, but they also exist in the same process space. It's because of this fundamental difference that Griffon can cast its MVC handing closer to the original vision of MVC.

Griffon and MVC

What does Griffon bring to the table when it comes to MVC? A framework API referred to as MVC groups: a hybrid of the old-school MVC pattern mixed with more modern concepts such as injection and convention over configuration. Griffon automates the creation of the model, view, and controller classes both at build time and at runtime. It configures them via injection to follow the original convention of paths of references, observed updates, and user interactions. It also has established file locations for the created groups and other conventions relating to the life cycle of the MVC group.

The MVC pattern is found in Griffon both at the architectural level and the presentation layer. The latter is due to Swing's inherent nature. Every component is designed to conform to the pattern; Swing components follow the MVC pattern, but most of them combine the view and the controller in the same class (for example, `JButton`) while keeping the model in a separate class (`ButtonModel`). Speaking of the architectural level, this is where the Griffon conventions come into play, as explained next.

Model

The model is responsible for holding the data and providing basic relationships within this data. Some prefer designing an anemic model: that is, they believe the data should be simple, and behavior should go into a different member of the MVC pattern, even establishing data relationships. The advantage of this design is that you know what models are capable of: holding data and nothing more. Anything else is found in the controller. Others prefer a richer domain model, where data can manage its own relationships and even talk to other models. We're happy to say that Griffon supports both visions; you're free to make your models as anemic or rich as you want them to be.

Models in Griffon, not to be confused with domain models, are used exclusively to help controllers and views communicate through data and events. Domain models, on the other hand, describe the application in terms of entities. They're usually rich and often have relationships with each other. An example of a domain model would be `Company`, `Employee`, and `Address` classes, whereas an example of a regular MVC model would be an aggregation of instances of the domain classes, like a `TableModel`.

VIEW

The view is responsible for displaying the data coming from the model in a meaningful manner; it can even transform the data from one representation into another. For example, a list of values can be defined in the model as a simple array or Java collection, and the view may display that data in tabular form using a `JTable` component from the Swing component suite. But if a table isn't desired, the view could display the same data using a pull-down menu or combo box. The data doesn't change, but its view representation can vary as needed.

CONTROLLER

The controller is the command center. Every operation that affects the model from the outside should be scrutinized and given the green light by a controller. The controller then carries the burden[8] of making all choices: whether data should be affected, and what the view's next state should be.

The model may update the view indirectly with new data as long as the view is listening to changes from the model, but it should not cause a change in the view's state without the controller's consent. How can the model update the view? Enter the Observer pattern.

HOW THE OBSERVER PATTERN WORKS

The basic idea behind the Observer pattern is that two components may interact with each other by having one of them push events to the other. The first component is known as the *emitter*, and the second is known as the *listener*. Listeners must register with an emitter before they get a chance to process events; otherwise they won't receive any. Emitters may accept as many listeners as they choose. In Griffon, any MVC member can be registered as a listener on a model class, by means of `PropertyChangeListeners` and `PropertyChangeEvents`.

Figure 1.9 shows the associations between each member. A solid line indicates a direct association, and a dashed line indicates an indirect association.

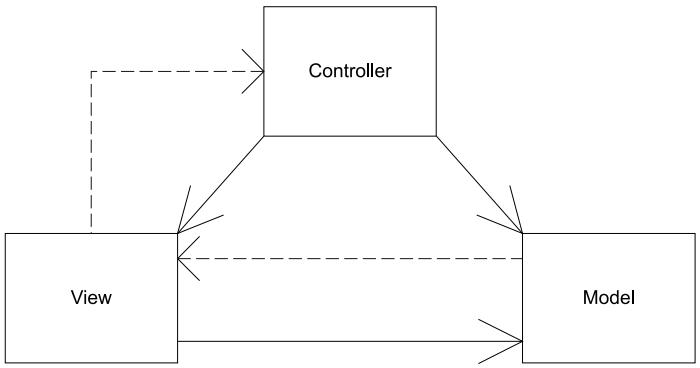


Figure 1.9 A diagram of the Model-View-Controller design pattern

[8] "Man is condemned to be free; because once thrown into the world, he is responsible for everything he does." —Jean-Paul Sartre

Griffon gains a lot of momentum by using this design pattern. It is, after all, a widely used design pattern in many frameworks, and plenty of developers use it as a guiding principle for keeping components connected. This means you may be able to pick up the pace quickly because you likely already know the core concepts.

You'll find the MVC pattern deeply ingrained in Griffon's design. At the top level, it helps in arranging artifacts by type and responsibilities. At a lower level, you'll encounter it in the UI components. Swing relies heavily on this pattern, although many times the view and controller responsibilities are found in the same class (`JTable`), whereas the model is handled by a separate class (`TableModel`).

NOTE In chapter 3, we'll discuss Griffon's usage of MVC in depth.

The members of an MVC group are clearly defined in a Griffon application. Their roles are well defined, but in order to make the most of them, you must be able to configure their relationships and some of their properties. This brings us to the next section: convention over configuration.

1.4.2 The convention-over-configuration paradigm

Configuration is one of the key aspects of any framework; finding the correct amount of configuration is a tricky task. It's well known that you can't please everybody, so compromises must be made and boundaries should be defined. For example, let's revisit Struts. Struts is a popular web application development framework that came with a high price: over-configuration. Every property of every component had to be defined in a configuration file. For a small application, that wasn't too bad; but for full enterprise applications, it meant a lot of work. And let's not go into what a nightmare it was to maintain such applications.

Imagine for a moment that you have a bookstore domain model. You'll most likely encounter `Book`, `Author`, and `Publisher` domain objects. Following the MVC design pattern, you'll have a controller for each domain object: say, `BookController`, `AuthorController`, and `PublisherController`. You'll also need at least one view for each domain object; and if your application provides CRUD-like operations in your domain, it's likely you'll want three different views for each one, such as `BookListView` (lists all books), `BookShowView` (shows one book at a time), and `BookEditView` (lets you update a book's properties). Figure 1.10 illustrates this model.

Setting up this basic application would require a lot of configuration on the Struts config file. Now imagine having a domain consisting of dozens of domain objects. Not a happy picture.

Did you notice what we just did? We assumed that the controllers have a particular suffix, perhaps to easily identify them when browsing the application's source code; the views also follow a naming convention. If you take the naming convention a step further, you can organize those components by responsibility in well-defined file folders, such as models, views, and controllers. If each component follows these simple rules, the previously required heavy configuration is no longer needed; bootstrapping

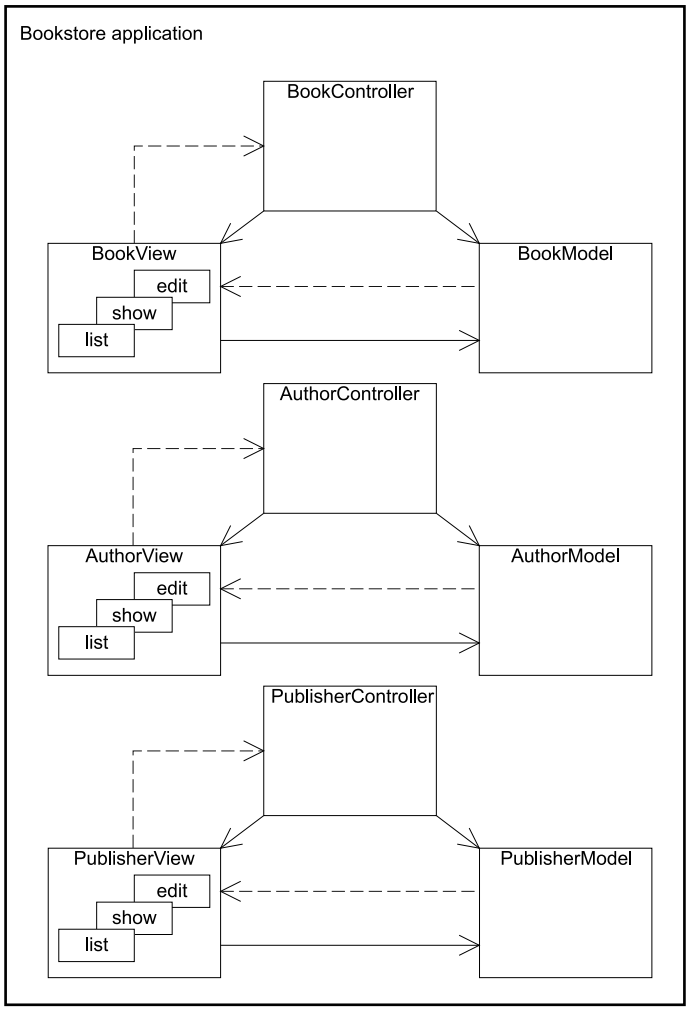


Figure 1.10 Bookstore application MVC model

code should be able to figure out how each component must be assembled given these conventions.

That is precisely the power of the convention-over-configuration paradigm (http://en.wikipedia.org/wiki/Convention_over_configuration). A developer should be required to configure a particular aspect of a component or a set of components only when that configuration deviates from the standard. This means that if you stick to a well-known set of conventions, you'll be able to create an application more quickly, because you'll spend less time configuring it; it's even possible to manage relationships between components this way as well.

By using the convention-over-configuration paradigm, Griffon can figure out a component's responsibilities, inspecting its name, it location in the directory structure, and perhaps some of its properties. Say goodbye to long and painful XML

configuration files; search no more for obscure configuration flags. All the configurable information you need to get your application off the ground is located close to the place where it's needed.

Griffon also manages an application's build cycle by providing a rich set of command-line scripts and bindings to the popular Ant project, all of which we'll discuss in chapter 2.

Being able to follow a predefined convention requires a rich environment where definitions can be created at the most convenient moment, even if that means at the last possible moment, at runtime. This calls for an environment that accepts a dynamic and highly adaptable solution, something the Java language can't provide but another language can—and that language is Groovy.

1.4.3 Groovy: a modern JVM language

Griffon relies on the power of Groovy to simplify development. Groovy complements and extends Java. Taking a closer look at how Groovy complements Java will help you understand how and why Griffon uses Groovy.

We mentioned before that the JVM is a great platform to develop with; for a time, the Java programming language was the only serious solution for getting things done. But time has caught up with Java. For many developers, it represents a conceptual cage without escape because the language changes slowly according to their needs.

Java was designed as a statically typed language, meaning that you must write as much type information as the compiler needs, even if that means repeating information that is obvious to you. Dynamic languages, on the other hand, require you to write less type information. Some languages are crazy enough to let go of all types![9] This in turn lets you deal with the particular task at hand, most of the time delaying type checks until runtime.

We could argue the *static versus dynamic* debate all day, and no one would be correct or happy in the end. The real problem is one of *essence versus ceremony*: how much do you have to write in order to fulfill the required task, aided by the compiler and runtime aspects of a particular language?

JAVA WITHOUT THE CEREMONY

Groovy is one of a particularly exciting batch of dynamic languages that run in the JVM. What makes it exciting is that it brings to the surface the essence of the Java language, while hiding the complexity and ceremony. But you can access that complexity and verbosity if needed.

> *Groovy is what Java would have looked like if designed in the 21st century.*
>
> —Scott Davis

Groovy was inspired by other popular languages, such as Smalltalk, Python, and Ruby; but it remains true to Java in its core. Java is in its DNA, after all.

[9] Madness! Where is the world heading?

FROM JAVA TO GROOVY

The main advantage of learning Groovy from a Java developer's perspective is that almost 98% of Java code is valid Groovy code. The syntax is so close to Java that you can, in many cases, rename your files from .java to .groovy, and both the Groovy compiler and the interpreter will be happy with them. This is of great use to you as Java developer, because you'll be able to learn Groovy at your own pace. Start with straight Java syntax, and then take baby steps to some of Groovy's features. As you become more confident, you'll add more features. Suddenly you'll realize that writing idiomatic Groovy code isn't that hard.

Because it's based on Java, Groovy interacts with any Java code you throw at it, be it a simple Java class, a Java library, or a Java-based framework.

> *Groovy is Java, Java is Groovy.*
>
> —Scott Davis

But other aspects of Groovy aren't found in Java.

CLOSURES AND METAPROGRAMMING FEATURES

If Groovy was a simple syntactic-sugar coating over Java, its usage wouldn't be compelling. Groovy brings modern programming features to Java as well. One clear example is closures, or anonymous functions as they're known in other languages. Closures have been the center of a heated and intense debate in the Java community—so intense that it could be categorized as a religious debate. Although people are still deciding the best approach for getting closures into Java, you can take advantage of Groovy's closures as soon as you pick it up; no need to wait for the debate's outcome and the next version of the JDK.

Other useful and powerful features found in Groovy are its metaprogramming capabilities. You can modify an object's behavior at any point, whether at compile time or at runtime. Yes, that's right: you can monkey-patch an object's behavior to bend it to your will. Of course, you must be careful: a great responsibility is bestowed on you when you harness the powers of metaprogramming. This reflects another remark made by Scott Davis when he paraphrased Erwin Schrödinger and his famous paradox (http://mng.bz/kM4S):

> *Groovy is Java, and Groovy is Not Java.*
>
> —Scott Davis

It seems to contradict Scott's first remark, but if you give it a little thought, both are true. We're sure that by the time you've finished reading this book, you'll see Groovy and Java in a different light: one of cooperation and synergy rather than adversity and hostility.

Griffon relies on Groovy in many ways. The most visible is in the view aspect of an application, as you'll see in the next chapter. It also relies heavily on Groovy's metaprogramming facilities to implement and solve the convention-over-configuration rules. This doesn't mean you have to become a Groovy expert just to be able to handle

the framework. As you'll soon find out, Griffon presents sound choices here and there without forcing you to take a single path from which there is no return.

If all of what we've discussed sounds too good to be true, rest assured that it's real. Griffon is able to accomplish this because it stands on the shoulders of giants that laid the path for some of its technical direction. But mostly, Griffon owes a lot of gratitude to Grails.

1.5 Summary

Your feet are now wet; you've seen the Griffon take off. Before you continue your journey, let's take a moment to remember what you've learned so far.

You started this chapter by setting up your development environment and using the `griffon create-app` command to create the GroovyEdit application. Next, you built a multitabbed text editor. You found that Griffon works in groups of code based on the Model-View-Controller paradigm. You implemented two MVC groups in this chapter; by splitting your code into groups of this kind, you organize it in a logical manner so that everything follows this well-known paradigm. This approach is discussed extensively in part 2 of the book.

The convention-over-configuration paradigm is applied in many places. You created all the GroovyEdit application files in specific folders and with a predetermined naming pattern. Maintaining an application of this kind is thereby immeasurably simplified.

Data binding between views and controllers comes naturally with Griffon. In the model, you define a set of values that you reuse throughout your MVC groups. The view components display the current status of the model, whereas the related controllers change them. No getters and setters need to be set between components, thanks to the `@Bindable` annotation and Groovy's short and concise syntax.

Next, you visited the jungle. The Java platform is a great place to develop desktop applications, but it's not without its fair share of traps and obstacles. Griffon avoids them by standing on the shoulders of giants: the Grails framework and its community, the Groovy language, well-known design patterns, and convention over configuration. Together they bring synergy and high productivity gains to the desktop development.

Finally, you took a brief tour of MVC and how Griffon's is different from web-based MVC. Using the convention-over-configuration paradigm makes the application structure predictable and easy to follow. And Groovy is the glue that holds it all together. The power and expressiveness of Groovy help you write concise, expressive, powerful code.

In the next chapter, we'll discuss the `griffon` command in further detail, plus the default build-time configuration options.

2 A closer look at Griffon

This chapter covers

- The structure of every Griffon application
- Conventional configuration
- Command-line utilities
- The application's life cycle

When starting the development of a brand-new desktop application, what are some of the typical questions that spring to mind?

- Where should the sources be placed?
- What about configuration files?
- Where should libraries and resources be placed, and how should they be managed?
- What about tests?

Isn't that a huge burden when starting a project? Now imagine working on an existing project. You might stare at the project structure and the source, trying to make sense out of it, perhaps looking for a common pattern that might help you.

When it comes to developing Java desktop applications, no specification describes what, when, and how things should be done. To make matters worse,

every company and, often, each project, follows its own approach, making it harder to switch from one development team to the next.

In this chapter, we'll look at how Griffon aims to bring order to this chaos. It does so by providing a basic structure that all Griffon applications follow to the letter. Each component has its place and purpose, making it easier for anyone to recognize its role in the application in the blink of an eye. That in turn facilitates application maintenance. We'll also explore Griffon's command-line tools, which make the job of building your application snappier. We'll round out this chapter with a look at the application life cycle.

Let's begin by examining how Griffon applications are structured.

2.1 A tour of the common application structure

If you remember the exercise from chapter 1, where you created a simple multitabbed file-viewer application, you'll recall that each member of an MVC group follows the naming and placing conventions applicable to its specific responsibilities. Filenames of all MVC group members have a particular suffix that unambiguously spells out what artifact they describe. Those files in turn are placed in directories that share the name with the corresponding suffix. As it turns out, there are more conventional directories to be found in a Griffon application.

Griffon's conventions are dictated by the common application structure shared by all applications. As you might already know, this structure was created when you invoked the `griffon create-app` command. Let's see what else is created by that command. Figure 2.1 shows the directory structure of the GroovyEdit application.

The core of the application resides in the griffon-app directory. There you can find the code for all models, views, and controllers, which together form the backbone of the application. You also find other useful artifacts there. Table 2.1 shows a breakdown of the contents of each directory.

This directory structure is easy to follow. Herein lies one of the strengths of the framework: convention over configuration. As you develop more and more Griffon applications, you'll come to appreciate that the applications all follow a common structure and layout. It makes moving between applications much easier. You don't have to take the time to refamiliarize yourself with the application to remember where the views, models, and controllers are located.

You may notice a particular directory named `griffon-app/conf`; as its name suggests, it's a place for configuration files. As you may recall from chapter 1, every Griffon application can be deployed in three modes: standalone, Web Start, and applet. The conf directory holds the files and configuration settings to both run and deploy applications in any of these deployment modes. We'll cover deployment modes in more detail in chapter 10 in the context of packaging options.

This directory is populated with a set of default configuration files when an application is created. Additional configuration files can be found here depending on which plugins you have installed. Yes, Griffon supports the notion of framework plugins. If a

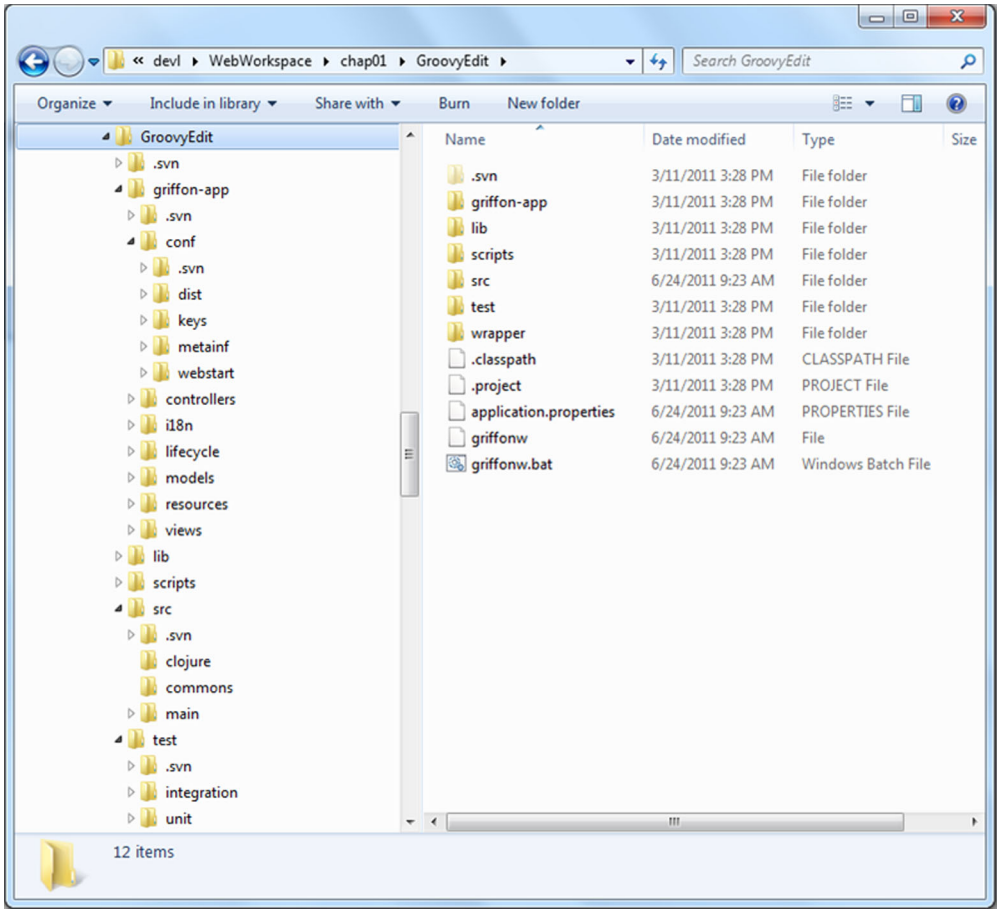


Figure 2.1 Directory structure of the GroovyEdit application

Table 2.1 Directory structure of a Griffon application. Each directory serves a particular purpose.

Directory	Description	Where to go for more info
griffon-app	Core Griffon artifacts	
+ conf	Configuration elements, such as Builder.groovy	Section 2.2
++ keys	Private keystore for Web Start/applet signing	Chapter 10
++ metainf	Files that should be included in the application's jar in the META-INF directory	
++ webstart	Web Start templates	Chapter 10
++ dist	Miscellaneous files used in packaging	Chapter 10
+ controllers	Controllers providing actions	Chapter 5

Table 2.1 Directory structure of a Griffon application. Each directory serves a particular purpose. *(continued)*

Directory	Description	Where to go for more info
+ i18n	Internationalization message bundles	
+ lifecycle	Application scripts to handle the life cycle	Section 2.4
+ models	Models	Chapter 3
+ resources	Application resources, such as images	
+ views	SwingBuilder DSL scripts	Chapter 4
lib	Jar archives	
scripts	Build-time scripts	Chapter 8
src	Other sources	
+ main	Other Groovy and Java sources	
test	Test sources	Chapter 9
+ integration	Application-wide tests	
+ unit	Component-specific tests	

particular feature isn't found in the framework itself, chances are that a plugin provides it. You'll learn more about plugins in chapter 11.

Let's look at the basics of configuring an application, which will be helpful when you need to tweak some of its settings.

2.2 The ABCs of configuration

When faced with the task of configuring a Griffon application, just think of it as being as easy as A-B-C, short for Application.groovy, Builder.groovy, and Config.groovy (see table 2.2). You can bet those names weren't chosen lightly.

Table 2.2 A-B-C summary

Configuration Area	Script name	Purpose
Application	Application.groovy	Runtime configuration of MVC groups
Builder	Builder.groovy	Defines and constructs the application
Config	Config.groovy, BuildConfig.groovy	All other runtime configuration Package and deployment configuration

Groovy scripts are used as configuration files instead of XML or any other markup language such as YAML or JSON. These scripts are the Groovy version of Java properties files.

To get a better understanding of Griffon and how to configure it, let's take a peek in each configuration file of the sample application (GroovyEdit) you worked with in chapter 1.

2.2.1 A is for Application

Your first stop in GroovyEdit's configuration files is `Application.groovy`. The following listing shows its contents as you left them in chapter 1.

Listing 2.1 Contents of griffon-app/conf/Application.groovy

```
application {
    title = 'GroovyEdit'
    startupGroups = ['groovyEdit']
    // Should Griffon exit when no Griffon created frames are showing?
    autoShutdown = true
    // If you want some non-standard application class, apply it here
    //frameClass = javax.swing.JXFrame'
}
mvcGroups {
    // MVC Group for "filePanel"
    'filePanel' {
        model      = 'groovyedit.FilePanelModel'
        view       = 'groovyedit.FilePanelView'
        controller = 'groovyedit.FilePanelController'
    }
    // MVC Group for "GroovyEdit"
    'groovyEdit' {
        model      = 'groovyedit.GroovyEditModel'
        view       = 'groovyedit.GroovyEditView'
        controller = 'groovyedit.GroovyEditController'
    }
}
```

1 Initialize specified MVC group

2 Declare MVC groups

As noted, Groovy configuration scripts are like Java properties files in disguise. The advantages are clear: you can visualize hierarchies and categories more easily, and you're able to use more types other than plain strings. For example, `application.startupGroups` is a list of values, and `application.autoShutdown` is a boolean value. Groovy can parse these scripts by means of `groovy.util.ConfigSlurper` and `groovy.util.ConfigObject`.

This script contains two top-level nodes: `application` and `mvcGroups`. The `application` node is responsible for holding the most basic information about your Griffon application, such as its title.

INITIALIZING THE MVC GROUP

Looking closely at the value of the `application.startupGroups` 1 configuration option, you'll observe two things: it's a list containing a single element, and the element is the name of one of the MVC groups you created for that application.

This configuration option tells the Griffon runtime which MVC groups should be initialized when the application is bootstrapping itself. Recall from chapter 1 that you had to explicitly initialize an instance of an MVC group of type `FilePanel`, but you didn't need to do so for the GroovyEdit group. Now you know why!

TERMINATING THE APP

The next configuration option, `application.autoShutdown`, controls whether the application will terminate when all frames and windows created by Griffon directly—that is, those created using the `application` node in a view script—are closed. As you'll soon find out, you aren't constrained to using the `application` node on a view script; other nodes let you build additional windows and dialogs. These nodes don't count as managed by Griffon for purposes of `autoShutdown`.

SETTING THE MAIN WINDOW CLASS

You already know that an application's main window is determined by its runtime/deployment target (the standalone and Web Start modes use a `JFrame`, whereas the applet mode uses `JApplet`). But if you'd like to switch to a different class, you need to provide a value for `application.frameClass`. The script suggests `javax.swing.JFrame`; certainly any other subclass of `javax.swing.JFrame` will be gladly accepted by Griffon.

CONFIGURING MVC GROUPS

The second top-level node, `mvcGroups` ❷, lists all MVC groups configured in your application. Typically an MVC group configuration defines the full qualified class names of each of its members. Although the default settings follow a naming convention, you aren't forced to follow it to the letter. But we're getting ahead of ourselves.

> **Can't wait to dig in to MVC groups?**
> If you want to know more about MVC groups right now, you're more than welcome to jump to chapter 6. Remember to come back, though, because there's more to learn about configuring your application, such as views and their builders.

The next file we'll review is Builder.groovy. This script holds the configuration of one of Griffon's key components: the `CompositeBuilder`.

2.2.2 B is for Builder

Remember the controller and view scripts in the FilePanel MVC group in chapter 1 (listings 1.7 and 1.8, FilePanelController.groovy and FilePanelView.groovy)? Those scripts rely on a Groovy feature called *builders*, which make creating hierarchical structures, such as Swing views, a breeze. Builders expose a series of nodes and methods that, when used according to their build rules, produce the expected results. Builders help you write code in a more expressive manner. In particular, `CompositeBuilder` is based on `FactoryBuilderSupport` (http://groovy.codehaus.org/FactoryBuilderSupport). This is important, because it enables `CompositeBuilder` to mix and match builders based on `FactoryBuilderSupport` as well. Clever extension hooks are also provided, as we'll discuss in chapter 12.

You may have noticed that whereas the `FilePanelView` file holds a Groovy script, the `FilePanelController` file holds a class definition. The reason behind this choice is that Views as scripts can be seen as a declarative approach to defining the visuals of

A word about builders

Many developers consider builders to be an eye-opening Groovy feature. Once you get to know them, you'll start to recognize patterns in your code where a builder is better suited to solve the problem at hand. Groovy offers a few choices for creating your own builders, and it packs many builders in its core distribution.

an application. Even so, in listing 1.7 the controller is able to tap into "magical" methods that handle threading in a simple manner; the controller class doesn't implement a particular contract or extend a certain class that provides those methods. Something else must be at work.

The answer to this conundrum is the `CompositeBuilder` and its settings. The following listing shows `Builder.groovy` as configured in the example application.

Listing 2.2 Contents of griffon-app/conf/Builder.groovy

```
root {
    'groovy.swing.SwingBuilder' {
        controller = ['Threading']
        view = '*'
    }
}
```

The `CompositeBuilder`, as its name implies, is a builder that mixes and matches other builders. This feature is the key to adding functionality to your views and controllers via additional nodes and will be examined in chapter 12. For now, let's concentrate on what the default configuration means to you as you're getting started.

Every Griffon application has an instance of `CompositeBuilder`. It's used as the provider of building blocks for views and controllers. You can think of a view script as providing the application structure, while Groovy is the mortar that holds everything together seamlessly. The default configuration sets up the basic information needed for a run-of-the mill Swing application to get up and running. Figure 2.2 illustrates the composite nature of the `CompositeBuilder`.

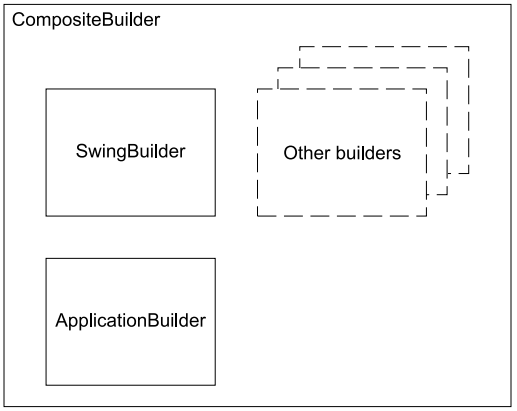


Figure 2.2 `CompositeBuilder` works with all builders based on `FactoryBuilderSupport`.

In listing 2.2, the top-level node, `root`, marks the default namespace used by the builder, which means that node names can be used as is. When an additional namespace is present, its value is used as a prefix for the node names it contributes. As a result, you can mix and match builders even if they contain colliding node names.

`SwingBuilder` is responsible for exposing all the nodes related to Swing components found in the JDK (and some other useful methods you'll discover along the way).

Notice that `SwingBuilder` has `view = '*'` in its settings. This means all the nodes and methods they provide will be contributed to all view scripts. Notice also that only `SwingBuilder` defines `controller = ['Threading']`. This translates into all nodes related to threading being exposed to controllers. That is the missing link in how controllers are able to use threading facilities without resorting to inheritance: the `CompositeBuilder` injects threading-related nodes and methods into controllers with that particular setting.

If all this builder and nodes talk is too much for you, don't worry! Its impact will come to you slowly as you progress with the book's examples. Just remember that if you ever need to adjust a builder's settings or expose additional nodes to a controller, `Builder.groovy` is the script you should tweak.

We've covered A and B; now it's time for C. This one is full of goodies, too.

2.2.3 C is for Config

The last two configuration files we'll review are Config.groovy and BuildConfig.groovy:

- *Config.groovy*—This file contains runtime configuration. Hmmm. Hold on a second: we just described runtime configuration as the responsibility of the Application.groovy script. The `Application.groovy` script only holds runtime configuration pertaining to MVC groups; `Config.groovy` holds all other runtime configuration the application may need. In the end, as illustrated in figure 2.3, both files are merged into a single object in memory from which an application can read all the settings.
- *BuildConfig.groovy*—This file is responsible for build-time configuration. It holds the relevant information pertaining to each running environment, and it defines configuration options that become relevant when you're packaging and deploying an application. You'll learn more about those options in chapter 10.

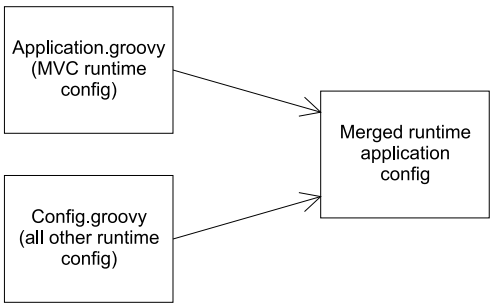


Figure 2.3 Merging the MVC runtime config and all other runtime config into the runtime application config

Let's continue with our exploration of runtime and build-time configuration by looking at some typical tasks and where the configuration is located.

CONFIGURING YOUR ENVIRONMENTS

Have you ever needed to keep separate configuration settings for your development and production code? If so you may have felt the pain of storing repeated information or, worse, having outdated or incompatible settings across these environments. Fortunately, Griffon makes this job easier by keeping all environment-related configuration at the same location.

Environments are a handy way to separate configuration settings depending on the current phase of the application you're dealing with. For example, you might want to skip signing jar files when prototyping your application, because the signing process takes a few additional seconds to finish. On the other hand, you'll want all your jars signed and verified before going into production, lest an ugly security error pop up in front of your target audience.

Griffon enables three environments by default:

- Development (dev)
- Test (test)
- Production (prod)

The default settings for this particular configuration section can be rather long. The following is a simplified version.

Listing 2.3 Environment settings on BuildConfig.groovy

```
environments {
    development {
                . . .      <-- Settings for dev environment
    }
    test {
                . . .      <-- Settings for test environment
    }
    production {
                . . .      <-- Settings for prod environment
    }
}

griffon {
    memory {
        //max = '64m'
        //min = '2m'
        //maxPermSize = '64m'
    }
    jars {
        sign = false
        pack = false
        destDir = "${basedir}/staging"
        jarName = "${appName}.jar"
    }
    webstart {
```

```
        codebase = "${new
     File(griffon.jars.destDir).toURI().toASCIIString()}"
        jnlp = 'application.jnlp'
    }
    applet {
        jnlp = 'applet.jnlp'
        html = 'applet.html'
    }
}
```

The `environments` top-level node is responsible for holding the settings for each environment, and you may define as much as you want for each one.

The `griffon` top-level node serves as a catch-all for any settings that weren't specified. This means that if your `BuildConfig.groovy` file looks like this short version, you'll never sign any jars no matter which environment you choose to run, because `griffon.jars.sign` is set to false and no other environment overrides that setting. If you're wondering where `griffon.jars.sign` is, you won't find it in the config file exactly like that. But you will find a `griffon` node, which contains a `jar` node, which contains a `sign` variable. This is precisely how Groovy configuration scripts enhance standard Java properties files.

You might be wondering how Griffon knows which environment it should use when running your application. The answer lies again in the rules of the convention-over-configuration paradigm. By default, Griffon assumes the development environment when you run an application using this command:

```
$ griffon run-app
```

Conversely, it uses the test environment when you test your application using this command:

```
$ griffon test-app
```

How do you specify which environment to use if you want to change the default? You tell the `griffon` command which environment you want. For example, the following command runs the application in standalone mode using the production environment:

```
$ griffon prod run-app
```

> **It works the same in Griffon as in Grails**
>
> The environment feature is another trait shared with Grails; there it's used to quickly switch between data sources. Seasoned Grails developers shouldn't have much trouble picking up how Griffon applications are configured. They're pretty much the same as their Grails cousins.

What if none of the default environments will work for your needs? In that case, you can define a custom environment with any name you want—for example, `special`.

You can instruct the `griffon` command that it should use that environment by adding a command flag like this:

```
$ griffon -Dgriffon.env=special run-app
```

`BuildConfig.groovy` has one last responsibility, related to additional configuration settings that other components might need.

CONFIGURING BUILD-RELATED SETTINGS

Every now and then, when you're building an application, you need to make a few tweaks to its build settings. Most of them are already exposed by the environments features we just talked about. In the event that you create a build script or a build event handler (explained in chapter 8) that requires a configurable choice, `BuildConfig.groovy` is a good place to put the configuration settings. For now, there's nothing much to see in the sample application, but rest assured that we'll visit this file again in chapter 8 when we discuss build events and in chapter 11 when we touch the subject of plugins and addons.

Now let's cover the settings that you can configure in `Config.groovy`.

CONFIGURING LOGGING

Given that logging is a runtime concern, you'll find its configuration in `Config.groovy`. If you look at the contents of the file (see the next listing), it starts with a single configuration node. Its name should give you a hint of what is being configured.

Listing 2.4 Contents of `Config.groovy` as created by default

```
log4j = {
    // Example of changing the log pattern for the default console
    // appender:
    appenders {
        console name: 'stdout', layout: pattern(conversionPattern: '%d [%t]
        ➥ %-5p %c - %m%n')
    }

    error  'org.codehaus.griffon'

    info   'griffon.util',
           'griffon.core',
           'griffon.swing',
           'griffon.app'
}
```

That's right: you can configure logging for a Griffon application by means of a logging DSL that works with Log4j. You can change two types of settings: the appenders to use and the logging level per package.

Appenders specify where messages are sent or displayed and how they're formatted. The default configuration specifies a pattern to be used with the `console` appender. This appender prints to the standard output any message that is logged. You can choose two additional appender presets: `file` and `event`. The former saves all messages into a particular file that may grow indefinitely, and the latter pushes application events for every matching logging call. (We cover application events in chapter 8.)

Logging *levels*, on the other hand, define the priority of a logging message. For example, in the default configuration, messages from the package `org.codehaus.griffon` (and its subpackages) will be sent to an appender only if their priority is `error` or higher. The following logging levels are available for configuration, sorted from most important to least: `fatal`, `error`, `warn`, `info`, `debug`, and `trace`. You can use two additional levels:

- `all`—Enables all levels for a particular package
- `off`—Disables all messages from a particular package

You configure a logging level by specifying the type (such as `error` or `info`) followed by the name of a package or a list of packages.

TIP To learn more about this logging framework and, in particular, the layout options for formatting messages, refer to Log4j's documentation.

Last, you can also set in Config.groovy any additional configuration flags that might be needed at runtime. No such flags are needed by default; that's why you don't see any in Config.groovy after the application has been created. But plugins (and addons) make extensive use of this file. We'll cover plugins and addons in chapter 11. You'll also see a good example of additional configuration flags in chapter 13, where we'll specify dynamic behavior for some artifacts.

This is all we'll say about configuring a Griffon application for now. The `griffon-app/conf` directory should be a familiar place: come back to this section if you have any doubts about how a Griffon application can be configured. We'll continue exploring Griffon's feature set by discussing command-line utilities.

2.3 Using Griffon's command line

You've already seen some of Griffon's commands when building the GroovyEdit sample application, such as `create-app` and `run-app`.

What's the difference between a command and a target?

A command is the entire line:

```
$ griffon create-app GroovyEdit
```

A target is the specific task you're asking `griffon` to do:

```
create-app
```

The Griffon command line is one more way Griffon makes your job easier. It encapsulates repetitive tasks into single commands to save you time typing. This section provides a detailed list of all the command targets available when you install the Griffon distribution.

Table 2.3 lists several of the targets you can call using the `griffon` command, gives a short description of each one, and also mentions in which chapter the command

Table 2.3 Available `griffon` targets, including a short description and where you can learn more

Target	Description	Where to go for more info
	Build targets	
`create-app`	Creates a brand new application	Section 2.3.1
`create-mvc`	Creates an MVC group and adds it to the configuration files	Chapter 6
`compile`	Compiles all sources (except tests)	Section 2.3.1
`package`	Compiles and packages the application	Section 2.3.1
`clean`	Deletes compiled classes and compile artifacts	Section 2.3.1
	Run targets	
`run-app`	Runs the application in standalone mode	Section 2.3.2
`run-applet`	Runs the application in applet mode	Section 2.3.2
`run-webstart`	Runs the application in Web Start mode	Section 2.3.2
`shell`	Runs an interactive Groovy shell with the application in the classpath	Section 2.3.2
`console`	Runs an interactive Groovy visual console with the application in the classpath	Section 2.3.2
	Miscellaneous targets	
`help`	Displays a list of available commands	Section 2.3.3
`stats`	Shows how many lines of code are in your application per artifact	Section 2.3.3
`set-version`	Sets the application version on the application's metadata file	Section 2.3.3
`upgrade`	Modifies your application to comply with a newer Griffon version	Section 2.3.3
`integrate`	Adds IDE-specific files to your application	Section 2.3.3
`create-script`	Creates a new Gant script	Chapter 8
`create-unit-test`	Creates a new unit test for any source artifact	Chapter 9
`create-integration-test`	Creates a new controller test	Chapter 9
`test-app`	Runs all application tests, both unit and integration	Chapter 9

Table 2.3 Available `griffon` targets, including a short description and where you can learn more *(continued)*

Target	Description	Where to go for more info
`set-proxy`	Sets proxy settings allowing buildtime tools access the network behind a firewall	Chapter 11
`create-plugin`	Creates a new plugin project	Chapter 11
`package-plugin`	Packages the plugin, making it ready to be installed	Chapter 11
`release-plugin`	Uploads a plugin to the plugin repository	Chapter 11
`list-plugins`	Provides a list of available plugins	Chapter 11
`plugin-info`	Displays information about a particular plugin	Chapter 11
`install-plugin`	Installs a plugin on an application	Chapter 11
`uninstall-plugin`	Uninstalls a plugin from an application	Chapter 11

will be covered. It's worth saying that all of these commands are implemented using a Groovy flavor of Ant (http://ant.apache.org) named Gant (http://gant.codehaus.org), which makes them highly customizable.

This is by no means a complete list of targets—future releases of Griffon might provide additional scripts. Also, an installed Griffon plugin may provide more command targets via scripts (as you'll see in chapter 11). You'll have the opportunity to provide your own commands as well, also using scripts.

> **What is Gant?**
> Gant is a Groovy-based tool that lets you script Ant tasks (see http://gant.codehaus.org). It uses the Groovy language as a domain-specific language (DSL) to describe a build manifest. You can think of it as a painless alternative to XML builds. Both Grails and Griffon take advantage of this powerful tool.

Let's review the command targets by category: build, run, and miscellaneous.

2.3.1 Build command targets

The build command targets allow you to create new artifacts, either source or compiled ones:

- `create-app`—Chiefly takes one argument, the name of the application to build, and then proceeds to create the entire application structure, configuration, and templates.
- The remaining targets—`compile`, `package`, and `clean`—are responsible for transforming the application sources.

- `compile`—Compiles all available source code into byte code. This includes all source code found under griffon-app (yes, the configuration files as well) and additional source code found under `src/main`.
- `package`—Assembles the application code into a jar, signs it along with additional jars found in the application's lib directory if jar signing is enabled, and compresses all jars for optimized download if packing is enabled. It also assembles the required files for Web Start and applet deployment. The default location for these artifacts is `$basedir/staging`, which is a setting you can modify by editing `BuildConfig.groovy`.
- `clean`—Deletes the working directory where all classes are compiled, along with the staging directory.

2.3.2 Run command targets

You may already be familiar with this group, because all `run-*` targets are found here, but this list provides a quick reminder:

- `run-app`—Runs the application in standalone mode—in other words, as if it were to be deployed as a regular desktop application on any platform.
- `run-applet`—Runs the application in applet mode. If the default configuration is in place, then application packaging signs and packs all jars.
- `run-webstart`—Runs the application in Web Start mode. Just as in applet mode, application packaging signs and packs all jars when configured to do so.
- `shell` and `console`—Run an interactive Groovy shell with the packaged application on the classpath, allowing you to run it for quick prototyping or API exploration. The difference between these targets is that `shell` is strict command line and `console` displays a visual tool. In case you're wondering, these targets are found in Grails, too.

2.3.3 Miscellaneous command targets

The last group of command targets we'll review in this chapter provide additional behavior and information. Although we don't cover the complete list, these command targets are important enough to mention:

- `help`—Displays the current Griffon version, sample usage of the `griffon` command, and a list of available commands. You saw the `help` command target in chapter 1; it verified that you had a working Griffon installation.
- `stats`—Also demonstrated in chapter 1; parses all of your application's sources and compiles a list of the number of lines of code per artifact type. This target helps you realize how much you can attain with so little code, thanks to the power of conventions.
- `set-version`—Updates the application's current version, found in the `application.properties` file. If you look back at figure 2.1, you'll see this file

located at the root of your application's directory. It's important to update this file using a command target because this file is also responsible for defining which version of Griffon the application is compatible with, and as such this file is an auto-generated one.

- `integrate-with`—Adds IDE support files. Command-line tools and a simple text editor are good enough to get you started, but if you need additional power, you might want to try a Java IDE that supports Groovy and Griffon. At the time of writing, you can specify the following command flags to integrate with a particular IDE or build tool: `eclipse`, `ant`, `gradle`, `textmate`, and `intellij`. For example, if you want to add Eclipse-specific support files, you invoke the `integrate` command as follows:

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

  You can now open your project from within Eclipse, because this command created a pair of files that every Eclipse project requires: .project and .classpath. You can find comprehensive coverage of setting your IDE to work with Griffon projects, as well as other productivity tools, in chapter 14.
- `upgrade`—Upgrades your application from one version of Griffon to the latest version installed. This target does its best to update all required files, but sometimes it needs help from external agents; this is where custom scripts and plugins play a part in the grand scheme.

There's one last topic to cover before we reach the end of this chapter, and it's important in terms of the facilities that must be provided by an application framework: the application life cycle.

2.4 Application life cycle overview

Before we jump into the plumbing of MVC groups, we should explain a key concern addressed by Griffon: the application's life cycle and its management. As discussed in chapter 1, each application should be responsible for bootstrapping itself, allocating resources, and configuring its components as they're being loaded, instantiated, and wired up. It should also be able to handle its shutdown graciously, liberating any allocated resources and performing any pending cleanup. If you're building a Java application by hand or using some other framework, it can be difficult to figure out where and how to implement that logic.

Again, Griffon shines and comes to your rescue. A Griffon application has a well-defined life cycle, with each phase neatly handled by a particular script. This life cycle provides easily identifiable places to put your application-specific life cycle logic.

Look at your application's structure in `griffon-app/lifecycle`, and you'll see several files, as listed in table 2.4. These scripts are responsible for handling each phase of the life cycle, and they're listed in the order in which the life cycle calls them.

Table 2.4 Life cycle phases and associated scripts

Life cycle phase	Script
Initialize	`Initialize.groovy`
Startup	`Startup.groovy`
Ready	`Ready.groovy`
Shutdown	`Shutdown.groovy`
Stop	`Stop.groovy`

CAUTION All life cycle scripts run in the event dispatch thread (EDT), and you have to be careful with the code you place in them. Any long-running computation will cause your application to appear unresponsive and sluggish.

Figure 2.4 shows the GroovyEdit application that you created in chapter 1.

All life cycle scripts run in the EDT. There are some alternatives for proper threading and handling operations in the EDT; feel free to jump to chapter 7 if you can't wait to learn about them.

The first stop in the chain of events handled by the life cycle is initialization.

2.4.1 Initialize

When you run GroovyEdit, initialization is the first life cycle phase that takes place (see figure 2.5). It's triggered after all configuration files have been initialized and read but just before any component is instantiated, meaning you can't access any MVC members yet.

But this is an excellent moment to tweak your application's look and feel, because not a single visual element has been constructed yet. You can do so either by following the standard Swing approach or using the provided utilities. The template for this script suggests a few tweaks, as shown in the following listing.

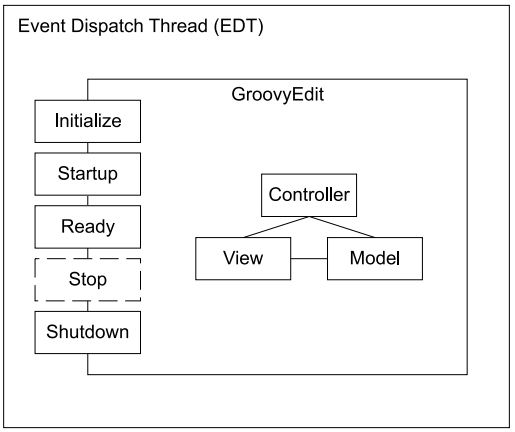


Figure 2.4 Application life cycle

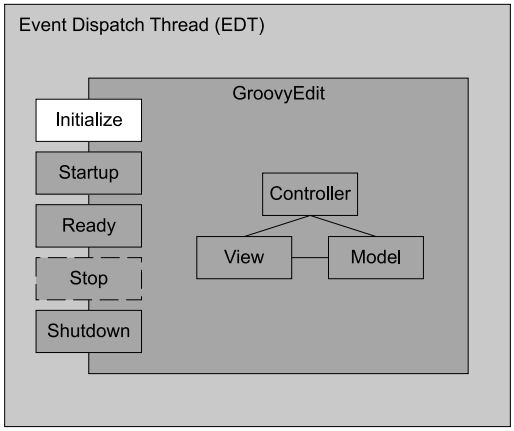


Figure 2.5 Application life cycle: initialize

Listing 2.5 Default contents of Initialize.groovy

```
import groovy.swing.SwingBuilder
import static griffon.util.GriffonApplicationUtils.isMacOSX
SwingBuilder.lookAndFeel((isMacOSX ? 'system' : 'nimbus'), 'gtk', ['metal',
➥[boldFonts: false]])
```

The code in this script configures the look and feel. It relies on a list of names that resolve to a particular `lookAndFeel` setting. For example, if you're running on JDK 6 or newer, it's most likely that Nimbus will be the chosen `lookAndFeel`; if it isn't available, then the next element in the list will be tried, up to the default provided by the Java platform, which is `Metal`.

This phase is also useful for performing sanity checks on resources and configuration settings, because you can abort or query the user for additional information. Just remember that none of the MVC groups are initialized at this moment, so any UI you display must be created manually.

The second stop in the chain is startup.

2.4.2 Startup

The script that handles the startup phase is called right after all startup MVC groups have been initialized (see figure 2.6). Remember the settings in `Application.groovy` (section 2.2.1)? One controls which MVC groups should be instantiated by default when the application runs: its name is `startupGroups`.

You're now able to reference any members of those MVC groups already initialized, and you can start any background work that requires those references.

Next in the chain is the ready phase.

2.4.3 Ready

At first, the ready phase may look superfluous. As with the startup phase, all startup MVC components have been initialized (see figure 2.7). The catch is that this phase

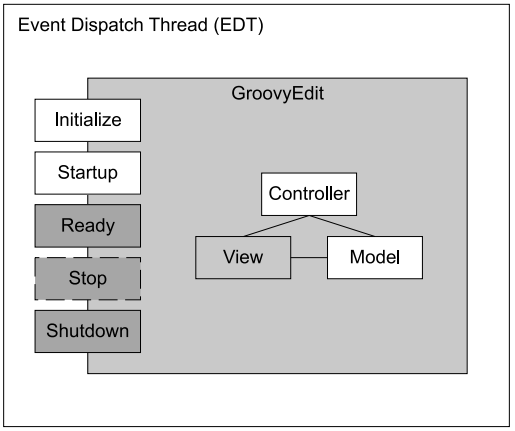


Figure 2.6 Application life cycle: startup

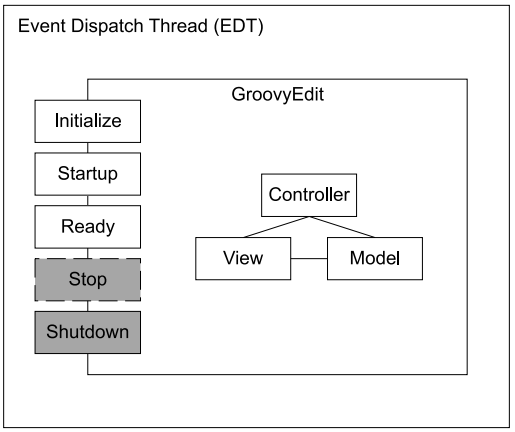


Figure 2.7 Application life cycle: ready

takes place after all events posted to the EDT queue have been processed. This means if any of the initialized startup view scripts or code run by the previous phase posted new events to the EDT queue, they should have been consumed by now. If you wanted to restore a previous editing session, this would be a good place to do it.

The main window of the GroovyEdit application is shown after this phase finishes. The application has been fully initialized at this point; no further life cycle scripts will be called until the application is ready to shut down in the next life cycle phase.

2.4.4 Shutdown

This is the last life cycle script called by Griffon. In the GroovyEdit application, this phase can be triggered by choosing File > Quit or via a `Window Close` event.

The shutdown phase represents the last chance for freeing used resources, saving configuration settings, and closing any connections your application might have opened. If you wanted to save a list of currently open files in GroovyEdit, this is where you would do it.

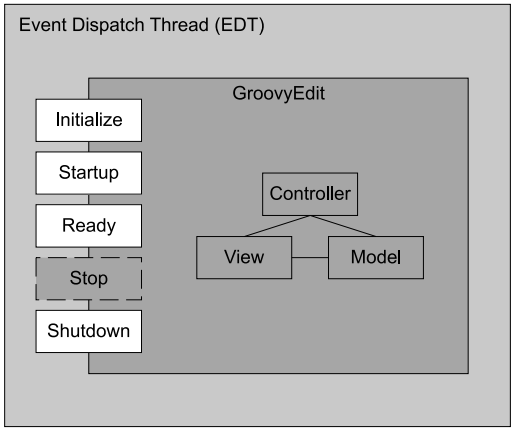


Figure 2.8 Application life cycle: shutdown

Once this phase has been executed, there is no turning back: your application will exit.

2.4.5 Stop

We know we said shutdown is the last life cycle phase of an application, but that's only partly true. Due to the three-way deployment targets supported by Griffon, this phase comes in handy when you're deploying to applet mode. Applets behave a bit differently than regular desktop applications or Web Start enabled ones. An applet is supposed to run whenever the page that contains it is displayed in a browser. So although startup works as expected with an applet, stop is a special life cycle phase for dealing with browsers.

An applet will sit there happily doing its job until one of the following happens: you close the browser window or tab, or you move away from the page to another.

In the former case, the application will cease to exist, in which case the shutdown phase is called. In the latter case, the applet will be stopped; if you want to

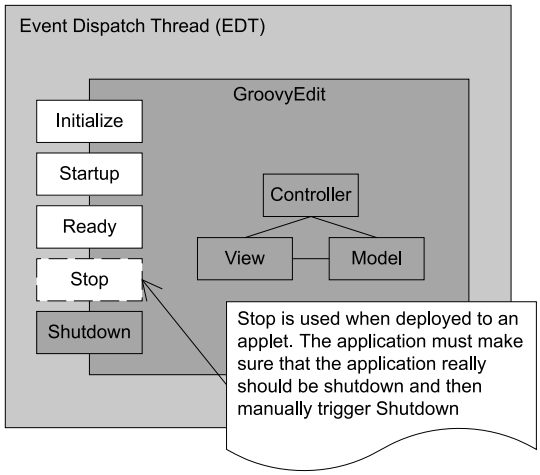


Figure 2.9 Application life cycle: stop

navigate back to the page where it's found, the stop phase is called and its handler script executed.

There you have it: the application's life cycle in a nutshell. It's simple, but it's a powerful feature in Griffon's arsenal against painful desktop application development. Instead of figuring out when and how each of these tasks should be invoked, you let Griffon call the phase-handler scripts when needed.

2.5 Summary

Phew! That was quite an exploration of a Griffon application's structure. To recap, every Griffon application shares the same directory structure according to the framework's conventions. This increases your productivity because every artifact has its place and name according to its role. Once you learn the conventions, finding your way around any other Griffon application should be a walk in the park.

You learned that using Griffon is as easy as ABC: `Application.groovy`, `Builder.groovy`, `Config.groovy`, and `BuildConfig.groovy`. You also learned about the basic command targets provided by the `griffon` command. These command targets take advantage of the framework's conventions and your application's configuration to do their job. Additional command targets will be covered in later chapters of the book.

Finally we touched on the subject of application life cycle management. It's the framework's responsibility to figure out the correct stage when an application is run. This relieves you of keeping tabs on how to do this kind of management, leaving you the task of deciding what to do when a particular stage is triggered.

This completes part 1 of the book. Next, we'll spend some time with models and learn how to automatically transfer values from views to models using bindings.

Part 2

Essential Griffon

Things get interesting in this part of the book. Here we'll cover the three MVC members that function as the cornerstone of every Griffon application. You'll learn about the responsibilities of models, views, controllers, and services. You'll find a thorough discussion on binding and observable events. By the end of part 2, you'll be confident in writing basic applications without a hitch.

3 Models and binding

This chapter covers:

- Creating models
- Creating observable properties
- Binding data from the model to the view

Any sufficiently advanced technology is indistinguishable from magic, at least according to Arthur C. Clarke. And getting data to automatically update itself in several places with a few simple declarations looks magical at first, but when you learn to identify the process that's occurring it seems more mechanical than magical.

Why are models used in the MVC framework, and what is their role? Models are to some extent a shared whiteboard, where a controller or a view can update abstract values and respond when it observes a change. The key in these instances is the binding of the data, and to the uninitiated the bindings can appear magical.

To understand and appreciate just how magical the binding support is in Griffon, it's necessary to go down to the lowest levels of code and see how you would construct these patterns, and then show how the bind calls build on them. It's like long division: you can teach a child to use a calculator, but if they know how the arithmetic works it seems a lot less magical and is conceptually easier to grasp.

But as with all magic, there are some dues to be paid. Before we dig into how it works, you need to understand why you need to make it work.

To give you some quick exposure to models and bindings, we'll start by looking at a form application with some bindings. If you've ever had to do binding manually, this will excite you. Once you have some appreciation for models and bindings, the next step is seeing how the model acts as a communication hub for the application. Then, you'll learn to make changes to the model observable to the application and how an application can automagically respond to changes in the model. Sometimes you need a little more control, so we'll look at controlling when an application responds to model changes. To bring it all together, you'll build a mortgage calculator.

3.1 A quick look at models and bindings

The goal of this section is to help you gain an appreciation of the importance and power of models and bindings. This preview will help you focus on the rest of the chapter.

You'll begin by building a simple application: a registration form. For this application, you need the user's name, last name, and address. Of course, you need a way to submit the information: a Submit button. Let's also assume that you need a way of clearing all the information: a Reset button. From a requirements perspective, you've been told that the Submit button shouldn't be available until the user has entered all the information, and the Reset button should be available once the user has entered any information. Most people are visually oriented, so the application should look something like figure 3.1.

Now that you know the functional requirements of the application and how it should look, you can get started building it.

3.1.1 Creating the project

If you've been working on the previous examples in the book, you know what's coming. First you create a project. Open a shell prompt into the directory in which you want to create the Griffon application, and create a new `form` application:

```
$ griffon create-app form
```

As you've come to expect, Griffon sets up the project directory. Now let's turn our attention to creating the model, view, and controller.

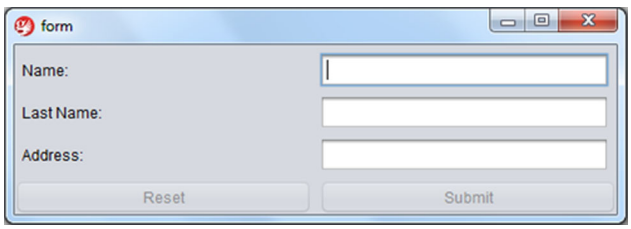


Figure 3.1 The completed registration application

3.1.2 Creating the model

Looking at figure 3.1, it's easy to see that the model will need to have the following properties: `name`, `lastName`, and `address`. But you also have requirements for enabling and disabling the Reset and Submit buttons. To accomplish that requirement, the model needs a couple of additional properties: `submitEnabled` and `resetEnabled`. You'll use these two properties to determine when the buttons should be enabled. The following listing shows how the model should look; please make sure to copy and paste the contents into your Model file.

Listing 3.1 FormModel.groovy

```
package form

import groovy.beans.Bindable

@Bindable
class FormModel {
  String name
  String lastName
  String address

  boolean submitEnabled
  boolean resetEnabled
}
```

In chapter 1, the `@Bindable` annotation was on the individual fields. What's up here? By moving the `@Bindable` up to the class level, all the fields become observable properties with `PropertyChangeSupport`. This is another way that Griffon makes our lives better.

That takes care of the information needs, but we're not done with the model just yet; now let's focus on the rules, as well as enabling and disabling the buttons. The buttons will use the value of the `submitEnabled` and `resetEnabled` properties to determine if the buttons should be enabled or disabled. How do you deal with the rules and logic of setting the values of the properties? You can create a helper closure (`enabler`) that contains the rules and logic. The next listing shows the new helper closure.

Listing 3.2 FormModel.groovy with enabler logic

```
package form

import groovy.beans.Bindable
import griffon.transform.PropertyListener
import static griffon.util.GriffonNameUtils.isBlank

@Bindable
@PropertyListener(enabler)
class FormModel {
  String name
  String lastName
  String address

  boolean submitEnabled
  boolean resetEnabled
```

Helper utility

1 PropertyListener definition

```
  private enabler = { e ->                          <-- ❷ Enable/Disable logic
    submitEnabled = !isBlank(name) &&
                    !isBlank(lastName) &&
                    !isBlank(address)

    resetEnabled = !isBlank(name) ||
                   !isBlank(lastName) ||
                   !isBlank(address)
  }
}
```

The code contains two additional imports: a `@PropertyListener` annotation and the `enabler` closure. The imports are straightforward and don't need any additional explanation. The `enabler` closure ❷ takes an event as input. The closure uses the `isBlank` helper to determine the value of `submitEnabled` and `resetEnabled`. Why use `isBlank` in the first place? For one thing, it's a utility method provided by the Griffon runtime, which means it's readily available at any time. Second, this utility is written in such a way that it doesn't require any external dependencies or additional libraries other than the Griffon runtime itself.

The enabler `closure` is in place, but how is it invoked? This is where the `@PropertyListener` annotation ❶ comes into play. The `@Bindable` annotation adds `PropertyChangeSupport` to all fields of the class. This means that when any of the field values change, a property change event is fired. The `@PropertyListener` is set up at the class level to listen to all property-change events associated with the class. When an event is fired, the `enabler` closure is invoked with the event. That's pretty slick. You can also apply `@PropertyListener` locally to a property; this will have the same effect as registering a `PropertyChangeListener` that handles change events for that property alone. One thing to keep in mind when you're using `@PropertyListener` in combination with closures like `enabler` is that it's highly recommended to apply the `private` visibility modifier to them. This keeps the code from bleeding out to other classes. You may recall that everything in Groovy is `public` unless declared otherwise; the inner workings of such closures don't concern classes other than the one that holds their definition.

With the model out of the way, it's time to focus on the rest of the application. Let's move on to the view to see how it uses the model's properties.

3.1.3 Creating the view

Now it's time to create the user interface. Take a quick look at figure 3.1: you need three `labels`, three `text fields`, and two `buttons`. The following listing contains the view code.

Listing 3.3 FormView.groovy

```
package form

application(title: 'form', preferredSize: [320, 240],
  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: emptyBorder(6)) {
    gridLayout(rows:3, columns:2, hgap:6, vgap:6)
    label 'Name:'
    textField columns:20,
      text: bind(target: model, 'name', mutual: true)
    label 'Last Name:'
    textField columns:20,
      text: bind(target: model, 'lastName', mutual: true)
    label 'Address:'
    textField columns:20,
      text: bind(target: model, 'address', mutual: true)
  }
  panel(constraints: SOUTH) {
    gridLayout(rows:1, cols: 2, hgap:6, vgap:6)
    button('Reset', actionPerformed: controller.reset,
           enabled: bind{ model.resetEnabled })
    button('Submit', actionPerformed: controller.submit,
           enabled: bind{ model.submitEnabled })
  }
}
```

❶ Bindings

You've seen view code before, and chapter 4 will go into more details, so we won't go into the view too deeply here. The view is set up as two panels. The first panel holds the `name`, `lastName`, and `address` `textFields` and their associated `labels`. The second panel holds the Reset and Submit buttons.

The important thing to focus on right now is the bindings ❶. The *binding* is the mechanism that associates the view component with the model property. The `textField` bindings contain an extra parameter, `mutual`. Setting `mutual` to `true` creates a bidirectional binding. When the view is changed, the model is updated, and when the model is changed, the view is updated. You also see bindings on the buttons. Here the binding determines whether the button is enabled or disabled. When the user clicks a button, the associated `actionPerformed` is invoked.

As you can see, the buttons invoke methods on the controller. Let's look at the controller next.

3.1.4 Creating the controller

The controller is responsible for orchestrating the application. In this simple application, there isn't much to do but reset the model values when the Reset button is clicked and submit the model values when the Submit button is clicked. For Submit, you'll print the model values. The next listing contains the code you need.

Listing 3.4 `FormController.groovy`

```
package form

import griffon.transform.Threading
```

```
class FormController {
    def model
    def view

    @Threading(Threading.Policy.SKIP)
    def reset = {
        model.name = ''
        model.lastName = ''
        model.address = ''
    }

    def submit = {
        println "Name: ${model.name}"
        println "Last Name: ${model.lastName}"
        println "Address: ${model.address}"
    }
}
```

Threading directive

Threading directives

Don't worry too much about threading right now. Chapter 7 is devoted to multithreaded applications, and it will fill in the blanks. In this case, `@Threading(Threading.Policy.SKIP)` causes the `reset` method to be executed on the same thread that called the action: the event dispatch thread (EDT), because the action will be linked to a button.

The controller is pretty straightforward—and now the application is complete. Go ahead and run it. It's amazing how much functionality you can enable with so little code.

Models and bindings are important parts of a Griffon application. Although we would love to take the credit for coming up with the idea to use models this way, Griffon stands on the shoulders of giants. In the next section, we'll take a quick tour of the history of models.

3.2 Models as communication hubs

As we discussed in chapter 1, the Model-View-Controller pattern has gone through a metamorphosis since its introduction in the late 1970s. Most relevant is the role of the model portion of the triad and how it's used in Griffon. Models in Griffon's MVC groups don't fill exactly the same role they did decades ago; but after a long, strange trip, the role of the model matches the original more closely than most models do in typical web stacks.

As originally implemented in Smalltalk-80 (http://en.wikipedia.org/wiki/Smalltalk), the model portion of the triad came in two different types (although the names weren't codified until later releases):[1]

- Domain model
- Application model

[1] http://c2.com/cgi/wiki?ModelModelViewController.

The distinction between the two is mostly in what the type models. The domain model is supposed to be ignorant of the UI and to serve the data needs of the particular problem domain being modeled.

The application model is fully cognizant of the UI. Its responsibility is to serve as an adapter between the domain model and the view by holding references to domain models using fields, properties, or collections. Many support classes in Smalltalk, such as `ApplicationModel` and `ValueModel`, supported the application model in this role. Griffon's model aligns with Smalltalk's usage of `ApplicationModel` and `ValueModel`. This is a bit of a departure from how Grails views models.

Let's continue by looking at how web frameworks view MVC and then contrast that with Griffon's usage of MVC.

3.2.1 MVC in the age of web frameworks

When web frameworks started using the MVC pattern, the role of the model was almost exclusively served by the domain model. The duties of the application model were split between the controller and the view. This was possible because the interaction model of a web application changed from a triangle to being more like a layer cake. This mirrors the classic three-tier web architecture that separated the database (model) from the web browser (view), with an application server (controller) to run the show.

This paradigm shift brought a renewed emphasis on the domain model and encouraged business rules that dictate correctness of data and the interactions themselves in domain-model objects—in other words, domain models became the most important component that everything else revolved around.

Web frameworks also are hampered by their connection between the model and the view because of this layer-cake structure: any connection between the model and the view must pass through the controller. There's no way a model can notify a view that there's new data available to be consumed without the controller taking an active role in the matter. They're also usually hindered by the fact that the view must initiate all activity and can't react to changes in the model (although there are libraries and frameworks to address this specific problem). Because of this, whenever you hear a web framework calling itself an MVC framework, its model is almost always of the domain model variety.

All this work in domain models on the web has resulted in some great object-relational mapping (ORM) libraries and database façade libraries that interact with plain old Java objects (POJOs). This effort isn't lost to a Griffon application, because all the objects needed to access these domain models can be placed directly in the model class and accessed by the view and the controller. These database-driven domain models, however, lie outside the scope of this book.

3.2.2 Rethinking the pattern

Looking beyond the domain model, Griffon allows for the application model to make a comeback. Unlike a web application, all the portions of the model, view, and controller exist in the same JVM. The proximity of the view to the model allows for the change in the model to precipitate updates in the view, bypassing the controller class. The view objects also have a direct reference to the model objects.

One of the differences between Smalltalk and Griffon is how the role of the application model is performed. Smalltalk placed a great emphasis on using objects as a reification of the variables in the model. Griffon places a greater emphasis on the declarations of the relationship. Griffon does generate objects in the background to manage the duties of the application model, but the developer doesn't need to interact with these unless they choose explicitly to do so.

In that sense, the model in a Griffon application becomes a communication hub: a place to store data and have other pieces of the equation react to it. A network service may retrieve a new instant message and store that in the model object. The relevant view can see this data loaded, automatically animate a globe to spin to the appropriate area, and post a floating text box with the new message from a user on the other side of the world. Once they're set up, these interactions are easy to declare.

Notification of these changes is the crux of the model. If a tree falls in a forest and no one is there to see it, then it doesn't matter if it makes a sound, because nobody will react to the event. In order to react to a change, you must be able to observe it. Observing the change in your beans is the next step.

3.3 Observable beans

Observable changes are one of the cornerstones of making binding work. Why are observing and being observable important? Because if you can't see something, you can't react to it. In programming, the act of observing a change can be difficult unless you take certain preparatory steps to ensure that efficient observation occurs.

One of the most accessible ways of observing a change is to look. But that can be surprisingly expensive. You have to look all the time, because you aren't sure when something will change, what is going to change, or even if anything can change. You could look now and then look again later. How do you know if something has changed unless you mentally noted it? One alternative is to react to the new look as if it were a change, but that leads to a lot of wasted effort when you're observing something that rarely changes.

The solution in these instances is to provide cues as to what can change, and provide facilities to track those changes. It's kind of like a stage magician: good magicians make it clear what they want the audience to follow. Sometimes it's a flashy wand, or attractive assistants, or the stage lights. But the effect is the same: if you follow the cues, then when something changes you're directly drawn to that change. The patterns used in observable beans share at least one aspect with stage magic: sometimes the change you're observing isn't what really happened, but it's the change the magician

wants you to see. The rabbit that was placed in the collapsing box may not be the same rabbit that was pulled out of the hat, but it's meant to look like it is. Similarly, the 10-digit phone number posted to the model may not be the same 7 digits entered into the text field: an area code may have been added.

JavaBeans provide a mechanism to point out where notable events may change, and they even provide a mechanism for the user of an object to be notified when the value of a property changes.

3.3.1 JavaBeans bound properties: the Java way

The upside of using JavaBeans is that it's a well-established pattern with clear meaning and a standard way to provide access. The downside is the large quantity of boilerplate text that goes in to creating an observable property. Consider the following example, which creates a model with two properties: `stringProperty` and `longProperty`.

Listing 3.5 JavaBeans bound properties in Java, the long way

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class MyModel {
  private final PropertyChangeSupport pcs =
    new PropertyChangeSupport(this);

  public void addPropertyChangeListener
  ➥(PropertyChangeListener listener) {
      this.pcs.addPropertyChangeListener(listener);
  }

  public void removePropertyChangeListener
  ➥(PropertyChangeListener listener ) {
      this.pcs.removePropertyChangeListener(listener);
  }

  public PropertyChangeListener[]
  ➥getPropertyChangeListeners() {
      return pcs.getPropertyChangeListeners()
    }
  }

  public void addPropertyChangeListener
  ➥(String propertyName,
  ➥PropertyChangeListener listener) {
      this.pcs.addPropertyChangeListener(
      ➥propertyName, listener);
  }

  public void removePropertyChangeListener
  ➥(String propertyName
  ➥PropertyChangeListener listener ) {
      this.pcs.removePropertyChangeListener(
      ➥propertyName, listener);
  }
```

1 Listener support

2 Listeners for all properties

3 Listeners for specific properties

```
  public PropertyChangeListener[]
  ➥getPropertyChangeListeners(String propertyName) {
      return pcs.getPropertyChangeListeners(
      ➥propertyName);
    }
  }

  private String stringProperty = "";

  public String getStringProperty() {
    return stringProperty;
  }

  public void setStringProperty(String newValue) {
    String oldValue = stringProperty;
    stringProperty = newValue;
    pcs.firePropertyChange("stringProperty",
    ➥oldValue, newValue);
  }

  private long longProperty = "";

  public long getLongProperty() {
    return longProperty;
  }

  public void setLongProperty(long newValue) {
    long oldValue = longProperty;
    longProperty = newValue;
    pcs.firePropertyChange("longProperty",
    ➥oldValue, newValue);
  }
}
```

❸ **Listeners for specific properties**

❹ **Methods for stringProperty**

❺ **Methods for longProperty**

Wow! That's a lot of code. What's surprising is that the code shared across the two properties takes up more space than the code handing the individual properties combined! But we shouldn't be too hard on plain old Java, because there could have been a lot more code. The `PropertyChangeSupport` field ❶ handles most of the bookkeeping of tracking and firing the property-change events. The handling code in that class would translate to about another four pages of code. The support isn't perfect, because you still need to create methods that match the bound properties pattern. First you wire the prescribed methods to handle listening to all property changes ❷, and then you add listeners for particular properties instead of all properties ❸.

Now that you've written well over half the code, it's time to write the property methods. First let's examine `stringProperty` ❹. The field and getter are exactly the same as for a nonbound property. The only difference is the setter. Instead of changing the field, you must first cache the old value. Then you set the field and fire the change event. It's important that you do these items in this particular order, because the JavaBeans spec requires calls to the getter during a property-change event to reflect the new value.

Looking at `longProperty` ❺, you can see that the evolution of the Java language has also made some of the handling easier. `PropertyChangeSupport` has `firePropertyChange` methods for firing property changes for `Object`, `int`, and `boolean`. How can

this code compile? Java 5 added support for autoboxing, so the `long` primitive values are automatically converted into `Long` wrapper objects.

Considering all the boilerplate code and nuances in the implementation of JavaBeans bound properties, it's no wonder many developers choose not to implement this pattern in their code. But how does Groovy make this easier?

3.3.2 JavaBeans bound properties: the Groovy way

Groovy has several language features that were specifically designed with JavaBeans in mind. The first is the notion of a `GroovyBean` property. Whenever a field is declared without a visibility modifier, Groovy automatically creates the boilerplate getter and setter for it in the JVM bytecode, including the private field that backs the property. To mirror the declaration of a property, Groovy also gives priority to JavaBeans properties whenever a field is accessed on an object. To be more precise, instead of a field access on an object, Groovy has property access. If there is a correctly constructed getter method, then the results of that method are used. Only if no property exists is the field on the object directly accessed. In other words, writing a class in Groovy like the following one

```
class Person {
    String name
}
```

has the same effect and produces equivalent bytecode to writing it the Java way:

```
public class Person {
    private String name;
    public static void setName(String name) { this.name = name; }
    public String getName() { return name; }
}
```

Groovy has a second language feature that assists in creating bound properties: the Abstract Syntax Tree (AST) Transformation framework. When Groovy encounters an annotation while compiling a class, the compiler will inspect the metadata on the annotation itself. Some metadata will instruct the compiler to load additional classes to do secondary alterations to the syntax tree of the compilation in progress. These annotations are called *AST Transformations*, and when attached to particular fields, classes, and/or methods, they can add boilerplate code that the developer need not write. One of the annotations that come packaged with Groovy handles the generation of bound JavaBeans.

MAKING SIMPLE THINGS EASY

When you want to mark a property in your model as one that can be observed, all you need to do is make sure the property is annotated with the `groovy.beans.Bindable` annotation. The Groovy compiler will see the annotation and automatically generate the long boilerplate to ensure that a property change will be observed. This is accomplished via AST annotations.

The `@Bindable` annotation can be applied in two places. First, it can be applied on the class itself. This causes all Groovy properties in that class to be treated as though

About AST annotations

The topic of AST annotations is interesting and could fill a book of its own. We've said before that Groovy supports a feature called *metaprogramming*: the ability to change the behavior of a class at runtime. It turns out AST annotations allow developers to have their say with regard to metaprogramming at compile time. For thorough coverage of the AST Transformation framework, we recommend that you see the AST chapter in *Groovy in Action*, second edition (Manning, 2012).

they're observable. In the following code, all the properties declared in the class will be observable via `PropertyChangeEvent` events:

```
import groovy.beans.Bindable
@Bindable class MyApplicationClass {
    String propertyOne
    int propertyTwo
    boolean propertyThree
}
```

The annotation can also be applied to individual properties in a class. This allows a class to pick and choose the properties that need to be exposed as bound properties. This may be useful to prevent properties that change too frequently or infrequently from being observable. It also allows the user to keep implementation details free from prying eyes.

In the following class, only the changes in `propertyOne` and `propertyThree` can be observed via `PropertyChangeEvent` events:

```
import groovy.beans.Bindable
class myApplicationClass {
    @Bindable String propertyOne
    int propertyTwo
    @Bindable propertyThree
}
```

By using the AST Transformation facilities introduced in Groovy 1.6, you can make standard read/write properties into bound properties. But in a large application, not everything is standard; sometimes you need a little more magic than is commonly called for.

MAKING DIFFICULT THINGS POSSIBLE

Often, a JavaBean property does more than stash a value. Not all changes are simple: sometimes they have side effects, sometimes the property is a contributing part of other properties, and sometimes the property may not represent what you think it does. That was part of the thinking behind the JavaBeans getter/setter pattern. We don't want to make the ease of use get in the way of the cool stuff.

How do you deal with complex setters without making users roll it all themselves? By using conventions. The supporting items, when added by the `@Bindable` transformation, follow the same pattern and naming conventions

The first thing the transformation does is look for an existing field of type `PropertyChangeSupport`. If it finds such a field, the magic trick is over. This assumes the user has a clear understanding of what they're doing and what they want to do. On the other hand, if such a field doesn't exist, you create one and name it `this$PropertyChangeSupport`. Yes, that's a weird name for a field, but rest assured that the JVM is perfectly capable of understanding such names. The Groovy compiler chooses that name to keep the chances of a name collision very, very low. The transformation then performs the burden of adding the boilerplate methods required to follow the JavaBeans bound properties pattern: `addPropertyChangeListener`, `removePropertyChangeListener`, and `getPropertyChangeListeners` in their various overloaded forms. Finally, you add one last method of convenience: `firePropertyChange`.

Once the groundwork is laid, the transformation can work on the custom parts of the code: the bound setter methods. If no setter methods exists, then one is generated using either an existing `PropertyChangeSupport` field or one that is generated by the transformation. If the setter exists, already defined by the user, then the body of the setter method is wrapped with code that will store the old value of the property and fire a property-change event with the new value before returning from the method. All this is handled without you having to write any boilerplate code.

This isn't the end of the road when it comes to bound properties. There's one last option: manual support. If there's another property in the class being written (or its superclass), then you can call the `firePropertyChange` method directly as needed. This provides one last escape hatch for bizarre properties that defy standard configurations.

The end result is that you as a developer don't have to worry much when you add an observable property to a class. As long as you follow the conventions, the compiler will work its magic. It's also good to know how you can deviate from the conventional path and enter into the realm of configuration. The following listing shows several options being used at once, each with specific effects.

Listing 3.6 Bound properties with `@Bindable`

```
import groovy.beans.Bindable

class ManyWaysToBind {
  long lastUpdate

  @Bindable String autoEverything

  @Bindable String customSetter

  public void setCustomSetter(String newValue) {
    lastUpdate = System.currentTimeMillis()
  }

  public void checkBlink() {
    boolean blink =(System.currentTimeMillis()/1000)%2
    firePropertyChange("blink", !blink, blink)
  }
}
```

1 Not bindable

2 Simply bindable

3 Custom setter

4 Manually bindable

This code sample demonstrates the four ways you can declare a bound property in Groovy. The first example ❶ is to have the property not be bound. Groovy will generate a non-eventful setter. The second example ❷ is the most common case of a bound property: a simple write to the backing field followed by a `PropertyChangeEvent` event. But all the code to handle that is generated by the compiler and not seen in the bean class. A less common example is the third case ❸, where other tasks need to be performed in parallel to setting the property. The transformation wraps the corresponding setter method body with the needed code to create a `PropertyChangeEvent` event. Finally there's the last case ❹, where nothing short of almost total control will suffice. If the transformation has added the support classes by being attached to other properties, then you can easily fire a property-change event whenever you need to. This isn't the best way to simulate a cursor blink, but there are worse ways.

3.3.3 Handy bound classes

In addition to creating your own model and adding `@Bindable` annotations, Groovy has two other handy classes that provide simple property-change support semantics without the ceremony of a full class: `groovy.util.ObservableMap` and `groovy.util.ObservableList`. You would typically choose one or the other when looking for an observable collection.

OBSERVABLE MAP

When it comes to property access, the map class in Groovy is one of the classes that get the most special treatment from the runtime (see the following listing).

Listing 3.7 `java.util.Map` access in Groovy

```
Map map = [:]

map.put('key', 'value')
result = map.get('key')

map['key'] = 'value'
result = map['key']

map.key = 'value'
result = map.key
```

A value stored in the map can be accessed in one of three ways: via the `put(K, V)` and `get(K)` methods, via subscript notation, and via property notation. The most interesting one for this discussion is the property notation. This makes a map look like a custom defined class without having to define the properties and methods of the class it's implementing. The ability of a map to support both types of notations for property access pretty much makes the distinction between a POJO instance (or bean) and a map disappear. The same can be said in regard to property access from a bean's point of view: both notations are supported.

Observable maps make all the key/value pairs stored in the map react as though they're observable properties. They also add the option to filter out what properties you want to fire the property events, as shown in the next listing.

Listing 3.8 `ObservableMap` in action

```
import java.beans.*

obj1 = new ObservableMap()
obj2 = new ObservableMap({k, v -> !(k ==~ /[A-Z]+/)})

listener = { evt -> println """$evt.propertyName:
  $evt.oldValue -> $evt.newValue""" } as
  ➥PropertyChangeListener

obj1.addPropertyChangeListener(listener)
obj2.addPropertyChangeListener(listener)

obj1.three = 3
obj2.three = 3
obj1.three = 'three'
obj2.three = 'three'

obj1.FOUR = 4
obj2.FOUR = 4

obj3 = new ObservableMap()
obj3.addPropertyChangeListener('passwd',
  { println "password was changed" } as
➥    PropertyChangeListener)

obj3.user = 'griffon'

obj3.passwd = 'rocks'
obj3.put('passwd', 'rocks!')
obj3['passwd'] = 'totally rocks!'
```

❶ **Assign values to existing keys**

❷ **Assign values to new keys**

To begin this example, you compare the functionality of two observable maps. The first object is a no-frills observable map, and the second is an observable map with a filter that prevents properties that only contain capital letters from firing events. A listener is attached to report any changes. When changing the objects, the first set of events fire identically, with both objects reporting a change from `null` to `3` and a change from `3` to the string `three` ❶. The next set of changes only fires once, because `obj2` has a filter that prohibits properties like `FOUR` from generating events ❷. Next you create a new object and only listen to changes for the property `passwd`. Setting the property `user` doesn't trigger the event listener you've registered on `obj3` because you're only listening to the property `passwd`. But if you set the value of `passwd` in any of the acceptable ways, a property change event is fired, thus sending the following message to the output:

```
password was changed
```

That message should appear three times because you changed the value of the `passwd` property exactly three times.

Observable List

The other class that gets a lot of built-in use from Groovy is the list. Because lists don't associate names with their content but instead associate ordinal positions, there's no mapping of properties as there is with maps. Instead, some of the operators in Groovy

are overloaded when used on objects that are lists: the subscript operator and the left-shift (<<) operator. The `ObservableList` fires its events when its contents change.

One important change from the property-change listeners for a map is the event object that's generated from an `ObservableList`. The properties don't change: the contents of the list change. In order to properly reflect that, you add an additional field to the property change event: `index`. This is the actual index in the list of the relevant change, as demonstrated in the following code.

Listing 3.9 `ObservableList` in action

```
import java.beans.*

list1 = new ObservableList()
list2 = new ObservableList({v -> v != null})

listener = { evt -> println """at $evt.index:
  $evt.oldValue -> $evt.newValue""" } as
    PropertyChangeListener

list1.addPropertyChangeListener(listener)
list2.addPropertyChangeListener(listener)

list1.add(1)
[2,3,4].each { list1.add(it) }
list2 << 1 << 2 << 3 << 4

list1.set(3, 'three')
list2[3] = 'three'

list1[2] = null
list2[2] = null
```

❶ Trigger change events by adding element

❷ Trigger change events by setting elements

You make two lists here. The first will react to all content changes, and the second won't report property changes if the element added is `null`. Next you create an element listener. Instead of worrying about the `propertyName` on the event object, you worry about the `index` on the event object. Adding an object to the list is treated the same whether you use the ordinary `add` method or the left-shift (or `insert`) operator, and each add generates a distinct event ❶. Changing the particular values also generates the same event, whether you do it via the method or the array accessor ❷. These two lists aren't identical; the second list won't generate events for `null` objects. If you set the second position to `null`, an event is generated only for the change to `list1`.

These are the principal ways you can create changes in Groovy objects that can be observed by an interested party. It's nice that you can see these changes, but that's only half the story. You want to do cool stuff with these changes, and it would be magical if you could do some of it automatically, as if one value was bound to another value…

3.4 Have your people call my people: binding

What is a binding? In Griffon, a *binding* has the following three constituent parts:

- *Trigger*—Tells the binding that it needs to update
- *Read*—Tells the binding what the new value is
- *Write*—Takes the new value and does something with it

JavaBeans bound properties provide the simplest manifestation of a binding. From a JavaBeans point of view, the trigger means the property is being changed, the read means the source is reading the value of the property, and the write means the target is storing the new property value in the variable.

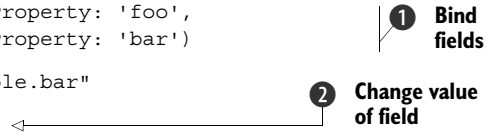
But you can make any of the three pieces of a binding as complex as you need to. When you write a Griffon application, you're probably using some complex binding magic without even knowing it. That's the point.

3.4.1 A basic binding call

The `SwingBuilder` class in Groovy contains more than just visual widgets; it also contains helper nodes that are useful in building GUI applications. One of those is the bind node that allows you to cleanly bind together the state of two objects. As we'll discuss later, it can bind both bound and unbound properties. Let's start with the simple tricks first, as shown in the next listing.

Listing 3.10 Simple binding example

```
import groovy.beans.Bindable
import groovy.swing.SwingBui
swing = new SwingBuilder()

class BindSample {
  @Bindable String foo
  String bar
}

sample = new BindSample(foo: 'One', bar: 'Two')
swing.bind(source: sample, sourceProperty: 'foo',
           target: sample, targetProperty: 'bar')

println "foo=$sample.foo bar=$sample.bar"

sample.foo = 'Three'

println "foo=$sample.foo bar=$sample.bar"
```



You can paste the contents of listing 3.10 into a file (for example, `bindExample.groovy`) and run it directly from the command line, as long as you have Groovy installed in your environment. Run the following command to satisfy your curiosity:

```
groovy bindExample.groovy
```

The execution of this script yields the following output:

```
foo=One bar=One
foo=Three bar=Three
```

Because this is an all-in-one example as a standalone Groovy script, you have to take care of a bit of ceremony that you normally wouldn't need to do. First, you import `SwingBuilder` and instantiate an instance of it. Normally you wouldn't need to do this in Griffon, but because this is a standalone script you must do it explicitly. The next step is something you would normally do in a model class: you declare the class and the `@Bindable` attribute.

With the ceremony over, you can get to the meat of the example. You now bind the field `bar` on `sample` to be the same value as `foo` ❶. This binding takes effect instantly; when you check the value of `bar` you see that it's now `'One'`, which was the value the field `foo` had when you instantiated the object. When you change the value of the field `foo` ❷, it's also reflected in the field `bar`, which is now `'Three'` as well.

That wasn't so hard, was it? You've specified all the required members of a binding: a source and the property to be read, along with a target and its property where the value will be written. But there are different ways to create a binding.

3.4.2 The several flavors of binding

There's more than one way to call the `bind` method. You're expressing the essence of the binding and extracting the needed ceremony from the context of the `bind` call. But just as with long division, before you can understand the essence of the binding, we need to examine a few fully expressed binding calls. There are three basic flavors:

- By and large the most common flavor of binding is a *property-to-property binding*.
- A less common but equally valuable flavor *separates the trigger from the read.*
- The last flavor to fully declare a binding is a *hybrid of the previous two forms.*

SOURCE AND TARGET PROPERTY BINDING

Here's a simple property-to-property binding:

```
bind(source: ownedCheckBox, sourceProperty: 'selected',
     target: model, targetProperty: 'owned')
```

In this case, the read and the trigger are expressed as the same thing: a bound property on a particular object instance. The instance that will provide both the change notification and the changed values is declared in the `source` attribute. The property that will both provide the value and provide the trigger when it's changed is declared in the `sourceProperty` attribute. Both of these attributes constitute the trigger and the read. The location to which you'll write the value is declared in two parallel values, the `target` attribute and the `targetProperty` attribute. Both properties together constitute the write.

EVENT TRIGGER, CLOSURE READ, AND PROPERTY WRITE BINDING

Separating the trigger from the read is usually done when the property providing the read value isn't observable, but other events in the widget may notify you that states need to be updated. One classic example is `JTextComponent`:

```
bind(source: myJTextArea.document,
     sourceEvent: 'undoableEditHappened',
     sourceValue: {myUndoManager.canUndo()},
     target: undoBtn, targetProperty: 'enabled')
```

The `text` property of the `JTextComponent` and its many subclasses doesn't provide a means to directly notify you when the value has changed, primarily because it isn't directly backed by a field but is the result of calculations done against the `Document` object backing the text field. The `Document` object does provide events that you can

latch on to in order to detect changes. To declare a non-property event trigger, you need to declare `source` and `sourceEvent` attributes in the `bind` call.

In addition to binding to an event, you can bind to values that aren't directly properties. For example, if you're updating an Undo button, the trigger is the document manager but the actual value is managed by a third-party `UndoManager` instance. In this case, you need to declare a `sourceValue` element and pass in a closure that will provide the value to be read. This attribute is entirely independent of any other attributes.

Targets, on the other hand, don't provide as much flexibility as sources and triggers. Both the `target` and `targetProperty` attributes are needed to determine where to write the value. If you need to do fancy stuff, you can convert the value to something else. Say, for example, you bind a numeric property of the model to an input field that has a currency format. The text coming from the input field will have a format that may not be appropriate for a numeric algorithm; this is where you require a converter.

SOURCE EVENT AND PROPERTY, TARGET PROPERTY

The last way to fully declare a binding is a hybrid of the previous two. It isn't nearly as common as the other two methods and is mentioned here for completeness. You can trigger from an arbitrary event on an object and also read the property from the same object:

```
bind(source: myToggleButton,
     sourceEvent: 'actionPerformed',
     sourceProperty: 'selected',
     target: model, targetProperty: 'toggleSelected')
```

The same `source` attribute can be shared between a `sourceEvent` and a `sourceProperty` element. In fact, they must share the same source if both attributes are declared. If you need to have different sources, you're better served by using the `sourceValue` attribute and expressing the read as a closure. Finally, the write is expressed by the `target` bean and the `targetProperty`.

That covers the three basic forms of a fully expressed binding. But there are ways to tease out the essence from the ceremony.

3.4.3 Finding the essence

You can use three approaches to more clearly express the essence of the binding:

- Provide the `source` property and `target` property as unnamed arguments.
- Imply `source` and `target` property bindings by using the `bind()` node as part of another `SwingBuilder` node.
- Express the trigger and read as closures.

IMPLICIT ARGUMENT PROPERTY

In chapter 4, we'll go deeper into the structure of a builder node, but for this discussion suffice to say that nodes can take arguments and attributes. Arguments don't have labels, but attributes do. When a bind node is passed an argument of type

String, it's presumed to be the value for the sourceProperty and targetProperty attributes if those attributes aren't passed in and are needed (see the following listing).

Listing 3.11 Implicit argument property examples

```
bind('selected', source: ownedCheckBox,
     target: model, targetProperty: 'owned')

bind(source: myJTextArea.document,
     sourceEvent: 'undoableEditHappened',
     sourceValue: {myUndoManager.canUndo()},
     'enabled', target: undoBtn)

bind('selected', source: myToggleButton,
     sourceEvent: 'actionPerformed',
     target: model, targetProperty: 'toggleSelected')

bind('selected',
     source: firstCheckBox,
     target: secondCheckBox)
```

The first three nodes in the example are reworked versions of the prior three examples, except that in the first and the third nodes the sourceProperty attribute is implied by the argument value, and in the second node it's the targetProperty attribute that's implied. Note that the argument value can also imply both the sourceProperty and targetProperty attributes, as shown in the fourth node. The result of the fourth bind node is that the state of the secondCheckbox selection is driven by the first check box.

CONTEXTUAL PROPERTY BINDINGS

Another instance in which you can glean the essence of the ceremony is when the bind node is constructed in the context of a view script and provides the declared value of the attribute. In this case, the object that the node is creating becomes either a source or a target object, and the attribute becomes the property for the implied portion of the binding. The four nodes shown in the next listing are semantically identical to the four nodes in listing 3.11, except that you recast them to be written as properties in the declaration of the visual nodes.

Listing 3.12 Contextual property binding examples

```
checkbox('Owned',
  selected: bind(target: model, 'owned'))

button('Undo',
  enabled: bind(source: myJTextArea.document,
    sourceEvent: 'undoableEditHappened',
    sourceValue: {myUndoManager.canUndo()})

button('State Toggle',
  selected: bind(sourceEvent: 'actionPerformed',
    target: model, targetProperty: 'toggleSelected')

checkbox('Second',
  selected: bind('selected', source:firstCheckBox))
```

What is notable about these examples is that half of the contextual properties result in the source being represented by the context, and the other half result in the target being represented by the binding. How does Griffon know where to use the context? It looks at what is already explicitly provided and then provides the rest from the context. But what happens if you provide both the source and the target in the binding but still do the `bind()` node as the value to an attribute? The context of the bean then isn't used to calculate the binding; instead, the attribute is set with a `BindingUpdatable` object, which stores the realization of the binding.

Peeking behind the curtain

How does the bind node deal with contextual properties? In part by using a *stand-in* object and using an attribute delegate in the `FactoryBuilderSupport` to finish the processing.

Semantically, the bind node is evaluated before the parent object is calculated. So at the time the bind node is processed, it doesn't have any way to get at the contextual property. The `BindFactory` does whatever work it can with the explicit portion of the node and then returns the half-built objects to stand in for its fully bound state.

Once the parent node starts processing, the `FactoryBuilderSupport` allows registered attribute delegates a chance to post-process the attribute values before it applies them as properties to the resulting object. The `BindFactory` registers a delegate that will identify the stand-in object and finish the processing of the binding with the name of the attribute and the instance of the object being constructed.

BINDING TO A CLOSURE

The final way to extract the essence of the binding from the full ceremonial declaration of the bind node is to express the trigger and read values as a closure containing the values to be queried. This is by far the most concise and expressive way to declare a binding in Groovy or Griffon. This is the preferred way to define bindings where the binding direction is from source to target only, as well as when the binding itself is very simple. The following snippet shows how such closures can be defined:

```
checkbox('Owned', selected: bind { model.owned } )

checkbox('Second',
  selected: bind { firstCheckBox.selected } )

label(text: bind { "$model.completed / $model.total Completed" } )
```

You'll note that only two of the samples from the previous set of examples have been repeated. Binding to a closure isn't possible for all scenarios. A binding closure can only be used to express the source and the source values. If a binding needs to trigger from a non-property event, or the binding gets its source context from the node, then a closure binding can't be used.

The closure binding is deceptive in its simplicity and power. The first two examples show a simple closure binding: a single object reading a single property value. The

third example shows a more powerful use: multiple properties being processed into a new value. In this example, you can see that the closure binding represents a fourth form of a fully expressed binding, one that triggers an update when one of several distinct properties determined at runtime are changed.

Peeking further behind the curtain

The ability to trigger from multiple properties is mostly a side effect of how the closure binding is implemented. Two language aspects of Groovy combine to allow for the closure binding to work. The first is the fully dynamic nature of each of its method invocations. When a method is invoked in Groovy, the invocation is passed through the metaclass to allow it to provide alternate options dynamically at runtime. This is what makes a dynamic language dynamic: the presence of some sort of Meta Object Protocol.

The second aspect of Groovy that allows the closure binding to work is the use of a delegate on the closure object. Each closure is represented by a distinct Java object, and a delegate can be assigned to that object. The unbound variables in the closure are then resolved against either the object in which the closure was declared or the delegate object.

The closure binding uses both of these features to get a listing of the objects and properties that will be inspected for operability. When the binding instance is bound, the properties for which changes can be observed are listened to for changes. When any of the properties change, the value of the closure is evaluated, the target is updated, and, if needed, new properties have listeners attached.

3.4.4 Other binding options

Beyond the basic requirements of the binding, you can wire in some other options that affect the processing of a binding update. The binding can convert values being read into other values, and it can also validate values and keep invalid values from being passed to the target. When you're using a binding, you aren't stuck with a one-to-one, take-it-or-leave-it update; values can be adjusted and even rejected. Extra attributes can be added to the bind node to provide this added functionality. There are also corner cases relating to some uses of the bind node that can be resolved by additional attributes.

CONVERTING VALUES READ FROM A BINDING

A common requirement is to translate one value into another. For example, a data field may have a public name such as Red, Green, or Blue, but the data model may require these values to be stored as `int`s: `0xff0000`, `0x00ff00`, and `0x0000ff`, respectively. This is clearly a task that should be done as close to the view as possible, to maximize the time that the data can be stored in its preferred format. To do this, you pass an attribute `converter` into the binding arguments and provide a closure. This closure is given the value obtained from the read of the binding, and the result of the closure is passed into the write of the binding. The following listing maps the common names of colors to the internal AWT object representing those colors.

Listing 3.13 Converting a binding value

```
def colors = [Red:   Color.RED,
              Green: Color.GREEN,
              Blue:  Color.BLUE]

comboBox(items:colors.collect {k, v -> k}, id:'combo')

label('Look at my colors!',
  foreground: bind (source:combo, 'selectedItem',
    converter: {v -> colors[v]}))
```

The issue is that the user expects to see the `String` names, but the label wants a color object. The closure you provide takes the `String` value and maps it to the regular value.

But what about instances where the bound values aren't translatable to a model value? How do you ensure that only the valid values are written to the target? Enter the `validator` attribute.

VALIDATING VALUES READ FROM A BINDING

In listing 3.14, the `validator` closure is called when the content of the text field changes, and if it returns a Boolean value of `true`, the binding update continues. If the validator returns any other value, the binding update is silently stopped, and the value read from the source is neither converted nor written to the target. In this listing you change things up slightly from the previous incarnation. Instead of a combo box that limits the user's entries, you make the user type a valid color name into a text field.

Listing 3.14 Validating a binding value

```
def colors = [Red:   Color.RED,
              Green: Color.GREEN,
              Blue:  Color.BLUE]

textField('Green', id: 'colorField')

label('Type a color!',
  foreground: bind (source: colorField, 'text',
    validator: {colors.containsKey(colorField.text) },
    converter: {v -> colors[v]}))
```

To ensure that you only try to change the color for valid values, you add a `validator` closure that checks to see if the color in the text field exists in the `colors` map. If it doesn't exist, then it's as if the binding doesn't exist. But if you pass the test, then the label's color is changed.

It's important to note here the relative order of evaluation of converters and validators. Validators are called and evaluated before converters so that if a value isn't valid, you won't attempt to convert or write the value. This is a good thing for the converter, because it can presume that the value has been vetted prior to being passed in to the converter. The converter doesn't have to check for bad values such as nulls or division by zero if they're stopped by the validator. In addition, if the converter has side effects (such as caching values), then those side effects occur only when the values are actually updated.

SETTING AN INITIAL VALUE

One corner case that can result from a contextual binding is that the attribute being declared on the node is the source, and usually the declaration of the value is where the bind node goes. There are two ways to solve this. The first is to place the bind node on the target attribute on the target node. But this isn't always desirable, for many reasons. It may make the code harder to read, or the target node may not be declared and may be a value passed into the script binding. In those cases, you can use the `value` attribute to specify a value to be passed into the declaring source node.

Why would you need to set the value? Because often the default isn't what you want to begin with:

```
checkbox("Check spelling before sending mail",
  selected: bind(target:model, 'spelcheck', value:true))
```

Some UI option should always be turned on by default. The `checkbox` widget defaults to unselected, and if you don't set a value, some people will always send poorly written email. After the binding is set up, the value of the source is set to the value set in the `value` attribute, and if this represents a change, the binding will automatically fire.

TURNING A BINDING OFF

Sometimes you don't need a binding to fire the updates automatically. The resulting object representing a fully assembled binding has the option to fire the bindings manually or on demand. This allows the binding to represent the data flow between two different properties without necessarily requiring them to be constantly in sync.

The `bind` attribute controls whether a binding is set to automatically update. The attribute accepts Boolean values, and by default it's set to `true`.

When would you *not* want a binding to be automatic? One example is a form that directly updates the application model preferences when the user clicks OK. But what if the user doesn't want the changes set immediately? By setting the `bind` option to false, the binding updates can be managed manually, as you'll see later in the chapter.

TWO-WAY BINDING

All the bindings we've covered so far establish a one-way street between source and target. In other words, the value travels from the source to the target. Sometimes, however, you want the value to travel in both directions. A first approach to solving this issue could be defining two bindings, exchanging the source for the target, like this:

```
checkbox(id: 'check', selected: bind('value', source: model))
bean(model, value: bind('selected', source: 'check'))
```

Don't worry too much about the `bean` node for the moment; we'll cover it in the next chapter. Suffice to say it allows you to use the builder syntax with any object. Unfortunately, these bindings will cause an endless loop of events as soon as one of the two properties changes value. This happens because one binding isn't aware of the other—they don't communicate in any way. You can use an additional property to get

rid of the endless loop; its name is `mutual`, and it takes a Boolean as a value. Keeping both properties in sync is done as follows:

```
checkbox(selected: bind('value', source: model, mutual: true))
```

You must pay attention to which object is set as the source, because it dictates the initial value transferred to the target. In the previous example, the `value` property of the `model` object is bound immediately to the `selected` property of the check box. The following example shows the inverse:

```
checkbox(selected: bind('value', target: model, mutual: true))
```

Take special notice of the subtle difference.

How do you get access to the objects managing the bindings? It's as if they have their own secret life. There are, however, ways to access that secret life. Fair warning: the next section includes specific details, and the discussion is fairly technical.

3.5 The secret life of BindingUpdatable

When you're creating a binding, you need to use several objects to monitor and manage all the moving parts. Listeners need to be added to events, notes about the values of objects being read need to be maintained, and the ultimate target of the write also needs to be referenced. Wouldn't it be nice if there was a nice, organized object to track the end result of a binding? The good news keeps coming, because such an object does indeed exist: `BindingUpdatable`! In this section, we'll look at the `BindingUpdatable` object and how you can manage bindstorms. You'll also see how to manually manage and group bindings.

3.5.1 Keeping track of bindings with the BindingUpdatable object

Enter the `BindingUpdatable` interface. This interface handles operations that may be of interest to an outsider that has found the object tracking the binding. The `BindingUpdatable` object's binding can be turned on or off, can be reset, can fire the update immediately, and in some cases can even run in reverse! This object doesn't expose every detail directly, because the details vary wildly between declarations of the binding. If you know what you're expecting, however, you can always cast or duck-type down to a more specific application.

The trick comes in getting the access to this `BindingUpdatable` object. The object returned from the bind node is a `BindingUpdatable`, so you would think that getting it would be simple. But one way won't work: setting it as an attribute on some other node. Why won't this work? Because it looks the same as a contextual binding, and this object is specifically checked for during the contextual binding magic (see the "Peeking behind the curtain" sidebar earlier in this chapter).

What options are left? You could try to use the assignment operator. This can result in some funny-looking (but effective) code:

```
label(text: statusBinding = bind(source:model, 'status'))
```

The problem is that many programmers aren't used to seeing assignments that aren't standalone expressions. Some neat hacks result from using the assignment operator, and they can yield terse code. But it's often called *write-only code* and is error-prone for even the most seasoned programmers. For maintenance reasons, clarity over cleverness should be the standard.

The last option, and the preferable one, is to use an `id:` attribute in your bind node. As we discussed in chapter 1 and will discuss in depth in chapter 4, every node built by the builders in Griffon accepts the `id:` attribute and stores the resulting object in the binding as though it were set with an assignment expression. You use it just like any other attribute:

```
label(text: bind(source:model, 'status', id:'statusBinding'))
```

This serves to keep the information about the binding in the binding node. It also has the nice side effect of making the code entirely declarative, with no imperative statements obscuring the intended declarations.

3.5.2 Managing the bindstorm: bind(), unbind(), and rebind()

When it comes to binding, one of its strengths can also be one of its greatest weaknesses: automatic updates. On the one hand, they magically make values update when the source value changes; on the other hand, the target value can unexpectedly change without much warning or explanation. Many bindings happening at the same time has gained a nickname: a *bindstorm*, where automatic updates fire *en masse* and sometimes trigger other updates that continue to fire. The orderly execution of the application has no choice but to run away and hide until the storm passes. To address this problem, the `BindingUpdatable` object has three methods to tweak the automatic nature of the binding in question: `bind()`, `unbind()`, and `rebind()`.

The `bind()` and `unbind()` methods work as a pair. The first serves to enable any automatic portion of the binding. This includes adding event listeners (including property-change listeners) to the appropriate objects for the binding. The `unbind()` method does the opposite: it disables any automatic portion of the binding. This generally results in the removal of any listeners that the current binding may have in place.

We need to point out two caveats about the finer parts of these two methods. First, the automatic update portion of the binding may not be driven by JavaBeans events. For example, the `animate()` node in the `SwingXBuilder` runs its updates from a `javax.swing.Timer` instance, so `bind()` and `unbind()` methods in this case start and stop the timer. This may have the side effect of preventing a program from exiting if the bindings are left in place. The second caveat is that these methods are *idempotent*, meaning 1 call to `bind()` has the same effect as 100 (until `unbind()` is called). The listeners will be added only once until they're removed.

But how do you keep a binding in good working order in a constantly changing environment? That is the purpose of the `rebind()` method: it causes a binding to unbind and rebind itself, but only if it's currently bound. This method usually wouldn't need to be called by external implementations, but there are corner cases

that Griffon can't handle by itself. One example is a closure binding where one of the observed properties is held in an array, and the array changes. There are no ways to track the update of the array by observation. A quick call to `rebind()` ensures that the listeners are properly attached without having to also check to see if the binding is currently active; the checking is done under the covers.

3.5.3 Manually triggering a binding: update() and reverseUpdate()

A `BindingUpdatable` object for a binding that isn't currently bound isn't totally useless. In addition to having its binding activated, the binding can also be manually fired via the `update()` method. When this method is called, the read and write portions of the binding are fired as if the trigger for the property change had caused their update to occur automatically. This is particularly useful when the `unbind()` method has been called or the binding was created with `bind: false` as an attribute. It's worth remarking that the read is performed on the source and the write happens on the target.

The `update()` method also has a corresponding method, `reverseUpdate()`, that will, if possible, reverse the role of the read and the write methods and do the update in reverse. This method won't work in all situations; in those cases, the reverse update will silently fail. In particular, this will happen if the binding is a closure binding or the source comes from a `sourceValue` closure. Principally this works with property-source bindings where the source property is writable.

3.5.4 Grouping bindings together

Having all these accessible features in the `BindingUpdatable` class allows for powerful manipulation. But some of the more complex binding scenarios often involve a large number of bindings that need to be managed in concert. Manually calling each one individually can be a drag, even with all the syntactic sugar that Groovy affords. This brings us to the final piece of the puzzle: *binding groups*. Binding groups allow you to aggregate multiple `BindingUpdatable` objects into a single `BindingUpdatable` object that passes the method calls to each component binding.

`SwingBuilder` (discussed in chapter 4) has a node named `bindGroup()` that creates an instance of `org.codehaus.groovy.binding.AggregateBinding`. The only attributes of note are the `id:` attribute for storing a reference to the binding group, and the `bind:` attribute for the initial binding state. The group usually begins in a bound state, and any bindings added to it will be bound (this is where the idempotent nature of `bind()` comes in handy) unless the `bind:` attribute has been set to false.

Generally you'll want to define your binding groups before declaring any of your bindings. That's because the best way to add a binding is to add a `group:` attribute in the binding as you declare it, passing in the binding group you bound earlier. This will automatically add the binding generated into the binding group.

For a concrete example, consider a form where the user may want to keep updates from hitting the model until they apply the changes explicitly. For brevity, let's consider

only two possible options, each represented by two Boolean fields in the model. The view code may look something like this:

```
bindGroup(id:'formElements', bound:false)

checkbox('Option A', selected: bind (target:model,
  'optA', group:formElements))
checkbox('Option B', enable: bind (target:model,
  'optB', group:formElements))

button('Apply', actionPerformed:
  { formElements.update() })
button('Reset', actionPerformed:
  { formElements.reverseUpdate() })

formElements.reverseUpdate()
```

Because this is an update-on-demand situation, you set the binding group to unbind its contents with the `bound` attribute being set to false. Then, in the widgets, the bindings to the selected property are added to the bind group `formElements`; you can later push updates to the model or pull updates from the model. And for good measure, at the end of the script, you pull the values from the model into the check boxes.

There's more than one way to add a binding to a binding group. Doing so using the `bindGroup` attribute make sense when you're declaring bindings in a view. But you aren't limited to creating bindings in declarative code. Two methods on the `AggregateBinding` allow you add and remove bindings to its internal set: `addBinding()` and `removeBinding()`. These methods (keeping with the theme) are also mostly idempotent. We say *mostly* idempotent because order does matter. The order in which a binding is added to the binding group is the order in which it will be called when the relevant `BindingUpdatable` methods are called. Removing the object will result in a later call to add the same object, placing that object last in line, as though it had never been seen.

3.6 Putting it all together

When you're creating a model and view that magically bind their values together, the binding can include a lot of moving parts. But as we've said, knowing about these moving parts is a lot like using long division: on some level you need to know how it works, but you can take most of it for granted. A full example will help illustrate the point that in general, once the bindings are in place, you can take their magic for granted. In this section, you'll create a mortgage calculator like the one shown in figure 3.2.

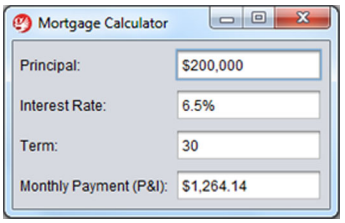


Figure 3.2 The completed Mortgage Calculator app

A ubiquitous part of any mortgage website is enabling the user to enter their loan amount, the interest rate, and the term (the number of years over which they want to

pay the mortgage). The resulting formula to calculate the principal and interest (P&I) payment is fairly simple (when compared to some other financial calculations):

$PI = P * (r / (1 - (1 + r) \wedge -N))$

In this formula, *PI* represents the monthly principal and interest due, *P* is the initial principal, *r* is the fractional monthly rate, and *N* is length of the loan in months. This is enough to get started.

3.6.1 Setting up the model

First you create a project. Open a shell prompt into the directory where you want to create the application, and create a new `MortgageCalc` application:

```
$ griffon create-app mortgageCalc
```

For a model, you want fields for the principal, rate, and term. You also need a property that will calculate the P&I value from the provided fields.

Listing 3.15 MortgageCalcModel.groovy

```
import groovy.beans.Bindable

@Bindable
class MortgageCalcModel {
    float principal
    float monthlyRate
    float months

    float getPayment() {
      return principal * monthlyRate /
        (1-Math.pow(1/(1+monthlyRate),months))
    }
}
```

The principal, monthly rate, and months fields are all observable, because you need to know when the model's writeable values have been updated so you can get the new payment value.

3.6.2 Defining a view

The next step is to create the view. You'll do this iteratively so you can see how some of the pieces of the binding make their way in; we'll show pieces of the view only as they're needed. The first pass is to create form elements for each model field and map them to their model fields (see the next listing).

Listing 3.16 Bindings on the model properties

```
label('Principal:')
textField(text: bind(target:model, 'principal',
  value:'330000'))

label('Interest Rate:')
textField(text: bind(
  target:model, 'monthlyRate', value:'6.0'))
```

```
label('Term (Years):')
textField(text: bind(
  target:model, 'months', value:'30'))

label('Monthly Payment (P&I) :')
textField(editable:false,
  text: bind(source: model, sourceProperty: 'payment'))
```

The first problem you run into is that the payment field doesn't automatically update when the user changes the editable values. This is because the `payment` property of the model isn't an observable property. The simplest solution is to have the payment field update when any of the other fields are updated:

```
label('Monthly Payment (P&I) :')
textField(editable:false,
  text: bind(source: model, sourceProperty: 'payment',
    sourceEvent: 'propertyChange'))
```

The next problem is that the data formats of the fields don't match those of the model. This is one of the primary uses of converters: converting one type of data to another type. With a float conversion, the `principal` text field should now look like this:

```
label('Principal:')
textField(text: bind(target:model, 'principal',
  value:'330000',
  converter: Float.&parseFloat))
```

Once the other input fields have the same converter, you should begin getting updates to the payment field. The payment values will be completely nonsensical at this point, with a payment in excess of the initial mortgaged amount. The problem (which the astute reader may have seen coming miles away) is that the formula is expressed in different units than the user expects: months and a fractional monthly rate versus years and an annual percentage. This is the second primary purpose of converters: to massage data values before they're set in the model.

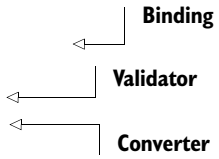
The next question is to decide where to put the logic for validating and converting the results. There's no absolute answer. But because the logic can be somewhat arbitrary, it sounds like it should be stored in the controller. With closures stored in the model, the code for the converters and validators looks something like this:

```
label('Interest Rate:')
textField(text: bind(target:model, 'monthlyRate',
  value:'6.5%',
  validator: controller.validateRate,
  converter: controller.convertRate))
```

Finally, you don't want to visually cram all the fields together. The fields also shouldn't hug the edge of the frame. There are many ways to do this, but for this example you'll use an empty border and a grid layout with vertical and horizontal padding. After putting all the pieces together, you get the final view for the application, as shown in the next listing.

Listing 3.17 MortgageCalcView.groovy

```
application(title:'Mortgage Calculator', pack:true, locationByPlatform:true)
    {
  panel(border: emptyBorder(6)) {

    gridLayout(rows:4, columns:2, hgap:6, vgap:6)

    label('Principal:')
    textField(text: bind(target:model, 'principal',
      value:'$200,000',
      validator: model.validatePrincipal,
      converter: model.convertPrincipal))

    label('Interest Rate:')
    textField(text: bind(target:model, 'monthlyRate',
      value:'6.5%',
      validator: model.validateRate,
      converter: model.convertRate))

    label('Term:')
    textField(text: bind(target:model, 'months',
      value:'30',
      validator: model.validateTerm,
      converter: model.convertTerm))

    label('Monthly Payment (P&I):')
    textField(editable:false,
      text: bind(source: model, sourceProperty: 'payment',
        sourceEvent: 'propertyChange',
        converter: model.convertPayment))
  }
}
```



The final task is to write what constitutes the validation logic.

3.6.3 Adding the missing validations to the model

In this example, the model holds all pertinent functions that affect its own values. In other applications, you might feel like placing the validators in a controller, which is OK, too. One reason for choosing the model instead of the controller is multithreading concerns. For the moment, we ask you to trust our judgment; all will become clearer in chapters 6 and 7.

There are laws restricting how long a mortgage can be held, how much interest can be charged, and, for some loans, how much principal can be borrowed. These decisions should in no way affect the visual representation, so they shouldn't be in the view. The other option is to place them in the model, and this is a decision between a concise model and a verbose model. For this example, you'll choose a concise model and place the constraints in the controller (see the next listing).

Listing 3.18 Updated `MortgageCalcModel` with validation and conversion logic

```
import groovy.beans.Bindable
import java.text.NumberFormat
import java.text.DecimalFormat
```

```
@Bindable
class MortgageCalcModel {
    float principal
    float monthlyRate
    float months

    float getPayment() {
      return principal * monthlyRate /
        (1-Math.pow(1/(1+monthlyRate),months))
    }

    private currencyFormat = NumberFormat.currencyInstance
    private percentFormat = new DecimalFormat('0.00%')

    def validatePrincipal = {                    <-- Principal validator
        try {
            float principal = currencyFormat.parse(it)
            return principal > 0
        } catch (Exception e) {
            return false
        }
    }
    def convertPrincipal = currencyFormat.&parse     <-- Principal converter

    def validateRate = {
        try {
            float rate = percentFormat.parse(it)
            return rate > 0 && rate < 0.30
        } catch (Exception e) {
            return false
        }
    }
    def convertRate = {
        return percentFormat.parse(it) / 12
    }

    def validateTerm = {
        try {
            def term = Float.parseFloat(it)
            return term > 0 && term < 100
        } catch (Exception e) {
            return false
        }
    }
    def convertTerm = {
        return Float.parseFloat(it) * 12
    }

    def convertPayment = {
        return currencyFormat.format(it)
    }
}
```

The model code also contains another often-overlooked aspect of simple GUIs like this one: ease of use. By using the currency and percentage formats from the `java.text` package, you present the numbers to the user in a format that more easily matches their common usage: with currency symbols, comma-separated number groups, and a

percentage sign for the rate. One nice side effect of using the percentage format is that it automatically converts the percentage values to fractional values. Earlier drafts of the code did this conversion by hand.

3.7 Summary

The model is a collection of data that exists to be shown and changed by the view and the controller. But in order to make the process flow smoothly, the view needs to have its data fed to it automatically. To do so, you mark the properties that you want to be observable with the `@Bindable` annotation (or use the handy `ObservableMap` class), and you find the places in the view class that need to be updated when properties in the model are updated. The bind node can be used directly on the attributes of the node or can be declared outside of the view tree.

When declaring a binding operation, it's essential that the three main pieces of the bind be declared in some fashion: what triggers the update, what provides the value, and where the value is placed. For simple property-to-property bindings, these are provided by the properties themselves (assuming that the property providing the value has been marked `@Bindable`). More advanced techniques trigger updates from JavaBeans events and provide source values from arbitrary closures. The result of the binding must go into a JavaBeans property. Finally you can do powerful things with the objects the Griffon framework creates to track bindings.

If we were magicians, we would be out of work by now, because we've laid bare some of the best parts of the magic trick that is data-model binding. Whenever you see a well-bound MVC group in action, it will still look magical. You'll know how things work, but you won't have to worry about the details. It's the magician who has to wash the rabbit fur out of his hair every night, not the spectators!

With models and bindings in your bag of tricks, it's time to move on to views.

4 Creating a view

This chapter covers

- A brief introduction to Swing
- The basics of a Griffon view
- Composing views with legacy source code

Views in Griffon are responsible for composing the visuals aspects of your application. Views are what the user interacts with. There are hundreds of components that you can use to create a view; and no matter which one you choose, you can compose the view in the same manner via a specialized DSL based on Groovy's SwingBuilder.

Griffon views can also be composed of other views, resulting not just in clever reuse of code but also in a way to display new elements on the fly.

In this chapter, we'll take a deeper look at Griffon views. We'll start by examining a classic Swing example and then compare it to a Griffon example that implements the same functionality. After you've see how much easier Griffon is, we'll look at special features of Griffon views and how you can use these features to keep your code organized when you build a large application. We'll end this chapter by showing how you can integrate views built with NetBeans GUI builder (formerly Matisse) and Abeille Forms Designer into a Griffon application.

We're about to enter the wild yet amazing world that is Java Swing, If you're not familiar with Swing, we encourage you to see *Swing*, 2nd edition (Manning, 2003) for a thorough primer. This chapter assumes you have a fair understanding of Swing.

We'll start by briefly reviewing Swing before digging into the Groovy SwingBuilder, just to cover the basics. If you're familiar with Swing concepts, feel free to jump to section 4.2 and get started with SwingBuilder. And if you're already comfortable with SwingBuilder, you can take the fast path to section 4.3 to learn more about the anatomy of a Griffon view.

4.1 Java Swing for the impatient

The basic premise of Swing is that the UI is a hierarchical tree structure that's the result of component composition. We're using the word *component* to mean any Swing/UI object. Components can be further divided into containers and plain components. A container can contain other containers or plain components. A plain component doesn't contain any other components. Typical containers in Swing are windows, dialogs, menus and panels; buttons, labels and menu items are examples of plain components. This isn't a complete list of the available Swing containers and components, but it's enough to get you started. Figure 4.1 illustrates the Swing components of a simple "Hello World" application.

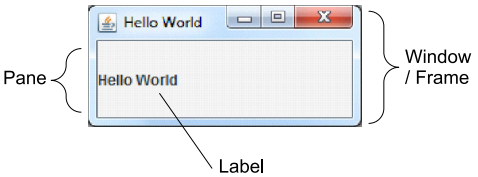


Figure 4.1 Swing containers and components

Containers and components work together through a parent-child relationship. Containers usually handle how their components are visually arranged by means of a helper layout object. Figure 4.2 illustrates the parent-child nature of the component hierarchy for the "Hello World" application in figure 4.1.

Figure 4.2 "Hello World" component hierarchy

Notice that the Swing component hierarchy is a bit more elaborate than you may have imagined when looking at figure 4.1. The Swing UI toolkit defines a number of intermediate components and containers that take care of handling user events properly and changing the overall UI state. Take for example the glass pane in figure 4.2. When that component becomes active, it will block any further events from being sent to the components behind it—that is, the `JLayeredPane` and its child components. In other words, it serves as a shield. Developers normally choose this technique to signal that the application is in read-only mode until the current action is finished. A glass pane is where you'll usually see a waiting clock or an animated icon.

You can change any object and its properties, of course, and plenty of options exist to do so. There are even options that mimic the layout of a web page.

In this section, you'll get a feel for Swing by looking at the classic Swing "Hello World" application. You'll also extend the application to accept user input. We'll

complete this section by making some observations about Swing development. On with the show.

4.1.1 "Hello World" the Swing way

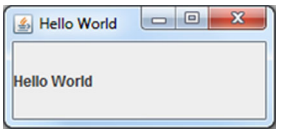


Figure 4.3 "Hello World" in plain Swing. It can't get much more straightforward than this.

You'll first create the classic "Hello World" in Swing. Nothing fancy, just a window that says "Hello World," as shown in figure 4.3.

The app is made up of a top-level component (a `JFrame`), which is a container that holds a plain component (a `JLabel`) that displays some text. The following listing shows the code for this simple example.

Listing 4.1 A bare-bones `HelloWorld` Swing application

```
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloWorld {
  public static void main(String[] args) {
    // unsafe Swing threading
    JFrame frame = new JFrame();
    frame.setTitle("Hello World");
    frame.setSize(new Dimension(200, 100));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(new JLabel("Hello World"));
    frame.setVisible(true);
  }
}
```

❶ Set properties on frame

❷ Create JLabel and add to frame

In case you didn't know already, Java Swing was designed around the time JavaBeans conventions[1] were laid out. It's no surprise that a Swing component can be configured by means of setting property values. Look what happens with the `frame` variable. An instance of `JFrame` is created using the default constructor. As is customary according to the JavaBeans conventions, a bean defines at least a no-args constructor. Then some of its properties are mutated ❶. A property, by definition, consists of a pair of methods that follow a naming convention: `T getProp()` and `set(T prop)`, where `T` stands for the property type, `getProp()` retrieves the property's value, and `setProp()` mutates the value accordingly. Finally a `JLabel` (a component capable of displaying text) is added to the frame's content pane ❷, and the frame is displayed.

This code should look familiar to those with a GTK (www.gtk.org), Qt (http://qt.nokia.com), or Standard Web Toolkit (SWT; www.eclipse.org/swt) background. It may even look similar to what you can do on the web with the Google Web Toolkit (GWT; http://code.google.com/webtoolkit/). This example drives home the point

[1] http://en.wikipedia.org/wiki/JavaBean.

Beware the thread

Listing 4.1 omits an important rule when working with Swing, one that's related to the JVM's multithreaded nature and Swing's single-threaded design. It's safe for sample code to omit this rule, because including the code that enforces it would result in longer examples that may obscure the point we're trying to make. We'll discuss threading issues in full detail in chapter 7. For now, we'll mark unsafe threading code with a comment.

that Swing applications are developed by means of composing a set of components that follow the JavaBeans conventions.

Let's dig a little deeper and make your simple application take some input and give you a response.

4.1.2 Extending "Hello World": "Hello Back"

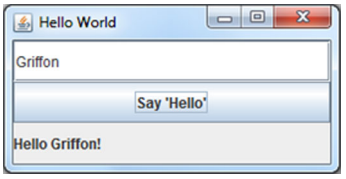


Figure 4.4 The "Hello Back" Swing application displaying the user's input after they click the button

You can extend listing 4.1 with two additional Swing concepts: layouts and event handlers. Instead of having a hard-coded message displayed on a label, you'll now prompt the user for a name. The application will respond using the user's input. Figure 4.4 shows the application running.

To implement this behavior, you need to do the following:

- Build an input field, a button, and a label
- Lay out the components in a meaningful manner
- Register an event handler on the button

The following listing shows the minimal code to make it work.

Listing 4.2 "Hello Back" application that reads user input and displays it in Swing

```
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class HelloBack {
  public static void main(String[] args) {
    // unsafe Swing threading
    JFrame frame = new JFrame();
    frame.setTitle("Hello World");
    frame.setSize(new Dimension(200, 140));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    final JTextField textField = new JTextField();
    textField.setColumns(20);
    final JLabel label = new Jlabel();
    JButton button = new JButton("Say 'Hello'");
```

1 **Create components you need**

```
    frame.setLayout(new GridLayout(3,1));
    frame.getContentPane().add(textField);
    frame.getContentPane().add(button);
    frame.getContentPane().add(label);

    button.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent evt) {
        String text = textField.getText().trim();
        if(text.length() == 0) return;
        label.setText("Hello "+ text +"!");
      }
    });

    frame.setVisible(true);
  }
}
```

❷ Apply layout

❸ Register event handler

Listing 4.2 fulfills the three goals. First you build the required input field, label, and button ❶. Notice that the variables that reference the input field and label are marked as final; this will become clear in a minute. Next you arrange the three components by changing the default layout used by the frame: instead of a `BorderLayout`, you set it to a `GridLayout` ❷ with three rows and one column. Then the mystery behind using the `final` keyword is revealed, when you set the required event handler on the button as an anonymous class ❸. As you may know, anonymous classes in Java can't reference a variable outside of their scope unless it belongs to its parent class or is marked as final. In this case, the variables are defined inside the same method that defines the anonymous classes, so you're forced to use the `final` keyword to make them visible. This is by no means the only way to implement an application with this behavior, but it's definitely the shortest.

Although this is just the basics of Swing, you should now feel more confident with it. Having a basic understanding of Swing will help you understand SwingBuilder and appreciate how much easier Swing development is when you use Griffon and SwingBuilder. Why do many developers complain about Swing if it appears to be so simple? Alas, Swing isn't without its kinks.

4.1.3 Swing observations

There are three factors working against Swing's good reputation, all of which will be explained next:

- *Java's verbosity level*—We've already established this point. Java is a great language, and Swing is a good UI Toolkit; put them together, and people go crazy, fast. So much code needs to be written to get them to work that not even an IDE can keep you from going mad.
- *No Java generics*—Swing was conceived and released long before generics made it into the Java language. Although some Java APIs received a facelift when generics were introduced (such as the Java Collections API), Swing was spared them. This means developers must continually check and cast objects returned by Swing classes. At the moment of writing the book, there are no plans in the foreseeable future to change this fact.

- *Difficulty of threading*—This is by far the most problematic issue with Swing. Threading is a hard task to tackle. It doesn't matter how well a Java library smooths out the two previously exposed factors, proper threading support can be the deal breaker.

Luckily, Griffon chose to use Groovy's SwingBuilder. SwingBuilder is a Groovy class for building hierarchical Swing structures. When used with the Groovy language, these structures provide a solution for each of the factors we've just mentioned, as explained in the next section.

4.2 Groovy SwingBuilder: streamlined Swing

SwingBuilder belongs to a group of helpful Groovy classes that follow the builder pattern.[2] The builder pattern is a commonly used technique for constructing complex object structures. The goal of the builder pattern is to encapsulate the construction details within the builder to provide the developer with an easier method of creating a complex object structure. SwingBuilder takes advantage of the Groovy syntax and lets you declaratively define a hierarchical structure with a small amount of code. Figure 4.5 illustrates SwingBuilder taking a declarative view definition to render the view.

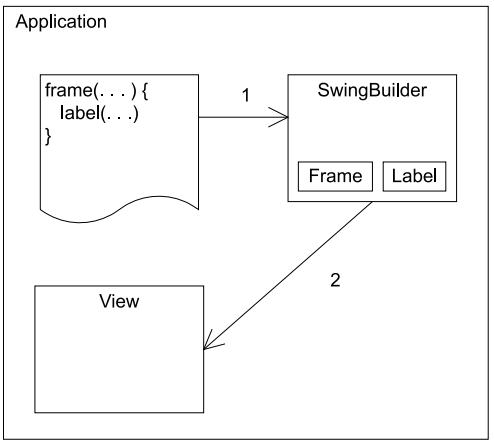


Figure 4.5 How Groovy SwingBuilder creates a view

Without going into too much detail for the moment, let's just say that Groovy's metaprogramming capabilities, its native map syntax, and its closures support are a great help in implementing a builder. You'll see all those features in action in a few moments.

In this section, you'll build and extend the same "Hello World" application as earlier, first using SwingBuilder and standalone Groovy scripts, then with Griffon. You'll see how much easier it is to do with Griffon and SwingBuilder than with plain Java. Because the builder understands the context and hierarchical structure, you can write significantly less code.

Let's tell the world hello with Griffon and SwingBuilder.

[2] http://en.wikipedia.org/wiki/Builder_pattern.

4.2.1 "Hello World" the SwingBuilder way

Let's revisit the "Hello World" application shown earlier in listing 4.1, this time implemented with SwingBuilder.

Listing 4.3 "Hello World" application implemented with Groovy's SwingBuilder

```
import groovy.swing.SwingBuilder
import static javax.swing.JFrame.EXIT_ON_CLOSE

new SwingBuilder().frame(title: 'Hello World',
  size: [200,100],
  defaultCloseOperation: EXIT_ON_CLOSE,
  visible: true) {
  label 'Hello World!'
}
```

❶ Unsafe Swing threading

The output is the same as that shown in figure 4.3. When running this application, you may notice that sometimes the label doesn't appear; this is a side effect of using unsafe Swing threading code ❶. You can manually change the size of the frame to force an update, and the label should appear. But if you're impatient to learn the proper way to build a Swing application with SwingBuilder while honoring the threading rules, wait no longer; the following listing provides the answer. SwingBuilder comes with a few methods that make threading a breeze.

Listing 4.4 "Hello World" with proper threading support

```
import groovy.swing.SwingBuilder
import static javax.swing.JFrame.EXIT_ON_CLOSE

new SwingBuilder().edt {
  frame(title: 'Hello World',
        size: [200,100],
        defaultCloseOperation: EXIT_ON_CLOSE,
        visible: true) {
    label 'Hello World!'
  }
}
```

Run in EDT (thread safe)

Remember that we'll cover everything you need to know about Swing and threading in chapter 7. For now, let's inspect the code in listing 4.4. If you want to, you can flip back to listing 4.1 and see how much the code has changed while still providing the same behavior.

First you need an instance of SwingBuilder; no surprises there. You use that instance to build a top-level component: a `JFrame`. You do so by invoking the `frame()` method on SwingBuilder. Note the parameters the method takes: `title`, `size`, `defaultCloseOperation`, and `visible` properties are available. It isn't coincidence that they resemble the properties you set in the Java version—they're the same properties! SwingBuilder uses the Groovy short map literal syntax. Whenever you see a method call in Groovy that looks like `method(param1: value1, param2: value2)`, make no mistake: Groovy will convert it into a map under the covers. The added benefit

with SwingBuilder, and builders in general, is that it will use the map's keys and values to set matching properties on the target object.

The second thing to notice is that SwingBuilder automatically converts a `List` into a `java.awt.Dimension`. This is another feature provided by Groovy, and it's why you don't need to import that class in the first place.

Finally, somehow the label is added to the frame, but there's no explicit call to do so—or is there? Notice that the `frame()` method takes an additional closure as a parameter; inside it, you'll find the label definition. The closure notation clearly conveys the idea that a parent-child relationship exists within the frame and the label; the builder knows that, and the label is automatically appended to the frame's children.

SwingBuilder reduces the amount of code you must write by allowing properties to be set without explicit calls to their corresponding setter methods. It's also aware of the current scope and context of a particular node; that's how it can embed child components on parent components without you having to make an explicit call to an add or register method on the parent.

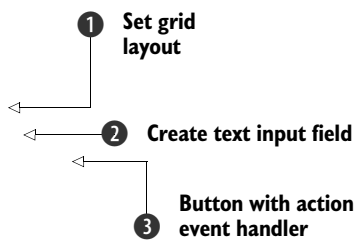
Now that you've seen the basic "Hello World" implemented with SwingBuilder, it's only fair that you do the same with the advanced version.

4.2.2 "Hello Back" with SwingBuilder

Again, you'll use SwingBuilder and Groovy to reduce the amount of code and visual clutter while retaining the same behavior as in listing 4.2. As we showed you with Swing, you'll enhance your simple app to prompt the user for a name in the next listing. The application UI will display that input.

Listing 4.5 Advanced "Hello World" with SwingBuilder

```
import groovy.swing.SwingBuilder
import static javax.swing.JFrame.EXIT_ON_CLOSE

new SwingBuilder().edt{
  frame(title: 'Hello World',
        defaultCloseOperation: EXIT_ON_CLOSE,
        size: [200, 140], visible: true) {
    gridLayout(cols: 1, rows: 3)
    tf = textField(columns: 20)
    button("Say 'Hello'", actionPerformed: {
      String text = tf.text.trim()
      if(text) lbl.text = "Hello ${text}!"
    })
    lbl = label()
  }
}
```



NOTE Notice the that `edt{}` is used to enable proper threading.

Let's inspect listing 4.5 part by part. It builds a frame just as listing 4.4 did, so you're on the right track. The new bits are the frame's layout and its children definitions. The layout is defined by using the `gridLayout()` node ❶, which in comparison to its

Java counterpart makes explicit that it has one column and three rows. We intentionally switched the order to highlight the fact that Groovy's map literal syntax when used with method calls effectively turns itself into named parameters.

The next element is the input field ❷, whose value is stored in a local variable named `tf`. Next is the button ❸ along with its event handler. Groovy is able to coerce a closure into an implementation of a single method interface, such as `ActionListener`. Add up the fact that Groovy closures accept a default parameter when none is defined, and you get a recipe for short and concise code. But you can define a parameter for the event handler closure, even its type, as shown in the following equivalent snippet:

```
button("Say 'Hello'", actionPerformed: { ActionEvent evt ->
})
```

It's convenient to define an inline `ActionListener` using the closure notation mentioned earlier, especially when the behavior invoked is small. But as you'll soon discover, it's better to define the closures elsewhere (such as in a controller) to boost readability and enforce code reuse.

The final element is the label, which is saved with a variable named `lbl`. You might wonder how it's possible for the button to reference the label when such a component hasn't been defined yet. Well, the label is available by the time the event handler is called; the difference is that the code associated with the event handler is executed not when the builder is assembling the node but rather when the application is up and running and the button is clicked. All local variables, such as `tf` and `lbl`, are stored within the context of the builder. The builder is also set as the delegate of the action closure. This is why you can access both variables inside that closure.

There you have it. Groovy's terse syntax coupled with SwingBuilder allows you to write an application with half the code it would take with regular Java. The code turns out to be more readable and expressive. And the application's behavior stays the same.

But what do all these things have to do with a Griffon view? We'll answer that question in the following section.

4.3 Anatomy of a Griffon view

In earlier chapters, we've shown you a few applications that rely on well-established conventions. A subset of those conventions specifies how views can be created, but we haven't fully explained them— until now.

In an application, the view(s) is the portion of the application that a user interacts with to fulfill some goal. It's what the user sees. Views are used to display information and take user input.

The last section gave you an inkling of what happens behind the curtains when Griffon builds a view. Yes, SwingBuilder plays a key part in the process; but as you'll soon find out, there's more to builders and views than meets the eye.

In this section, we'll look at the role of builders in the creation of views. You'll see that builders are made of nodes, and we'll show you what nodes can do.

4.3.1 Builders are key to views

At first glance, a view script is like any other Groovy script, with the peculiarity that it runs under a specific context: an instance of SwingBuilder. SwingBuilder is always available as a delegate, meaning all nodes and variables can be resolved against that particular SwingBuilder instance. That's half true; more accurately, a view script runs under an instance of `CompositeBuilder` rather than SwingBuilder.

We mentioned `CompositeBuilder` in chapter 2 when we discussed Builder.groovy;[3] that file contains the blueprints for creating a `CompositeBuilder`, which is a special class that helps build a view. Let's review the contents of a typical `griffon-app/conf/Builder.groovy` file:

```
root {
    'groovy.swing.SwingBuilder' {
        controller = ['Threading']
        view = '*'
    }
}
root.'SwingGriffonAddon'.addon = true
```

This file contains two builder definitions. The first definition tells the `CompositeBuilder` that an instance of SwingBuilder must be used. All nodes and methods will be appended to the view, whereas all nodes and methods related to the `Threading` group will be appended to controllers. By *append,* we mean the Griffon runtime will use metaprogramming to extend the behavior of the application's classes. You know this because the `view` property has `*` as its value, whereas the `controller` property has a list made up of a single element that happens to be the group containing all threading-related methods.

In addition to `*`, you can use the values listed in table 4.1. These values have more specific meanings. The properties that can be used as targets are the names of each MVC member belonging to a group. You saw `view` and `controller` already, but know that there may be additional members, such as `model`.

Table 4.1 List of acceptable values per target

Value	Effect
`*:factories`	Contributes all node factories only
`*:methods`	Contributes all explicit methods
`*:props`	Contributes all explicit properties
`*`	Contributes all nodes, methods, and properties

SwingBuilder defines a number of node groups, usually paired by behavior or similar characteristics. You've seen `Threading`, but there are also `Windows`, `TextWidgets`,

[3] Don't forget your Griffon ABCs: application, builder, config.

Containers, and Binding node groups, to name a few. Please refer to SwingBuilder's javadoc to learn more about the currently available groups.

The second builder definition instructs the CompositeBuilder that it must load a runtime plugin (or, as we like to refer to it, addon) whose name is SwingGriffonAddon; all of the addon's nodes will be contributed to views automatically. Addons are Griffon's answer to extensibility. By applying an addon to an application, you can extend its functionality. You'll see how it's done in chapters 11 and 12.

Adding new builders to the application's configuration is easy. Append a few lines resembling the SwingBuilder line you just saw. For example, you can embed JideBuilder (http://griffon.codehaus.org/JideBuilder) in this manner:

```
root {
    'groovy.swing.SwingBuilder' {
        controller = ['Threading']
        view = '*'
    }
    'griffon.builder.jide.JideBuilder' {
        view = '*'
    }
}
root.'SwingGriffonAddon'.addon = true
```

As an alternative, remember that Builder.groovy is a Groovy view of a properties file. You can also append the following line to the default Builder.groovy file:

```
root.'griffon.builder.jide.JideBuilder'.view = '*'
```

Whichever option you pick, you must remember to place all of JideBuilder's libraries and dependencies in your application's lib directory; otherwise you'll get exceptions while compiling your application.

> **Get the plugin**
>
> Adding builders to your builder configuration can be automated via plugins. All the builders shown at http://griffon.codehaus.org/Builders have companion plugins that do just that. We'll discuss how you can make your own plugins in chapter 11. We'll also provide plenty of information on how views can be extended via builders in chapter 12.

As you can see, configuring the CompositeBuilder component requires no dark magic at all.

Now that you understand where nodes come from, you can put them where they belong: in a view script.

4.3.2 Nodes as building blocks

Initially, all SwingBuilder nodes are available to be used in a view. SwingBuilder comes with a lot of useful nodes from the start. We've already mentioned the rule that you

obtain the name of a node based on the name of a Swing class: drop the first *J* from the name and lowercase the following character. Thus `JButton` becomes `button`, `JTextField` becomes `textField`, and so on.

Nodes not only instantiate a particular `JComponent` (in the case of SwingBuilder nodes) but also let you change an instance's properties using property syntax. No more lengthy calls to setter methods. Nodes are also aware of their surroundings. You've seen how a parent node knows when and how a child node must be appended. Most nodes use their nested closure as the source of child nodes. Others are smarter and use the closure in other ways.

For example, you've seen the combination of `action` and `button` nodes. What you might not know is that the `action` closure can be defined in place in the action node. No, we're not talking about setting the closure as the value of the action's `closure` property, but rather about defining it as a nested closure on the action node itself. This is how it's done:

```
action(id: "clickAction", name: "Click me!") { evt ->
    println "You clicked on ${evt.source.class.name}"
}
button(clickAction)
```

Clicking the button results in the following being printed to the console:

```
You clicked on javax.swing.JButton
```

There are other nodes that rely on this technique. We'll cover registering nodes in chapter 12.

Using preconfigured variables

It's important that you become familiar with all of the preconfigured variables that are available to you in a view script, because many times you'll need to lean on them to get the results you want:

- `app`—Every member of an MVC group has access to a variable named `app` that points to the instance of the currently running application. It's of type `griffon.core.GriffonApplication`.
- `controller`—If the current view has a controller associated with it, then this variable points to that instance. It's possible to have a view without a controller instance assigned, or even to have a `FooView` with a `BarController`. It all depends on how you configure your application.
- `model`—The `model` variable points to a model instance. This variable may be null for the same reasons laid out with the previous variable. But most of the time, both `model` and `controller` point to valid instances, which also match the view's logical group. This is the behavior as per the default configuration of MVC groups upon creation.

We'll cover in detail how MVC groups can be configured when we reach chapter 6.

In addition to preconfigured variables, which Griffon defines to make your job easier, you also have access to a number of extra nodes that help you create custom components.

4.4 Using special nodes

More Swing components exist than those found in the JDK, but it would be a huge task to hunt them all down and make a SwingBuilder node for each one for them. To solve this problem, SwingBuilder includes a few extra nodes that aren't related to a particular Swing class, which can come in handy when you need to insert a custom Swing component.

Before we get into the subject, remember that each SwingBuilder node accepts a value of the same type as the node, or a subclass. For example, you can use a label node as follows:

```
label(new JLabel(), text: "This is a label")
label(new MyCustomJLabelClass(), text: "Custom text")
```

This snippet assumes `MyCustomJLabelClass` is a subclass of `javax.swing.JLabel`. This fact is important because the special nodes we're about to discuss also rely on it.

In this section, we'll look at five special nodes: `container`, `widget`, `bean`, `noparent`, and `application`. We'll examine what they do and how to use them. Let's get started by looking at `container`.

4.4.1 Container

The `container` node lets you embed any `JComponent` instance into the hierarchy without restrictions. You may append child nodes to it by defining a nested closure. You may set properties on the instance as you would with any other node. The following are valid usages of the `container` node:

```
container(new JTextArea(), cols: 20, rows: 10)
container(new MyCustomJComponent(), name: "componentName") {
    button("This is a button")
}
```

As it turns out, this node functions like a pass-through node for any `JComponent` instance. The first line declares a `textArea` that is 20 columns by 10 rows. It's identical to

```
textArea(cols: 20, rows: 10)
```

If you're uncertain about the superclass of the object you'd like to embed, `container` is your best shot.

4.4.2 Widget

The `widget` node works similarly to the `container` node with a single difference: it doesn't support nesting of children. This means that although you can embed a `JPanel` instance with it, you can't attach any child components—you'll get an exception if you attempt to do so. The following snippets show sample usages of this node:

```
widget(new JPanel())
widget(new JTextArea(), cols: 20, rows: 10)
widget(new JPanel()) {
    borderLayout()
}
```

In this case, the first two uses of `widget` are fine, and the third fails.

If `widget` is more restrictive than `container`, why have it in the first place? Well, to signal that the node you're embedding is essentially a leaf node and should by no means contain any nested content.

4.4.3 Bean

The `bean` node can be seen as an über version of `container` because it lets you embed any bean instance—it doesn't matter if it's an instance of `JComponent`. This fact has important implications. If the instance you set as the node's value is indeed a `JComponent`, then the `bean` node works the same as `container`: it's a pass-through node, and the instance is added into the hierarchy. But if the instance isn't a `JComponent`, it won't be embedded into the hierarchy.

You can take advantage of that fact to bind to and from model properties using the short bind syntax. Here is an example of what it would take to bind a `textField`'s `text` property to a `name` property on the model:

```
textField(text: bind(target: model, targetProperty: "name"))
```

Now, using the `bean` node as a pass-through for the model instance, you get the following snippet:

```
textField(id: "textSource")
bean(model, name: bind { textSource.text })
```

Arguably, you added a new line of code, but this version will come in handy the next time you'd like to bind a source to at least two targets.

This node is also useful to configure any instance using properties. It's as if the instance had its own node, but without a specific name tied to its class.

4.4.4 Noparent

This node has a funny name, but it does exactly what it says. `Noparent` is a node that accepts child content, but that child content isn't embedded into the hierarchy. Why is this useful? Well, given that any `JComponent`-based node inserts its return value into the hierarchy immediately, you won't be able to manipulate a reference of a `JComponent` to tweak its properties without inserting it into the hierarchy at that point. Neither `container`, nor `widget`, nor `bean` will help you. But `noparent` will.

Let's look at an example. The following snippet shows what happens when you tweak a button inside a panel that has a `borderLayout()` set:

```
panel {
    borderLayout()
    label("contents")
```

```
    button(aButtonReference, text: "New Text")
}
```

Instead of the expected result shown in figure 4.6, you the result shown in figure 4.7: the button has replaced the label!

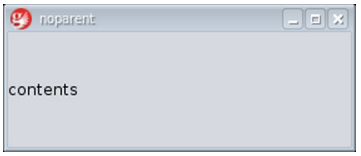


Figure 4.6 Expected result

If the button tweak is surrounded with a `noparent` node, then you'll get the expected result—in other words, you'll get figure 4.6. This is how you can do it:

```
panel {
    borderLayout()
    label("contents")
    noparent {
        button(aButtonReference, text: "New
     Text")
    }
}
```

Figure 4.7 Actual result

Easy as pie. We need to talk about one more node.

4.4.5 Application

The last node we'll discuss is the `application` node. This one doesn't belong to SwingBuilder per se, but to `ApplicationBuilder` (one of Griffon's internal builders).

As we mentioned in chapter 1, this node shields you from the implementation used to construct a main frame for your application. Whether your application is running in standalone or applet mode, this node makes sure to create the appropriate top-level container for you. In standalone mode, the container is a `javax.swing.JFrame`; in applet mode, it's a subclass of `javax.swing.JApplet`.

But you can change the underlying implementation to be used in standalone mode. Recall from chapter 2 that the `Application.groovy` config script holds the configuration of all MVC groups plus a few other properties pertaining to the application. One of those properties controls the class to be used as the top-level container. You can, for example, choose to use a `JRibbonFrame` (http://flamingo.dev.java.net) from the Flamingo Swing components suite.

That's all for now regarding special nodes. Although Griffon goes a long way toward making desktop development easier, in a large application the code can still get unorganized. In the next section, we'll discuss how to organize the code and manage your view scripts and their helper classes or scripts.

4.5 Managing large views

By now, you know Griffon takes care of creating a default view script every time you create a new MVC group. You also know that a view script is the place to define all UI elements by means of the SwingBuilder DSL. Armed with this knowledge, you may build your UI to a point that the view script is too big or no longer maintainable. What can you do in this case?

In this section, we'll discuss a few approaches to view management. We'll start with a tried-and-true approach: reusable code.

4.5.1 Rounding up reusable code

Remember that a view script is like any other Groovy script. This means you can refactor reusable code into classes or closures. If you choose to create classes, you have absolute freedom to choose where to put them. You can leave them in the same directory as your views, you can put them under `src/main`, or you can put them in a new directory of your choosing inside `griffon-app`. The important thing is that the compiler will find your sources and compile them along with your views. But we certainly recommend that you stick with the conventional locations, because doing so will make it easier for other developers who join the project to understand how the code is laid out.

If you choose to place reusable code in the form of closures, make sure those closures are defined before they're used, and be sure to omit the `def` keyword. If you use `def`, the closure becomes a method definition on the script; this will change the delegate used on the closure and result in unexpected behavior.

If your view script is still too big, consider the next alternative: additional scripts.

4.5.2 Breaking a large view into scripts

Another option you have at your disposal is breaking a view script into smaller scripts. The trick is figuring out a way to embed the components provided by a secondary script into the primary one. Enter the `build()` method.

The `build()` method takes a `Class` or a `Script` instance, evaluates its contents, and, in the case of a `Script`, returns the last expression evaluated. Let's see how you can take advantage of this.

Meet SwingPad

Griffon bundles a few sample applications that showcase the framework's features and strengths. One of these applications is called SwingPad. You can think of SwingPad as a view script–oriented version of `groovyConsole`. Figure 4.8 shows the outcome of running a small view script in SwingPad.

The SwingPad UI uses the screen real estate to display the following elements: a menu bar, a toolbar, the main content, and a status bar. If you were to implement this UI in a single view script, it might look like the following listing.

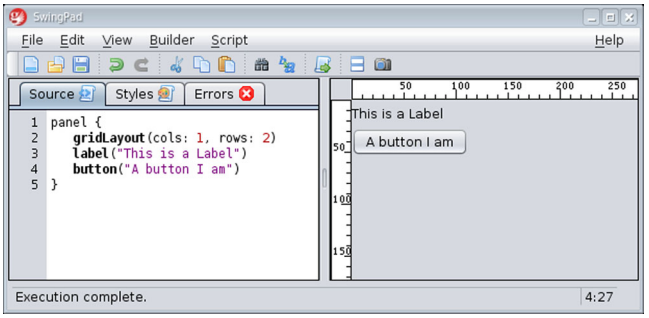
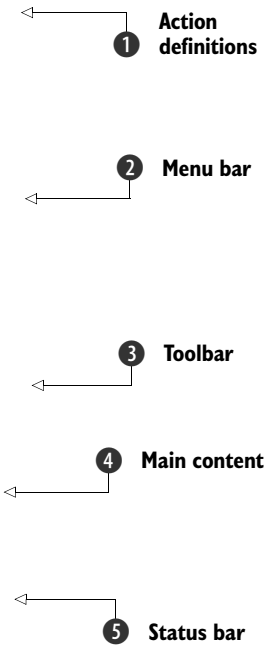


Figure 4.8 SwingPad running a small view script. The right side is the outcome of the code in the editor on the left side.

Listing 4.6 Brief layout of SwingPadView.groovy before breaking it up

```
actions {
   action(id: "openAction",
   name: "Open...",
   closure: controller.fileOpen)
   . . .
}
application( ... ) {
   menuBar {
      menu "File" {
          menuItem openAction
          . . .
      }
      . . .
   }
   toolBar {
       button(saveAction)
         . . .
   }
   splitPane {
       panel { /* code editor goes here */ }
       panel { /* view output goes here */ }
   }
   statusBar {
      . . .
   }
}
```



Phew! That's a lot of code. Imagine if you reprinted `SwingPadView`'s entire contents with the single view script approach—you can bet it would take a lot of pages. And that's precisely the problem with keeping the code in a maintainable state: it's too big to fit in a single editor screen. But you can appreciate the structure of listing 4.6. Swing actions reused across the application are denoted at ❶ (you can see a glimpse of them at ❷ and ❸). And there are clearly code areas that match the UI elements previously identified (❷, ❸, ❹, ❺).

But you can do better than this.

CREATING SMALL SCRIPTS

What if you put each UI element (including the action definitions) into its own script, and leave `SwingPadView` with the responsibility of piecing them together? This is where the `build()` method shines. The next listing shows a revised version of `SwingPadView` in which every element has been relocated to its own script.

Listing 4.7 Revised `SwingPadView`

```
build(SwingPadActions)
application(...) {
   menuBar build(SwingPadMenuBar)
   toolBar build(SwingPadToolBar)
   widget build(SwingPadMainContent)
```

```
    statusBar build(SwingPadStatusBar)
}
```

That's more manageable, wouldn't you agree? But there's a catch. Notice that the UI element scripts are embedded directly using a node after being built. This means each script must return a component that matches the expected type; otherwise the host frame won't know where to place them. You may notice in the SwingPad codebase that some scripts make an explicit return, whereas others don't; for example, `SwingPadMenuBar` returns a reference to the `MenuBar` node that was built, but `SwingPadToolBar` doesn't. There's a simple explanation: remember that views are also Groovy scripts. The return value of a Groovy script is the last expression that was evaluated. In the case of `SwingPadMenuBar`, additional elements are being instantiated after the `menuBar` node, which is why you must explicitly return a reference to the `menuBar`; on the other hand, in `SwingPadToolBar`, the last expression evaluated was the toolbar node itself—no need for an explicit return value in this case. The exception to this rule is the main content pane, given that all other areas will be embedded in the appropriate place due to their types.

You may wonder what happens with action variables defined in the `actions` script. How can you reference them from other scripts? The answer is that all the scripts share the same SwingBuilder instance as delegate. They also share the same binding. Thus any variable tied to the builder can be seen from other scripts. The same principle applies to any variables (including your own closures) found in the binding. This is the sole reason why we suggested that you drop the `def` keyword when defining reusable blocks of code in the form of closures.

There is yet another alternative for splitting up your code base to keep it nice and tidy.

4.5.3 Organize by script type

In the previous section, you saw how to split `SwingPadView` from a single script into a primary script and five secondary ones. One of those scripts is named `SwingPadActions` and contains all the action definitions. Now imagine what would happen in a bigger application where each view had a companion `actions` script. Doesn't it seem as though actions belong to their own type like views do? Is there something you can do about this?

In chapter 6, we'll explain how to compose an MVC group using configuration. As a preview, you can alter the number and types of an MVC group.

For illustration purposes, let's say you want to add an `actions` element to Swing. Begin by creating a new directory called actions under griffon-app:

```
$ mkdir -p griffon-app/actions/griffon/swingpad
```

Place all your action scripts in that directory. Next, edit the application's configuration file. Remember which file it is? *A* is for application, so go to `griffon-app/conf/Application.groovy` and search for the definition of the groups you'd like to tune. For example, you may find the following definition in SwingPad:

```
SwingPad {
    model      = "griffon.swingpad.SwingPadModel"
    view       = "griffon.swingpad.SwingPadView"
    controller= "griffon.swingpad.SwingPadController"
}
```

All you have to do is insert a new member definition that follows this convention: the name of the member matches the name of the directory. The next snippet shows how the SwingPad MVC group would look if you added the new member:

```
SwingPad {
    model      = "griffon.swingpad.SwingPadModel"
    actions    = "griffon.swingpad.SwingPadActions"
    view       = "griffon.swingpad.SwingPadView"
    controller= "griffon.swingpad.SwingPadController"
}
```

Doesn't it look like the `actions` member belonged there from the start? You can do the same for any other groups. Be careful with the member-definition order, because actions are required by the view. By placing the `actions` definition before the `view` definition, you instruct Griffon that the `actions` member must be initialized before the `view` member.

We're almost finished, but we have one last topic to discuss in this chapter: how to deal with legacy views.

4.6 Using screen designers and visual editors

We've discussed several ways to create s UI using the SwingBuilder DSL. But you may not be able to build a new application from scratch every time. Sometimes it's best to reuse an existing view, which may rely on external libraries and/or layouts.

In this section, we'll show you step by step how to integrate legacy views that depend on popular designer tools and libraries. Specifically, we'll look at how to integrate NetBeans GUI builder– and Abeille Forms Designer–based views into a Griffon MVC group.

4.6.1 Integrating with the NetBeans GUI builder (formerly Matisse)

About the time of JavaOne 2006, a new layout for Swing was announced: the SwingLayout. Also known as GroupLayout, this layout and its related helpers belong to a larger project within the NetBeans platform called the NetBeans GUI builder (formerly known as Project Matisse). This UI designer tool lets you design a UI using drag-and-drop and property-editing techniques. It has become popular with many Swing developers.

Although UI elements created with the NetBeans GUI builder are easy to update using the tool, the same can't be said if you choose to do it in a programmatic way. Simply put, GroupLayout was designed with tool support in mind. Porting a GroupLayout-based panel or frame is hard to accomplish by hand—it's best to leave the work to another tool. That's right, we're talking about the Griffon command line.

You may have noticed a peculiar command name when you typed `griffon help`: the `generate-view-script` command. This script is able to read GroupLayout definitions and export them into a SwingBuilder friendly script. From there you can apply all the SwingBuilder techniques you already know

Let's see an example of the script's usage.

A NetBeans GUI builder view

Assume for a moment that your company has a standard set of dialogs and frames that need to be used in every application. One such component is a Login dialog, which may look like figure 4.9.

Granted, it looks a bit sparse, but it gets the job done. If you feel like creating the same dialog in NetBeans GUI Builder, then please do so; we'll be here waiting for you. But remember, before you continue, copy the code for this dialog into your Griffon application's source directory. Either `griffon-app/views` or `src/main` is a good place. Alternatively, you can drop a jar containing the compiled classes into your application's lib directory. On to the next step.

Generating the view script

Now comes the easy part. `generate-view-script` takes a single argument: the full qualified class name of your legacy view. If you don't define an argument, the script

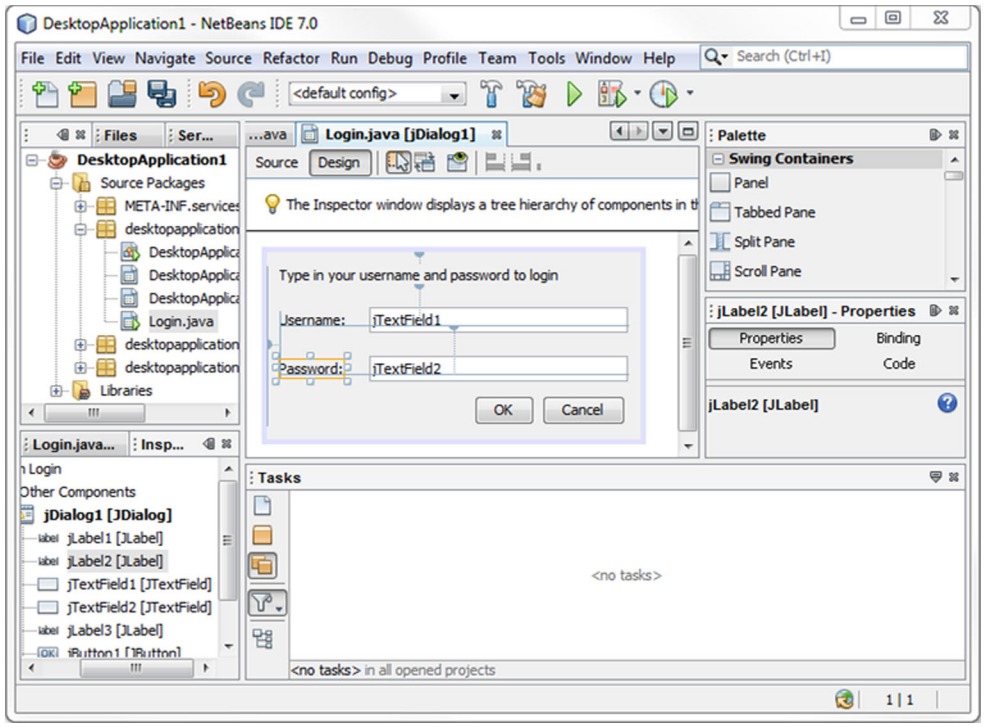


Figure 4.9 A basic legacy Login dialog. This dialog was created using the NetBeans GUI builder visual designer.

will prompt you for one. Before you proceed, make sure you've copied the jar that contains `GroupLayout`'s classes into your lib directory. If you have NetBeans installed, it will be located in `modules/ext` under the name `appframework.<version>.jar`. Assuming the name of class you want to include into your application is `LoginDialog`, the following command invocation should do the trick:

```
$ griffon generate-view-script LoginDialog
```

After the command has finished, you should see a new script named LoginDialog-View.groovy located in griffon-app/views. The script looks like the following listing.

Listing 4.8 Generated LoginDialogView.groovy script

```
widget(new LoginDialog(), id:'loginDialog')          <-- (1) Main UI container

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
```

Well, well. Isn't this interesting? Notice that the generated script uses the special nodes we discussed back in section 4.4. The main component described by the view is embedded into the script (1) using the `widget` node. You can change it to `container` if you intend to make additional tweaks on the component's children. Next you see the usage of the `noparent` node in conjunction with the `bean` node to define references to the dialog's contents. If you didn't use the `noparent` node, the dialog's contents might be inserted in the wrong place in the node hierarchy.

BINDING THE FIELDS

From here you can fine-tune the script, say by adding bindings between model and view, or by adding references to controller actions. The next listing shows an updated version of the generated script after adding a few binding and action references.

Listing 4.9 Updated LoginDialogView.groovy

```
widget(new LoginDialog(mainFrame, true), id:'loginDialog')
noparent {
    bean(loginDialog.usernameField, id:'usernameField',
         text: bind(target: model, targetProperty: "username"))   <-- Bind to model property
    bean(loginDialog.passwordField, id:'passwordField',
         text: bind(target: model, targetProperty: "password"))
    bean(loginDialog.okButton, id:'okButton',
         actionPerformed: controller.loginOk)                     <-- Assign action
    bean(loginDialog.cancelButton, id:'cancelButton',
```

```
            actionPerformed: controller.loginCancel)
}
return loginDialog
```

Now you need to wire this view into an MVC group.

Creating the Login MVC group

To create a new group, type the following at your command prompt:

```
$ griffon create-mvc login
```

This command creates a model, view, and controller tied to the `Login` group. You can safely delete `LoginView` because you already have a view—you just need to configure it. Open `Application.groovy`, and locate the definition of the login group. Table 4.2 shows the necessary configuration changes.

Table 4.2 Changing the view associated with the login MVC group

Original code	Revised code

`mvcGroups {`
`  'login' {`
`  model = 'LoginModel'`
`  view = 'LoginView'`
`  controller = 'LoginController'`
`  }`
`…`
	`mvcGroups {`
	`  'login' {`
	`  model = 'LoginModel'`
	`  view = 'LoginDialogView'`
	`  controller = 'LoginController'`
	`  }`
	`…`

`LoginDialogView` is now tied to the `Login` MVC group. All that's left to do is fill in the model and controller with the properties defined in listing 4.9.

Giving the Login dialog some behavior

To test whether the view is communicating with the controller, you'll have `LoginController` respond to input. If you enter a username and password, a message dialog appears with the data you entered (see figure 4.10). And if you cancel the dialog, a confirmation message appears (see figure 4.11).

The controller actions that implement this functionality are shown in the following listing.

Figure 4.10 The user entered `griffon` and `random_password` in the input fields of `LoginDialog`, and then clicked OK.

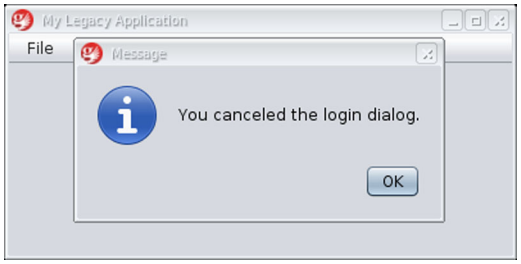


Figure 4.11 The user canceled `LoginDialog` by clicking Cancel.

Listing 4.10 Implementation of `LoginController`

```
import javax.swing.JOptionPane

class LoginController {
    def model
    def view

    def loginOk = { evt = null ->
        doLater {
            JOptionPane.showMessageDialog(view.mainFrame,
"""You entered the following information:
username: ${model.username}
password: ${model.password}
""".toString())
        }
    }

    def loginCancel = { evt = null ->
        doLater {
            JOptionPane.showMessageDialog(view.mainFrame,
                "You canceled the login dialog.")
        }
    }
}
```

Action results in figure 4.10

Action results in figure 4.11

You still need to create an instance of the login group and display the dialog. That's the responsibility of a controller, and we'll give controllers wide coverage in the next chapter. But if you can't wait any longer to solve the puzzle, remember what you did in your first Griffon application. That's right, you used a method called `createMVCGroup()`.

The next integration scenario we'll look at involves Abeille Forms Designer.

4.6.2 *Integrating with Abeille Forms Designer*

Abeille Forms Designer (http://java.net/projects/abeille/) is another great option for visually designing UIs. Abeille is an open source tool that comes with a WYSIWYG editor and serves as an abstraction over JGoodies FormLayout (http://java.net/projects/forms/) and the JDK's `GridBagLayout`.[4] Figure 4.12 shows how `LoginDialog` would look like had it been made with Abeille Forms Designer.

[4] See http://madbean.com/anim/totallygridbag for a funny look at `GridBagLayout`.

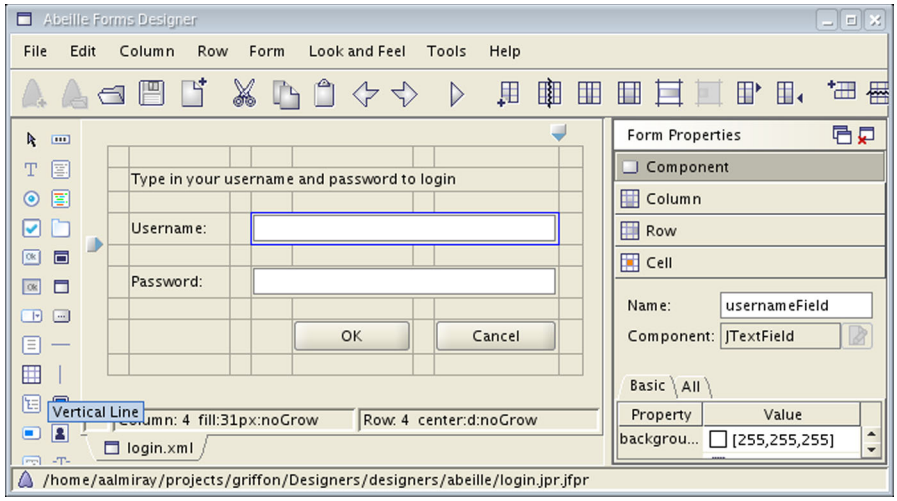


Figure 4.12 The `LoginDialog` panel edited with Abeille Forms Designer. Notice that the grid can be made explicit to guide you in placing the components.

We must tell you up front that there's no support for Abeille in the core framework, but you can still use its features if you install a plugin. (We'll cover all things related to plugins in chapter 11.) You can install the latest version of the Abeille Forms plugin by issuing the following command at your command prompt:

```
$ griffon install-plugin abeilleforms-builder
```

Once it's installed, you'll get a few new nodes that can be used in your view scripts, the most important of which is `formPanel`. This node can read a form's definition, preferably in XML format.

> **Plugins for adding new nodes**
>
> Installing plugins is another way to add new nodes to `CompositeBuilder`. A number of plugins do just that. We'll show you how to create such a plugin in chapter 11. We'll also show you how to interact with the most common plugins that provide new nodes in chapter 12.

Assuming you placed the login form definition (in XML format as exported by Abeille Forms Designer) under griffon-app/resources, you can then tweak `LoginView` as shown in the following listing.

Listing 4.11 `LoginView` configured with an Abeille Forms form

```
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)
    }
}
```

❶ Load and build form

❷ Tune form's components

There are a few differences from the NetBeans GUI builder example. The dialog definition is made explicit due to the form containing just a panel definition. You load and build the form based on its XML definition ❶. Then you fine-tune each form component by adding binding and action references ❷. Unfortunately, this code must be written by hand, but as you can see the availability of special nodes makes this task simple. As a matter of fact, the code resembles what you saw back in listing 4.9 after you fine-tune it.

Don't worry too much if you couldn't get these examples to run. After all, we didn't provide you with the code for the legacy forms! But you can compare notes and see the full source code in the book's source repository.[5]

4.7 Summary

This was a whirlwind ride into Griffon views. We started this chapter by giving a brief overview of the Swing toolkit. You wrote a simple Java Swing "Hello World" application and then had it say something back to you. You also used Groovy SwingBuilder to implement the "Hello World" application. You saw that Griffon does a nice job addressing the drawbacks of the Java Swing version, including threading.

Next we looked at the anatomy of a Griffon view. You saw that builders are key to creating views and that a builder's DSL makes it easy to create a component tree. Along the way, you learned that Griffon has special nodes to help you use custom and third-party components. We also looked at a couple of techniques for working with large, complex views. For those who prefer using screen painters, Griffon has that covered too: using the output from the NetBeans GUI builder and Abeille Forms Designer in a Griffon application is pretty easy using the special nodes.

You now have a good understanding of Griffon view. But a view by itself doesn't do much—doing something with user input is the role of controllers and services. In the next chapter, we'll take a dive into those components.

[5] http://github.com/aalmiray/griffoninaction.

5 Understanding controllers and services

This chapter covers:

- Controllers and their responsibilities
- Services
- Metaprogramming and inspection capabilities on artifacts

Wading through Griffon's MVC is quite a journey, but we're almost done reviewing what it has to offer. The final piece we'll look at takes care of routing all of the user's input to and from the appropriate handlers. We're talking about the brains of your application: the controller member of the MVC triad. In Griffon, this member is mostly represented by controllers; services are also used to a lesser extent.

Simply put, controllers have the responsibility to react to inputs, usually coming from the UI. Inputs may also come from other locations, such as a service or an application event (application events will be covered in chapter 8). But regardless of the input's source, the controller will most likely update the model, which in turn may update the view. Figure 5.1 illustrates the controller reacting to multiple inputs.

Controllers also have the ability to create and destroy other MVC groups, as you saw back in chapter 1. The `GroovyEditController` creates a new MVC group (`FilePanel`) for every tab that's required. The `FilePanelController` cleans up the group when the tab is disposed.

Controllers are positioned right in the middle between inputs and outputs, orchestrating who talks to whom and where inputs are routed, and thus interacting with many components. Given this, they're a good location to apply metaprogramming to components when such a need arises.

In this chapter, we'll discuss what makes controllers tick, their relationship to services, and the metaprogramming options you can apply to any application artifact.

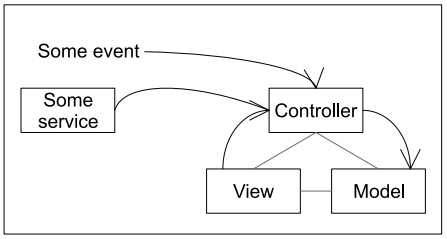


Figure 5.1 Controller receives stimulus from event, service, or UI then updates the model

What is an application artifact?

Artifacts are major building blocks of a Griffon application. You've already been using them. Out of the box, Griffon supplies the following artifacts: model, view, controller, and service.

Let's take a closer look at controllers and their parts.

5.1 Dissecting a controller

A controller, in its most basic form, can be thought of as a collection of action handlers. By now, you shouldn't be surprised when we tell you that Griffon adds a couple of convenience mechanisms to make working with controllers easy. Just like views, controllers have access to injected properties and methods that will make your job easier. In some circumstances, when a controller is fully initialized, you may want to perform additional setup or initialization. For example, you may want to cache some data from a web service or a database. Griffon provides a post-initialization hook to help you; we'll look at that too. Once we have that foundation set, we'll examine the primary purpose of controllers: actions.

Let's get started on our dissection of controllers by looking at injected properties and methods.

5.1.1 Quick tour of injected properties and methods

Controllers don't exist in a vacuum. They interact with other parts of the application, such as their view, the model, and the application. To understand controllers, let's preview some of the information we'll cover in chapter 6: the properties and methods that are injected into every controller managed by Griffon. We'll start with four properties.

APP

The `app` property points to the current running application. This property is useful because you can access the application's configuration through it. By manipulating

`app.config` (an instance of `groovy.util.ConfigObject`), you can determine what options and flags were set in `Application.groovy`.

Through the `app` property, a controller can inspect other MVC groups, because the application instance keeps a map of all currently instantiated MVC groups. Going back to the GroovyEdit example, it's possible for the `GroovyEditController` to determine how many instances of the `FilePanelController` have been created either by counting the number of tabs on the view or by inspecting `app.controllers` and checking for their type. An application also exposes views (`app.views`), models (`app.models`), and builders (`app.builders`).

The `app` property also comes in handy when you're dealing with application events. Thanks to `app`, controllers can send events to other components of your application.

MODEL

The `model` property, when defined, lets the controller access the model that's related to its group. A controller usually updates the view via the model, assuming the proper bindings are put into place. But there's no strict requirement for a controller to always have a model. If you feel your controller doesn't require a model, you can safely delete this property from the code; there won't be an error, because Griffon will check for the property declaration before setting the value.

VIEW

The `view` property references the third member of the MVC triad: the view. Like the model, this property is optional. In some cases the controller may interact with the view solely by updating the model, in which case deleting this property from the code is safe.

BUILDER

The last of the optional properties, `builder` points to the builder used to create the view related to this controller. The view and the builder may share the same variables, but they're two distinct objects: the view is a script that tells you what you just built, and the builder is a `CompositeBuilder` instance that allows the controller to build new UI components if needed.

Now let's look at the methods every controller shares.

CREATEMVCGROUP() AND BUILDMVCGROUP()

These two methods allow a controller to instantiate a new group. The former relies on the latter: whereas `createMVCGroup()` returns a List of three elements (model, view, and controller), `buildMVCGroup()` returns an `MVCGroup` instance with all the elements that were created, including the builder. This is because an MVC group may have additional configured members, such as actions or dialogs. This being said, when you only care about the canonical MVC group members, the first method is preferred; but if you need to reference additional MVC members, the second method is the one you should use.

Here's an example usage of `createMVCGroup()`. You use the assertions to verify that you got the correct types for each member as defined in the group's configuration:

```
def (m, v, c) = createMVCGroup("filePanel", [tabPane: view.tabPane])
assert m.class == FilePanelModel
assert v.class == FilePanelView
assert c.class == FilePanelController
```

Notice that you're taking advantage of a cool feature found in Groovy since version 1.6: multiple assignment. Compare the previous snippet with an invocation of `buildMVCGroup()` with the same arguments:

```
MVCGroup group = buildMVCGroup("filePanel", [tabPane: view.tabPane])
assert group.model.class == FilePanelModel
assert group.view.class == FilePanelView
assert group.controller.class == FilePanelController
```

That's right, you get each member keyed by its type, and the value is the proper instance of each member.

DESTROYMVCGROUP()

This is the counterpart of the previous two methods. The `destroyMVCGroup()` method removes the group reference from the application, thus making each MVC member a candidate for garbage collection if no one else holds a reference to any member of the group. The following snippet shows how this method can be used:

```
def (m, v, c) = createMVCGroup("filePanel", [tabPane: view.tabPane])
...
destroyMVCGroup("filePanel")
```

It's important that every group is destroyed at the appropriate time. The application will destroy every group that remains at the time of shutdown. But there may be times when you need the group to be removed before the shutdown phase is triggered, such as when closing one of the tabs in the GroovyEdit application; that's when you'll use this method.

WITHMVCGROUP()

As we just explained, `createMVCGroup()` and `destroyMVCGroup()` are two sides of the same coin. They help you create and destroy a particular group. The `withMVCGroup()` method is a handy mixture of the two that makes sure the created group is destroyed immediately after it's no longer of use. For example:

```
withMVCGroup("dialog",[owner: window]) { m, v, c ->
    m.message = "Account processed succesfully"
    m.title = "Result"
    c.show()
}
```

The previous snippet assumes there's an MVC group called `dialog`: it's a short-lived group by design, whose job is to display a customized dialog. The group will be automatically destroyed once the dialog is dismissed by the user. How does the code know? It's likely that the dialog is a modal one, which means that when the dialog is shown, it will block the current window until the dialog is dismissed. In terms of code, the closure used as a parameter to `withMVCGroup()` is halted after the call to `c.show()` is executed.

When the dialog is dismissed, the closure will resume execution; but there are no more sentences to execute, thus returning control to the `withMVCGroup()` method. The method in turn realizes there's nothing left to do and immediately proceeds to destroy the group.

NEWINSTANCE()

Last is the `newInstance()` method. Its responsibility is to create a new instance of a particular class, with an associated type as metadata. Why is this method important? Because it triggers an application event every time it's invoked. You'll see the repercussions of such an event before the end of the chapter, when we discuss complex services.

This method takes two arguments: the class to be instantiated and an optional type. The following snippet shows its usage:

```
newInstance(BookService, "service")
newInstance(Book, "")
```

In the first example, you create an instance of the `BookService` class and let every listener know that this instance is of type `service`. This is of course just for demonstration purposes; there's no need to explicitly instantiate a service like this, as you'll see later in this chapter. Next you instantiate a `Book`. Given that this class is a regular bean and has no ties to Griffon's artifacts, you omit the type by setting the second argument to an empty string. It could also have a null value.

This sums up the properties and methods that every controller has. Now let's look at how you can do some additional setup of the controller using the post-initialization hook.

5.1.2 Using the post-initialization hook

Naturally, with every Java or Groovy class, you can define a constructor that performs initialization tasks as you see fit. But what if you'd like to perform additional initialization after all members of a controller have been injected? That's the reason we have the `mvcGroupInit()` method.

> **NOTE** Remember that the initialization order of MVC members is determined by their definition order in `Application.groovy`. The order is set to model, controller, view by default.

The signature of the method is this:

```
void mvcGroupInit(Map<String, Object> args)
```

It's an optional method, so nothing bad will happen if you delete it from the source generated by the default template, because it won't be called if it's not present. The template adds it for your convenience and to remind you that you may perform additional initialization with it.

Remember the map argument that the `createMVCGroup()` and `buildMVCGroup()` methods require? It's the same map you get as the input for `mvcGroupInit()`. You may recall from the GroovyEdit application that `FilePanelController` defined this

method. It did so to keep track of its `mvcId`, read the file's text, and place the text on the model. The following snippet reproduces the contents of that method:

```
void mvcGroupInit(Map<String, Object> args) {
   model.loadedFile = args.file
   model.mvcId = args.mvcId
   doOutside {                                  <-- Load file outside EDT
      String text = model.loadedFile.text
      doLater { model.fileText = text }         <-- Update model inside EDT
   }
}
```

This snippet also serves as a reminder that you can add other methods to controllers, depending on the configuration you set in `Builder.groovy`. Suffice it to say that threading-related methods are added to controllers by default. Those methods are `edt{}`, `doLater{}`, and `doOutside{}`, which will receive full coverage in chapter 7. For now, we'll just say that these three methods make your life much easier when it comes to multithreading code.

It's time to dive into the main responsibility of a controller: being an action handler.

5.1.3 Understanding controller actions

We've reached the core of a controller. Actions are the main reason for a controller's existence. You've seen them before in previous examples, and now it's time to define them properly.

An action is nothing more than a closure property or a method that follows a conventional set of arguments. It looks like the following when defined as a closure property:

```
def openFile = { evt = null -> ... }
```

The alternate form, for an action defined as a method, looks like this:

```
void openFile(evt = null) { ... }
```

Why allow two modes? The reason behind this design selection is that developers like to have choices too. Some prefer the closure property notation, because it aligns perfectly with the conventions of Grails controllers. Others prefer a method definition, because that's what they're used to, coming from Java. One thing is certain: it doesn't matter which mode you pick. Griffon will make sure it works.

But there's an advantage to using closure properties over method definitions. When it comes to testing, it's easier to overwrite an action implemented as a closure than it is to mock out a method.

About the conventional arguments: an action is usually tied to an event generated by the UI. It may vary in type depending on the element to which you tie it. For example, it may be an `ActionEvent` if you set the action as an `ActionListener`. Or it may be a `MouseEvent` if you set the action to handle `mousePressed` on a button. Leaving the type of the `evt` parameter as undefined gives you enough elbow room to switch an action from one place to another without needing to change the action's signature. What do we mean by this? Say your intention is to react to events generated when a

button or a menu is clicked. These components generate events of type `ActionEvent`. Then you change your mind and would like to have the action react to mouse movements, which are typically handled by `MouseEvent`. If the type of the `evt` argument is strictly set and left unchanged, then it's likely you'll get a runtime exception when running the application. But if the type of `evt` isn't set, you can freely assign the action to react to `MouseEvents`. This assumes that the action code doesn't depend directly on behavior available exclusively to a particular event type.

There's another advantage to omitting the type on the `evt` parameter: testing. Assume for a moment that a controller has the following action:

```
def handleEvent = { evt = null ->
    if(evt?.source?.selected)
       doSomething()
    else
       doSomethingDifferent()
}
```

You'd like to test this code, but it may be a difficult task because the code expects a nested element with a specific property to be defined in the `evt` argument. You may need to create an event instance and populate it with the correct data, but which event class should you use? Because there are no types involved, you can use Groovy's duck-typing approach. Yes, we're suggesting you use a map as the value for `evt`. The following code shows how to do this:

```
def evt = [source: [selected: true]]
myControllerInstance.handleEvent(evt)
evt.source.selected = false
myControllerInstance.handleEvent(evt)
```

The second line causes `doSomething()` to be called, and the last line causes `doSomethingDifferent()` to be called.

One last thing we must cover about method arguments is the recommendation that you define a default value. The template suggests that you set a null value, but it could be a predefined map, as shown in the previous snippet. It can be any value that makes sense for your action. Why would you need such a default value? Think for a moment about what default values in Groovy allow you to do. That's right: you can call the method (or closure) without defining a value for a particular parameter. This means you can call `handleEvent` in either of the following forms:

```
handleEvent([source:[selected: true]])
handleEvent()
```

This can greatly simplify the code you'll need to write to call an action from within its own controller—or any other MVC group member, for that matter.

> **NOTE** This is all you need to know about a controller's responsibilities for now. But there's more. In chapter 8, we'll explain the relationship between controllers and application events.

Reusing business logic via services is an efficient way to scale an application, and we'll look at that next.

5.2 The need for services

We've showed you what makes a controller tick. As you know, every MVC group may have its own controller. As you'll learn in chapter 6, two or more MVC groups may share a few of their members, or even the controller. But this sharing ability doesn't scale when what you need is a place to put your code where any controller or application component can access it. You need good old-fashioned modularity.

Not to get all philosophical, but the purpose of the controller is to control. Controllers respond to actions and events and see that the appropriate code is executed. Then the controller returns a response if appropriate. In general, in large applications that need to reuse code, in order to make it more manageable, reusable, and testable, the controller should delegate to some other component to fulfill business logic. Frequently, this other component is called a *service*. A service is an organizational technique for encapsulating logic for reuse.

Services are flexible and have numerous uses. One example is accessing remote web services. Let's say you have a customer relationship management (CRM) solution that exposes access via web services. Your application may need to access customer information from multiple components within the application. Instead of duplicating the code in each of the components, this is a good time to use a service. Figure 5.2 illustrates extracting web-service access used by two controllers into a shared service.

Griffon comes with services support. A service in Griffon is stateless, akin to its MVC brethren, and it follows a naming and location convention. There's also a creation script that uses a simple template.

In this section, you'll see how to create simple and complex services. A *simple service* is a lightweight service that has few or no dependencies on other components. When

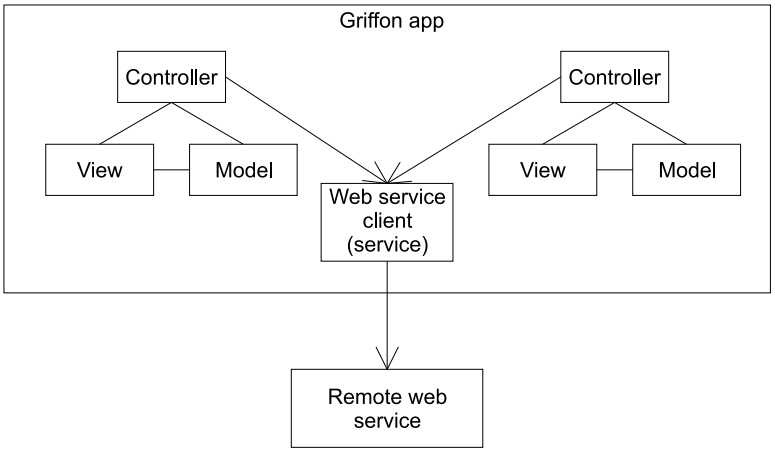


Figure 5.2 Controllers sharing logic to access a remote web service

a service gets more complex and has dependencies on other components, a more involved approach is available.

Let's look at how simple and complex services are created with Griffon.

5.2.1 Creating a simple service

The recommended approach for creating services is to use Griffon's `create-service` command. In a brand-new Griffon application, go to your command prompt and type

```
$ griffon create-service simple
```

This will create a new artifact named `SimpleService.groovy` under griffon-app/services. It will also create a `SimpleServiceTest.groovy` file under test/unit. This is why we recommend that you use the script; not only does it create an artifact that follows the naming conventions and contains a skeleton implementation, but it also generates a test script for you.

Now let's peek into the generated code:

```
class SimpleService {
    def serviceMethod() {
    }
}
```

It couldn't be any easier than this. A service class is like any other Groovy class you've encountered so far; there's no magic to it. It may contain as many methods as you like, with the knowledge that public methods define the service contract. Service instances are automatically created and managed by the Griffon runtime; they're treated as singletons, although if you look again at the service you just defined there's nothing stopping you from creating your own instances. But it's good to leave Griffon to do its own thing.

How do you wire up a service into a controller? You may be hoping that a convention exists to help you attain this goal—and you're correct! Griffon injects an instance of a service on each MVC member that has a property name, which matches the simple name of the service. With the service injected into the controller, as illustrated in figure 5.3, the controller can invoke methods on the service.

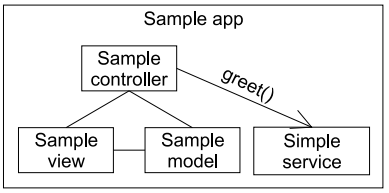


Figure 5.3 Sample controller invoking the `greet()` method on a simple service

Does this sound familiar? It works the same as injecting model, view, and controller references on MVC members. Let's look at an example.

Edit SimpleService.groovy in your favorite editor, making sure its contents look like this:

```
class SimpleService {
    def greet(String who) {
         "Hello $who"
    }
}
```

`SimpleService.greet()` implements a "Hello World" style service call. It returns its argument formatted as a greeting.

Now let's inject this service into a controller. Remember, you only need to define a property that matches the simple name of the service; in this case, it will be `simpleService`. Assuming you had a service class `com.acme.DeliveryService`, its simple name would be `deliveryService`. It's important that the name match—otherwise the service instance won't be injected. You can define the type of the service as well. It makes no difference to Griffon, but it may be important when editing your code in an IDE or a power editor that supports code completion:

```
class SampleController {
   def simpleService
   void mvcGroupInit(Map<String, Object> args) {
      assert simpleService.greet("Griffon") == "Hello Griffon"
   }
}
```

This is a contrived example, because the controller has no actions. But it serves to verify that the service instance has been properly injected and that calling its service method results in the expected output.

Now you know how to inject services into controllers. The option of lightweight service injection will work as long as your services are lean—in other words, they don't have dependencies on additional components. Fortunately, there's a solution to this problem too, which we'll cover in the next section.

5.2.2 Creating a Spring-based service

As your application becomes more robust, you may need a more sophisticated service implementation. Assume for a moment that you have a service that requires an additional dependency on another component. This dependency may be a data source element that lets the service interact with a database, or it may be a JMS destination queue or some other custom component exposed by your application. Now imagine that those dependencies require additional setup as well. The list could go on and on.

Inversion of Control[1] (IoC) frameworks (or dependency injection,[2] as some prefer to call this approach) have risen to solve the problem of properly setting up a graph of dependencies (sometimes including circular references!). Arguably the most popular of such frameworks in the Java space are Guice (http://code.google.com/p/google-guice/) and the Spring framework (www.springsource.org/); there's plenty of information available that can help you get up to speed on them.

Griffon comes with a couple of plugins that can help you use Guice or Spring. You'll be able to set up complex services, as long as you follow the framework's rules.

[1] http://en.wikipedia.org/wiki/Inversion_of_Control.
[2] http://en.wikipedia.org/wiki/Dependency_Injection.

In this section, you'll configure a complex service using the Spring plugin, because a handful of additional Griffon plugins take advantage of Spring support. Some of these plugins may be familiar to you if you come from a Grails background.

The first thing you need to do is install the Spring plugin. To do so, go to your command prompt, making sure you're inside an application directory, and type the following command:

```
$ griffon install-plugin spring
```

Good. Let's move on.

CREATING A SERVICE

Now create another service, named `complex`:

```
$ griffon create-service complex
```

Let's take a few more baby steps. First you'll make sure both `SimpleService` and `ComplexService` will be injected into your `SampleController` via Spring injection, and then you'll tweak `ComplexService` to have additional dependencies.

MODIFYING YOUR SERVICE AND CONTROLLER

Edit `ComplexService`, and change its default service method to look like this:

```
class ComplexService {
   def call(String name = "") {
       "complex replies: $name"
   }
}
```

Now go back to SampleController, and change its contents to look like this:

```
class SampleController {
    def simpleService
    def complexService

    void mvcGroupInit(Map<String, Object> args) {
        assert simpleService.greet("Griffon") == "Hello Griffon"
        assert complexService.call("Griffon") == "complex replies: Griffon"
        println "All is well"
    }
}
```

Run the application. If no errors appear on your console and you see the message "All is well," then you're good to go with the next step. If there are errors, check to make sure the names of the services are correct. Next, you'll give the complex service a dependency.

CREATING A CLASS AND ADDING IT TO YOUR SERVICE

Create a new class, and call it `Thing`. Make sure you place it under `src/main/Thing.groovy`. The file contents should look like this:

```
class Thing {
    String value
}
```

Go back to `ComplexService`, add a `Thing` property to it, and change the implementation of the `call()` method to use the new property:

```
class ComplexService {
   def thing

   def call(String name = "") {
       name ? "complex replies: $name" : thing.value
   }
}
```

You're almost done.

MODIFYING THE CONTROLLER AGAIN

Now you'll change the controller code again, and save the wiring setup of `Thing` into `ComplexService` for last. Open `SampleController` in your editor, and type the following:

```
    void mvcGroupInit(Map<String, Object> args) {
        assert simpleService.greet("Griffon") == "Hello Griffon"
        assert complexService.call("Griffon") == "complex replies: Griffon"
        println complexService.call()
    }
```

Now you're ready for the final step.

INJECTING THINGS INTO COMPLEXSERVICE

You need to instruct the Spring plugin that an instance of a `Thing` must be injected into an instance of `ComplexService`. There are many ways to configure injection in Spring. Perhaps the most popular is via XML, but many developers tend to shun anything XML related. Don't fret, there's a groovier solution.

Rising from the core of the Grails framework, you find `BeanBuilder` (www.grails.org/Spring+Bean+Builder). This builder lets you configure an `ApplicationContext` using a groovy DSL, much as you do with Swing and the SwingBuilder DSL, or Ant build files via `AntBuilder`. Figure 5.4 illustrates the process of using `BeanBuilder` to inject resources .groovy configuration information into the `ApplicationContext` instance.

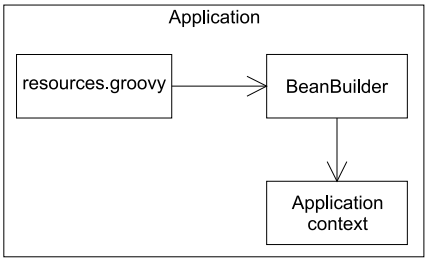


Figure 5.4 BeanBuilder processing the resources.groovy file to build the application context

It turns out the Spring plugin bundles `BeanBuilder` and its supporting classes, meaning the Spring beans DSL is yours to use on Griffon applications as well. Then how do you take advantage of the Spring beans DSL? Create a new file named `resources.groovy` under src/spring, and type in the following:

```
beans = {
    thing(Thing) {
        value = "All your Griffon are belong to us!"
    }
}
```

TIP The plugin creates the `src/spring` directory when you install it.

The `BeanBuilder` DSL requires that you define a top-level variable named `beans`. Its value must be a closure containing bean definitions. A bean definition has the following elements:

- `name`—In this case, `thing`
- `class`—In this case, the `Thing` class
- `optional`—A nested closure containing property definitions

For the test example, we decided to grace `thing`'s value with a popular meme[3] from the early 2000's; you may be familiar with it.

You're good to go. Run the application again, and if everything goes right you should see the famous meme phrase printed on your console after a few logging statements from the Spring plugin. The output will be similar to figure 5.5.

Imagine wiring up data sources, JMS queues, and the like. It doesn't seem that difficult now, does it?

You've seen how simple and complex services can enhance your application's behavior. Services are often called from a controller, but given the choices of dependency injection at your disposal you may inject them into other components too.

Still, the behavior provided by controllers and services may not be enough to cover your application's requirements. Sometimes you'll need to enhance a particular class or a set of classes that belong to the same type, like models, for example. This goal can

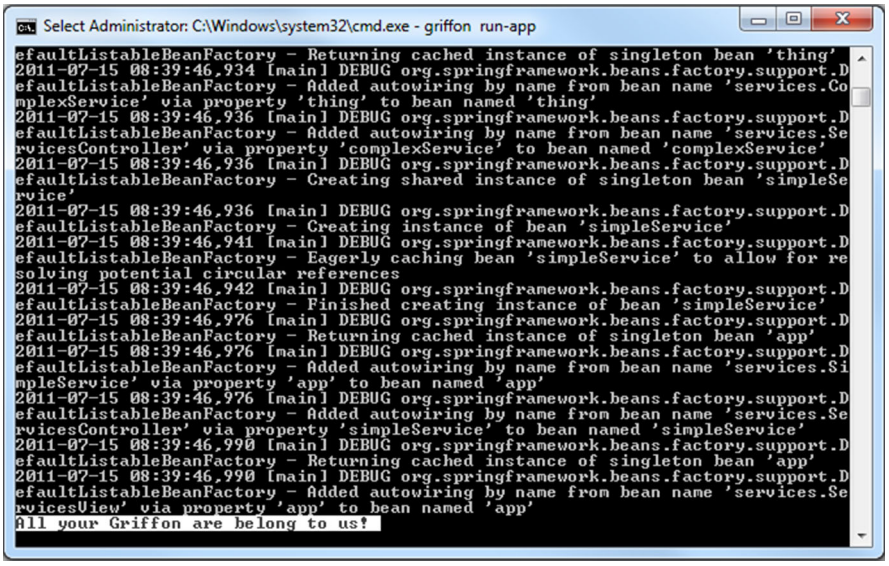


Figure 5.5 Complex service results

[3] http://en.wikipedia.org/wiki/All_your_base_are_belong_to_us. It keeps popping up from time to time!

be achieved in several ways; we'll discuss in the next section one that we're sure you'll find useful.

5.3 Artifact management

We've discussed the various features of Griffon's MVC implementation using a common set of artifacts: models, views, and controllers. We also added services into the mix. They each possess their own individual properties, but they also share common traits. For example, each artifact is located in a specific directory that shares a name with the artifact's type. Every file has a unique suffix that clearly indicates the artifact type. Because of this, Griffon is able to group all the artifacts into a runtime representation that we call `GriffonClass` (see figure 5.6) A `GriffonClass` is a metadata class that holds all the relevant info pertaining to artifacts, such as `FilePanelController` or `GroovyEditView`. The type of metadata you have access to is specific per artifact. For example, you can inspect controllers to figure out the names of all actions they expose. Or you may want to know the names of all the service methods that a particular service class defined.

In this section, we'll discuss how the Artifact API comes into play. For example, it can be used at runtime to figure out the names of all the actions exposed by a controller. Suppose you wanted to build a form-based application where all interactions, represented by buttons or menus, were automatically mapped to the actions exposed by a controller. Having a list of all available actions in the controller would certainly make your job easier. You could then query all artifacts by means of the Artifact API, which is available to you though the `ArtifactManager`. Let's see how it's done.

5.3.1 Inspecting artifacts

Every Griffon application has an implementation of the `ArtifactManager` interface. At application startup, Griffon loads artifact metadata and makes it available for querying via `ArtifactManager`. You can access `ArtifactManager` by asking the application instance for it; you can either call the `getArtifactManager()` method on the `app` variable or use property access and call `app.artifactManager`.

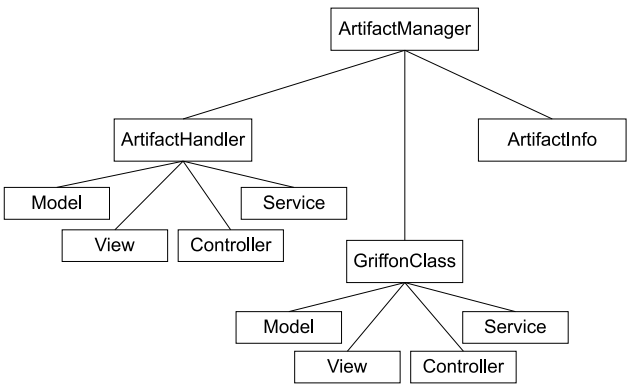


Figure 5.6 Core Griffon classes that perform artifact management

Coming back to the hypothetical scenario we presented at the beginning of the previous section—gathering all actions exposed by a controller—the following snippet shows how this can be done in a view script:

```
def griffonClass = app.artifactManager.findGriffonClass('AuthorController')
griffonClass.actionNames.each { actionName ->
    button(actionName, actionPerformed: controller[actionName])
}
```

Table 5.1 summarizes the methods and properties you can use to query artifact metadata.

Table 5.1 A comprehensive list of methods and properties available on `ArtifactManager`

Method	Returns
`findGriffonClass (String className)`	A `GriffonClass` instance whose artifact class matches the fully qualified class name sent as argument. Returns null if no match is found. Example: `findGriffonClass ("com.acme.AnvilDeliveryService")`
`findGriffonClass (Object object)`	A `GriffonClass` instance whose artifact class matches the class of the object sent as argument. Returns null if no match is found. Example: `findGriffonClass(AnvilDeliveryServiceInstance)`
`findGriffonClass (String name, String type)`	A `GriffonClass` instance whose artifact class matches the combination of class and type. Returns null if no match is found. Example: `findGriffonClass ("Book", "controller")`
`findGriffonClass (Class clazz, String type)`	A `GriffonClass` instance whose artifact class matches the combination of class name and type. Returns null if no match is found. Example: `findGriffonClass (Author, "controller")`
`getClassesOfType (String type)`	An array of `GriffonClass` instances whose type matches the specified argument. Never returns null; an empty array is returned if no match is found. Example: `getClassesOfType("service")`
`getAllClasses()`	An array of `GriffonClass` with all available artifact classes. Never returns null.
`is<type>Class (Class clazz)`	True if its argument is an artifact whose type matches `<type>`, or false otherwise. Example: `isModelClass(FooModel) == true`
`get<type>Class (Class clazz)`	A GriffonClass whose class matches the provided argument. Example: `getViewClass(com.acme.BarView)`
`<type>Classes`	An array of `GriffonClass` where `<type>` is a valid artifact type. Never returns null. This is the only dynamic property exposed by the `ArtifactManager`. Example: `controllerClasses`

You may have noticed that some of the methods and properties use a `<type>` placeholder. This is because those methods and properties are dynamically generated when

you use them. That is, there is no `controllerClasses` property on `ArtifactManager` until you call it for the first time. Why is this? Because as a developer, you have the ability to define new artifact classes. How else would `ArtifactManager` know about your new artifact types?

Now that you know how to query for artifact metadata, let's see what you can do with it. The `GriffonClass` class has many methods that can be used to inspect an artifact, and the most useful are described in table 5.2.

Table 5.2 The most commonly used methods of `GriffonClass`

Method	Behavior
`getApp()`	Returns the current application instance. Every artifact has this method.
`newInstance()`	Creates a new instance of the particular artifact. Instances created in this way benefit from the framework's bean-management capabilities, such as service injection and event firing.
`getArtifactType()`	Returns the type of the artifact, such as `controller` or `view`.
`getClazz()`	Returns the real class of the artifact this `GriffonClass` describes. Example: `com.acme.AnvilDeliveryService`
`getFullName()`	Returns the fully qualified class name of the real class. Example: `com.acme.AnvilDeliveryService`
`getPackageName()`	Returns the package name only, if it exists. Example `com.acme`
`getShortName()`	Returns the class name without any preceding package. Example: `AnvilDeliveryService`
`getPropertyName()`	Returns a name suitable to be used as a property. Example: `anvilDeliveryService`
`getName()`	Returns the short class name without the trailing type convention. Example: `anvilDelivery`
`getNaturalName()`	Returns a string representation that is suitable for human consumption. Example: `firstName` becomes `First Name`

In addition, custom subclasses and implementations of `GriffonClass` can expose more methods. For example, the `GriffonClass` for controllers (aptly named `GriffonControllerClass`) allows you to query all controller action names. In contrast, the `GriffonClass` for services (`GriffonServiceClass`) has a method for querying the names of all service methods defined by a particular service. You'll find that plugins and addons can be used to deliver new `GriffonClasses`, such as charts and wizards.

Now that you know how to query for artifact metadata and what to expect of such metadata, we're ready to explore the metaprogramming capabilities that the Artifact API enables.

5.3.2 Metaprogramming on artifacts

Groovy supports metaprogramming at compile time and runtime. Lower-level metaprogramming occurs at compile time. This can be done by using the AST transformation, which we are obliged to remind you isn't for the faint of heart. `@Bindable` is a perfect example of this type of metaprogramming. Higher-level metaprogramming occurs at runtime. This is the most typical, and it's well documented in many sources; *Groovy in Action*, 2nd edition (Manning, 2012, www.manning.com/koenig2) is a great source to start with. `@Bindable` is also the one we'll illustrate.

Every class in the Groovy system has a companion `MetaClass`. Groovy uses this meta descriptor to implement much of its Meta Object Protocol[4] (MOP). When a method is invoked on an object or a property is accessed, the MOP gets to work. The MOP can follow several paths to resolve a method invocation; but for our purposes the short story is that if the `MetaClass` has the method definition the MOP is looking for, then the MOP will invoke it; if not, the MOP will try the class, resulting in an exception if the method isn't found.

> **Behind the scenes**
>
> One of the `MetaClass` features is that you can attach new methods and properties to it at virtually any point while the application is running. This is true if the `MetaClass` is an instance of `ExpandoMetaClass`,[5] another fine addition to the Groovy language that was incubated in the Grails project. Most of the time, when you set up additional methods on a Groovy `MetaClass` you'll find an `ExpandoMetaClass` under the covers.

Enough with the theory. Let's move ahead into exploiting the introspection abilities provided by `GriffonClass`.

5.3.3 Artifact API in action

Let's say you need to build a form-based application that deals with personal records found in a database. We'll keep the code short for the moment—you won't see any database access shenanigans—but rest assured that Griffon has good support for connecting to databases and executing queries, thanks to its plugin system.

Say that an initial version of the application looks like figure 5.7.

This screen suggests a particular structure for the MVC group that handles it. You can create a model that holds `firstName`, `lastName`, and `address` properties. You'll have to find a way to deal with proper capitalization for each property label. The actions can be safely stored in a controller; label capitalization also plays a role here. But the application isn't complete; additional properties will be added to the model. If you hard-code all values and properties in each MVC member, you'll quickly reach a

[4] See "Practically Groovy: Of MOPs and mini-languages," http://mng.bz/us97, for an example usage of the Groovy MOP.

[5] http://groovy.codehaus.org/ExpandoMetaClass.

Figure 5.7 First iteration of the form. There are three input fields and two actions.

point where there's too much repetition. You need a better way to mine the information found in the model and controller. This is where the Artifact API comes in.

First you'll define the model with the three properties you just saw, as the following listing shows.

Listing 5.1 `FormModel` with three bindable properties

```
@groovy.beans.Bindable
class FormModel {
    String firstName
    String lastName
    String address
}
```

That takes care of the model. Don't you love the simplicity of building observable beans with Groovy and Griffon? The property names resemble the labels shown in figure 5.7, but you haven't figured out a way to properly capitalize them; you'll leave that task to the view.

Next you'll define the controller in the following listing. This is where you'll catch a glimpse of the Artifact API at work.

Listing 5.2 `FormController`, which can handle any model properties

```
import griffon.util.GriffonNameUtils
import griffon.transform.Threading
class FormController {
    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 ->
                "${GriffonNameUtils.getNaturalName(name)} = ${model[name]}"
```

Access model properties (1)

```
            }.join('\n')
        )
    }
}
```

As expected, the controller has two actions that match the labels on the buttons shown in figure 5.7, although they also present the capitalization issue of the model properties. What's interesting are the lines where the controller queries the model for its `griffonClass` instance ❶. Models have a custom `GriffonClass` that provides additional introspection capabilities; we mentioned this earlier. A model `griffonClass` exposes a method that returns a list of names of all the observable properties the model defined. In this case, this list will contain `firstName`, `lastName`, and `address`. That's precisely what you need. Inspecting the behavior of the controller further, notice that the `clear` action resets the value of each property, and the `submit` action opens a dialog with each value preceded by its label. And the mystery of proper capitalization is finally solved: Griffon has a number of utility classes in its arsenal, and one of them, `GriffonNameUtils`, excels at transforming strings. Capitalization is one of the transformations.

Next, you'll define the view, as shown in the following listing. Notice that you're using the `GriffonNameUtils` class again.

Listing 5.3 `FormView` with two panels: model properties and controller actions

```
import griffon.util.GriffonNameUtils as GNU
application(title: 'Form',
  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')
        }
    }
}
```

❶ **Accessing model properties**

❷ **Access controller actions**

You can appreciate at ❶ that the same trick is used in the controller to query the model for all its observable properties (that is, asking the model's metadata about all the properties it holds that are of interest for this application to work). The left panel has a special layout[6] that places each row nicely. A row is composed of a label and a text field. The text of the label is properly capitalized thanks to `GriffonNameUtils`—aliased to `GNU` using one of Groovy's tricks to shorten a class name. Controllers also have a special `griffonClass` of their own. This particular `griffonClass` has a method that provides the names of the actions declared by the controller. Again this is precisely your goal, and you put that method to good use ❷.

Now that all the portions of the code are ready, you can set up the application for running it. Install the MigLayout plugin by invoking the following command at the console prompt:

```
$ griffon install-plugin miglayout
```

MigLayout

MigLayout is a good Lava layout manager. In listing 5.3, it's used in the constraint definitions. MigLayout defines the appearance and behavior of the fields and buttons. You can find out more at www.miglayout.com.

Once the plugin is installed and the application is running, filling out the form and clicking Submit (see figure 5.7) should result in a dialog similar to the one shown in figure 5.8.

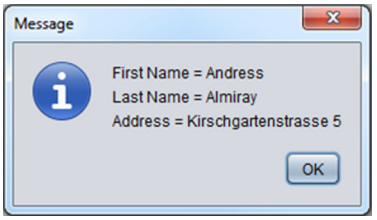


Figure 5.8 A dialog opens when you click the Submit button. It shows all the information entered by the user on the form.

Let's verify that the model and controller introspection are working as we just described. In theory, adding new properties to the model should result in additional labels and text fields being displayed on the left side the form; a similar thing should happen on the right side of the form if actions are added to the controller. Update the model by adding two more properties: `city` and `country`. Then update the controller by adding a new action, like this:

```
def quit = {
    app.shutdown()
}
```

Launch the application once more. Lo and behold, the UI reflects your changes! Figure 5.9 shows how the UI looks now.

Not bad at all. And you only had to add a few properties to the model and controller. You can expect additional features for each custom `griffonClass`. You can inspect

[6] `migLayout()` from the MigLayout plugin.

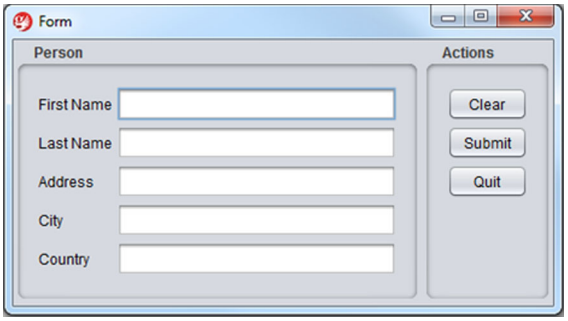


Figure 5.9 The form displaying the new model properties and controller actions

them by using the options available to you, be it IDE code inspection or browsing the API documentation that comes bundled with the Griffon distribution.

5.4 Summary

Controllers are a vital part of the MVC pattern. They're responsible for routing inputs and outputs between the other members of the MVC triad. Controllers in Griffon share common properties with their models and views.

You've seen how action handlers can be defined on a controller. You've also seen that controllers can take you only so far when it comes to defining an application's logic. Sometimes you may need to put the logic on a service layer.

You created a simple service to see Griffon's lightweight service support. We also examined a more complex service and saw the need for robust service support using Guice or Spring.

Finally, we covered the Artifact API, which gives you access to artifact metadata. Think of it as introspecting into your application internals.

In the next chapter, we'll look at how models, views, and controllers form MVC groups, and you'll learn how to create and use MVC groups in your applications.

6 Understanding MVC groups

This chapter covers

- Declaring MVC groups
- Creating MVC groups
- Using MVC groups

We touched on the subject of MVC groups in previous chapters. As a matter of fact, we covered each of the default individual members in the last three chapters. But there's more to groups than what you've seen so far. At this point, you know that a group comprises model, view, and controller members, each of which follows a naming convention. But did you know you can define additional members in a group that don't necessarily follow the MVC pattern? Or that you can define a group with only a view and a model? Also, recall from the first example in this book that you can programmatically create new group instances on the fly, not just use those initialized by default upon application startup.

In this chapter, we'll discuss all these features and more. Our goal is to help you gain a better understanding of the inner workings of MVC groups.

Let's begin the journey by recapping how MVC groups can be created and how they behave. Some cautionary advice: the first two sections explain in detail what the framework does when instantiating and managing groups. If you're only interested

in working with MVC groups directly, you can skim through these sections to get a basic understanding of the underlying mechanism and jump directly to section 6.3. Of course, we recommend that you visit sections 6.1 and 6.2 if you have doubts or concerns about the topics they discuss.

6.1 Anatomy of an MVC group

The easiest way to create an MVC group is via the Griffon command line. Assuming you're already inside a project like GroovyEdit (see chapter 1), you can create a `filePanel` group like this:

```
$ griffon create-mvc filePanel
```

Executing this command does two things: it generates a set of files representing the members (model, view, and controller) of the MVC group, and it changes the configuration files to tell the framework that you have a new MVC group.

Griffon generates four files, each in the appropriate functional directory: a model, a view, a controller, and a test file. Each of the files also has a name derived from the group name. And each of the files is in the same package, even though they live in different directories. Why? Because of convention over configuration: placing together the files that fill the same responsibility encourages you to make sure the labor stays appropriately divided.

Let's walk through each of the generated files. Don't worry, this will be a quick look. Because of Groovy's power as a dynamic language, it strips away a lot of the ceremony you may expect to see—unless you've been using Groovy for a while already, in which case its relative brevity won't be a surprise.

6.1.1 A look at each member

The first stop on our MVC group tour is the model. Griffon generates a model file for you based on the name of your MVC group in griffon-app/models. In this case, the name of the file is `FilePanelModel.groovy`:

```
import groovy.beans.Bindable

class FilePanelModel {
   // @Bindable String propName
}
```

Yes, this code snippet is the entire contents of the file. Because you don't yet know what properties you want to add to your model, you don't add any; the comment serves to show how simple it is to create a bound property named `propName`. Uncommenting that line is all that's required to get an observable property! But you knew this already, because models were discussed in ample detail back in chapter 3.

Next up is the view. The view files live under griffon-app/views/ and are named based on the group name. This view will be called `FilePanelView.groovy`. Remember the pattern? It's a convention: all MVC group members are stored in griffon-app/<portion name>/, and the names are <MVC Group name><portion name>.groovy. If you put

your MVC group in a package, then the file will also be in an appropriately named set of directories beneath the storage directory, just like any other Java file:

```
application(title:'GroovyEdit',
  //size:[320,480],
  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') // deleteme
}
```

It's a bit longer than the model, mostly because you need to do something to get a minimally functional MVC group going. The first thing you'll notice is that there's a lot of nesting. That is because GUIs in Griffon are written in a declarative fashion (this goes here, that goes there, these interact like this) rather that in the imperative fashion (create the panel, create the button, add the button to the panel, create a listener, add the listener to the button). This may seem weird at first, but it greatly improves the readability and maintainability of the application.

In the default view, you first declare an outer application (it could be a JFrame or an applet; Griffon smooths over the differences for you). You set the Griffon logo as the frame icons, and you do this in the view rather than apply a default because you'll likely want to create your own icon for your applications. Inside the frame, you create a label indicating that the content goes here because—guess what?—the content goes here. What kind of content? The components, widgets, and pixels that make up the visual part of your application. Feel free to go back to chapter 4 to refresh your knowledge of views and widget nodes.

Your third stop is the controller. As per convention, you'll find the controller file in `griffon-app/controllers/FilePanelController.groovy`:

```
class FilePanelController {
    // these will be injected by Griffon
    def model
    def view

    void mvcGroupInit(Map args) {
        // this method is called after model and view are injected
    }

    /*
    def action = { evt = null ->
    }
    */
}
```

The truth is that almost all of this content could be erased, and the MVC group you generated would still function. This code is a stub for features you'll almost certainly need to

add to get a reasonably useful MVC group—for example, one that defines actions that respond to menu inputs, as you saw back in chapter 1 with `GroovyEditController`.

At the top of the class definition are two fields to hold the references to the model and the view. Griffon will inject these fields into the class when it instantiates it for you, so all you need is appropriate properties for them to wind up in. There are uncommon cases where you may not need to reference the model or the view, so you could conceivably delete these properties.

Next you have a method named `mvcGroupInit`. This method serves as a constructor of sorts; it's called after all of the portions are instantiated and fields have been injected. You may remember we mentioned this method in the last chapter. We'll go into more detail in sections 6.2 and 6.3.

Finally, a commented-out section of code represents the execution portion of an action. In a well-behaved Swing application, actions are king, so first-class support of actions is important. Actions can be implemented by defining either closure properties (as the template suggests) or public methods. The choice is yours. There's a slight performance advantage if the action is implemented as a public method, though; recall our discussion of controller actions from chapter 5.

Thought you were done, didn't you? Sorry, but you're not finished until you have proper testing code. Because it isn't strictly a part of the MVC group, only part of the convention applies: the testing stub will be in the `tests/integration` directory. It will be named as expected, though—`FilePanelTests.groovy`:

```
class FilePanelTests extends GroovyTestCase {
    void testSomething() {
        fail("Not implemented yet!")
    }
}
```

Again, the generated code is fairly sparse. Generated tests are wired to fail if left unimplemented. It's up to you to decide how they should be implemented. After all, Griffon alone can't decide the best way to test your production code. Griffons may be magical beasts, but they're only mounts you ride into battle: you still need to fight the battle. You can flip to chapter 9 to see how best to win the testing battle.

Finally, after all the necessary files are generated, you need to register your MVC group with the framework.

6.1.2 Registering the MVC group

Where do you register a group? In the `Application.groovy` file that lives in `griffon-app/conf`. We won't look at the whole file here; the following code shows just the parts relating to MVC groups.

Listing 6.1 MVC group declarations in Application.groovy

```
. . .
mvcGroups {
    // MVC Group for "FilePanel"
```

```
    'filePanel' {
        model      = 'FilePanelModel'
        view       = 'FilePanelView'
        controller = 'FilePanelController'
    }

    // MVC Group for "GroovyEdit"
    'groovyEdit' {
        model      = 'GroovyEditModel'
        view       = 'GroovyEditView'
        controller = 'GroovyEditController'
    }
}
. . .
```

The declaration of the MVC group is fairly straightforward: in the `mvcGroups` section of the configuration, you declare the name of the MVC group and open a block describing all the portions that compose the group. The group portions are assigned to the names of the classes representing that portion of the MVC group. Note that the class names are defined as strings, for the sole reason of avoiding eager class resolution when the group configuration is loaded and parsed. The MVC group type itself is also in single quotes; this enables you to create groups that may not be groovy identifiers, possibly including spaces, dots, and any other characters you feel like adding. A key aspect of group names is that they must be unique within the application. If a duplicate name is encountered, the last group registered will win over the previous ones.

But what is this second MVC group doing in your pristine application? The `groovyEdit` MVC group configuration listed in this example existed before you created the `filePanel` MVC group; it was created when the application was created and is the MVC group that serves as the master group of the whole application. We'll go over the ins and outs of multiple MVC groups in section 6.3.

The last declarative detail is how MVC groups are bootstrapped.

6.1.3 Startup groups

In the `Application.groovy` file, there's a property named `application.startupGroups`. This property is a list of the MVC groups that should be automatically started up when the application framework starts up. These groups are created without parameters and in the same order as they're found in the list.

By default, the MVC group created as a part of the initial application is added to the list of startup groups. You're free to add any number of groups that you declare later. You could even leave the list empty, having no startup groups initialized automatically. Of course, you would then need to bring up your MVC groups manually in one of the life-cycle scripts like Startup or Ready. You may choose this option if one of the initial groups requires additional parameters, for example.

This concludes our quick summary of creating an MVC group from scratch. We turn our gaze now to the runtime aspects of an MVC group, starting with finding out how a group can be instantiated at any time.

6.2 Instantiating MVC groups

You've seen the static view of an MVC group, where the code lives, where the declarations about its details are made, and what defaults the framework puts in place. But how do these pieces come to life? What is the man behind the curtain doing? We'll now look at the different methods that play a role in the group's life cycle.

6.2.1 Creation methods

MVC groups usually don't come into being by themselves. The one exception is the startup groups discussed in section 6.1.3. After the bootstrap processes all initial groups, you're on your own to instantiate any group whenever you deem necessary. The good news is that doing so is fairly simple. Three related methods are available to every MVC member and the application instance: `buildMVCGroup()`, `createMVCGroup()`, and `withMVCGroup()`.

> **Artifact API tip**
> Every artifact that implements `griffon.core.GriffonArtifact` has access to the methods we'll discuss shortly. All basic artifact types (model, controller, view, and service) implement this interface. To safely determine whether an artifact supports this feature, you can query its `GriffonClass` for its `clazz` property.

The first two methods are identical except for the return type. `buildMVCGroup()` returns an instance of `griffon.core.MVCGroup` where each member can be found by name, and `createMVCGroup()` returns a three-element array, with the contents being model, view, and controller, in order. When coupled with the multiple-assignment syntax introduced in Groovy 1.6, the `createMVCGroup()` method can make for some very readable code. But when your MVC group is more than model, view, and controller, the `buildMVCGroup()` method gives you access to all the created members.

The methods have one required parameter and a couple of optional ones:

- `groupName`—The only required parameter for `createMVCGroup()` and `buildMVCGroup()`. It's the MVC type to instantiate. `groupName` is the same as the key used in the `Application.groovy` file.
- `groupId`—Optional. This is a name you want to give to this particular instance of the MVC group. If the name isn't provided, the MVC type is used in its place. The value of this parameter is important when you want to create multiple instances of the same group type. In chapter 1, the GroovyEdit example used this parameter, assigning a different value per tab.

  Calls to `createMVCGroup()` and `buildMVCGroup()` check for collisions with existing MVC groups. If the specified group name already exists as a group instance, then the application may react in two ways: it either throws an exception alerting you (the developer) that the code is performing an illegal operation, or it

alerts you of the problem but doesn't throw an exception. In that case, the old group is destroyed and the new one takes its place. You can alternate between these two behaviors by changing the value of a configuration flag in Config.groovy:

```
griffon.mvcid.collision = 'warning'
```

Valid values for this flag are `warning` and `exception`. Throwing an exception is the default behavior.

- `params`—Optional map of arguments to be passed into the creating process. You can use Groovy-style named parameters with this method. According to normal Groovy conventions, each named parameter will be amalgamated into the map that is passed into the method as its third parameter. These parameters can be used to pass in configuration data, contextual data, or even portions of other MVC groups. In the GroovyEdit example, this parameter was used to let the newly created group know which title it should use for its tab, as well as where it should add the new tab: in other words, the owning `JTabbedPane` instance.

The third method that lets you create a new instance of an MVC group is `withMVCGroup()`. This method is aware of the life cycle common to all groups.

A group created with this method will be automatically destroyed as soon as it's no longer of use. The typical case is a modal dialog that requires a few customizations before displaying itself, perhaps capturing some input and returning back to the original caller. Here's an example of what we just described. Suppose a group named `query` is responsible for capturing a user selection in a model property named `choice`. The group's controller also has an action named `show` that takes care of displaying the dialog:

```
def output = null
withMVCGroup('query') { m, v, c ->
    c.show()
    output = m.choice
}
```

Using this construct liberates you from explicitly destroying the group instance after it has been put to use. Destroying a group will be covered later in this chapter.

6.2.2 Marshaling the MVC type instances

Now that you've told the Griffon framework what MVC group you want to create and given some parameters for this creation, the framework will dutifully run off and create the MVC group and hand it back. Ordinarily, this process is a black box where magic occurs. How Griffon creates the MVC groups may appear magical, but once you know the secret, it isn't. If you look behind the curtain, you see only a few interactions that, when observed without understanding, appear to be magic. But if you know what the interactions are, you can invoke what appears to be deeper magic later.

This section will dive deep into the technical details in order to dispel the magic and also give you a better understanding of what happens when a group is instantiated.

Armed with this information, you should be able to make better choices when configuring groups and making group relationships.

METACLASS PREPARATIONS

When creating the group, the first task the framework accomplishes is loading the Java class for the MVC group members. This is the plain Java part. What happens next is Groovy.

If the artifact or its superclass implements the `griffon.core.GriffonArtifact` interface and the superclass is `Object`, then the Griffon compiler switches the superclass to one that contains the `app` object and four method definitions (`createMVCGroup()`, `buildMVCGroup()`, `withMVCGroup()`, and `destroyMVCGroup()`). If the artifact or its superclass does *not* implement the `griffon.core.GriffonArtifact` interface, the app and the four method definitions are injected directly into the artifact's byte code. The `app` field is a reference to the `GriffonApplication` object that serves as the central touchstone for the whole application, where all of the magic pixie dust is stored. The four injected methods are required for managing MVC groups. The first three methods exist for creating groups and the last one for destroying groups (more on that in the next section). These injections for an MVC member are compulsory; every member gets them. And they will overwrite any existing field or methods by the same name, so don't even try!

The next step for the framework is to create the builder. The builder declarations in `Builder.groovy` have a syntax that allows various (features/behaviors) groups of builder nodes and properties to be injected into specific MVC portions. The default configuration has the threading group of methods being injected into the controller portion of the MVC group. Chapter 7 has more details on how to use those injected methods. But the injection of these properties and methods is under the control of the developer of the application.

INSTANTIATIONS AND INJECTIONS

The next step is for the framework to instantiate each member. Note that until this point, you've been manipulating the metaclasses that represent the members, not actual instances of an object. When creating the instances, the framework creates the objects in true JavaBeans fashion by calling the public no-args constructor on the object class. There's one wrinkle in this step, though. If a parameter passed into the `buildMVCGroup()`, `withMVCGroup()`, or `createMVCGroup()` method matches the name of a member, then you'll use the provided value instead of creating a new one. Section 6.2.4 will discuss when and where you would want to do that.

After the members are initialized, they're stored in the `app` object. The `app` object has a storage facility that exposes `models`, `views`, and `controllers` as properties, and a catchall `groups` property as well. These properties are exposed as maps where each member is keyed by its owner `groupName`: the unique group name, not the group type. Section 6.3 will cover this in greater detail.

The next listing depicts a controller with four properties defined. Two of them have special meaning because they follow the naming conventions for MVC members;

the other two are run-of-the-mill properties. We'll use this example to explain what happens during a group's instantiation, as triggered by the call to the `buildMVC-Group()` method also shown in the listing.

Listing 6.2 Code injection example

```
class SampleController {
  def builder // injected by Griffon
  def model   // injected by Griffon
  def foo     // injected by Griffon
  def bar     // injected by Griffon
}

buildMVCGroup('sample', foo:1, baz:2)
```

Now that Griffon has live objects representing the members of your MVC group, you can start injecting values into the properties. Three types of properties are injected into the portion instances:

- `builder`—The builder that is created as part of the group. This property is easy to spot; it's always called `builder`. You may want to access this field because it has the same variable scope as any of the scripts that make up the MVC group, such as the view script. This is how a controller can peek into elements defined in a view, for example, as long as those elements were either assigned to a variable in the view script or had a value set for their `id` property (a useful trick you might remember from chapter 5).
- *Properties whose names match portion names (*`model`, `view`, *or* `controller`*)*—These properties may vary with the particular setup of your MVC group. The intent of this class of property injections is to allow the portions of the MVC group to see and interact with each other, as if they were created as one object but having distinct identities. For the MVC pattern to work successfully, it's essential that the controller have access to both the model and the view directly. By default, the controllers created by the `create-mvc` script have the necessary fields for injection of the model and the view. In listing 6.2, only the model will be injected, because only the model has a property.
- *Properties whose names match parameter names passed in*—These properties can be tricky to spot. They create wonderfully terse code, but they also can be considered too clever. This is where some good software engineering discipline comes in handy. When you're declaring a property that's meant to be injected, leave a comment around the property stating that fact. In the example, you passed in two named arguments, `foo` and `baz`. You also have two remaining properties in the controller, `foo` and `bar`. The framework will inject the value for `foo` into the controller, but it won't inject the `bar` argument or inject anything into the `baz` property because the names don't match up between the arguments and the properties.

Variables that are injected and passed into scripts and argument maps

- `app`—The `GriffonApplication` instance representing your app. (This will always be present, regardless of whether a property is present.)
- `mvcType`—The type of the MVC group.
- `mvcName`—The name of the MVC group. Must be unique.
- `model`—The model instance.
- `view`—The view script.
- `controller`—The controller instance.
- `<other portions>`—Other portions specified by the MVC group.
- `<other args>`—Anything passed in as a named parameter to `build/with/createMVCGroup`.

But what do you do with arguments that aren't injected and properties that aren't injected to? How can you initialize them? That is an excellent question that we'll address next.

6.2.3 Initializing group members

Another question you may be asking is, "When do I get to play?" We've been discussing how the framework creates some of the core parts of the application, and as yet it has been a mostly hands-off experience. Often, conventions and injection patterns won't fill the bill. Compelling graphics are rarely made via injection, and we've reached the point where the MVC group members take an active role in their life cycle: initialization.

Griffon treats the initialization of the members in one of two wildly different fashions: scripts that are executed and classes that have methods called. You can think of this as the difference between two server-side technologies: JSPs and servlets. When it comes down to executing the bytecode, they are the same thing—a Java class that implements the `javax.servlet.Servlet` interface.

One other detail needs to be reiterated and will likely come into play as you write your members. They are initialized in the order in which they're declared in the `Application.groovy` file. By default the order is model, then view, then controller, but you can change that order if you need to. The most common case for needing to do so is when the controller creates other sub MVC groups and injects portions of the child view into the view binding, so the view can wire those components directly into itself.

Classes and the mvcGroupInit() method

The simpler of the two initialization mechanisms is the means that Griffon applies to MVC portions classes, not scripts. Classes are members that don't implement the `groovy.lang.Script` interface. In the classes instance, the framework looks for a method named `mvcGroupInit()` that takes a single argument `java.util.Map`. The map contains the builder and all the instantiated portions, as well as any named parameters passed into the call to `buildMVCGroup()` or `createMVCGroup()`.

Notice that the contents of the map passed in as the sole argument is the same set of data that Griffon looks at to consider injections. This isn't an accident, because some of the arguments may not be injected into the portion that is being initialized. This is especially true for model portions. The reason is that some of the arguments you pass in may only be needed for initialization and aren't needed for the life of the portion.

SCRIPTS AND SCRIPT EXECUTION

The other type of portion that Griffon encounters is Groovy scripts. Groovy scripts look like extended code snippets but are converted into fully functional classes implementing the `groovy.lang.Script` interface.

When an MVC group member is being initialized and it's a script, the script itself is executed in the context of the builder that has been generated for the MVC group of which it's a member. Even though Groovy scripts can expose methods as if they were object instances, any method named `mvcGroupInit` will be ignored in a script, deferring to the execution of the script.

One of the key differences of an MVC group script execution is that the execution occurs as a desired effect of the `builder` object's `build(Script)` method. When the build method of a Griffon builder is called on a script instance, it does more than simply execute the script. Before the execution, the metaclass of the script instance is manipulated so that when a method is executed or a property is referenced, the builder is given a chance to intercept those calls and use the factories registered in the builder. Because of this metaclass integration, the script can declare the GUI in a context-free fashion. That is, there's no need to prefix nodes with a variable that identifies the real builder type that contributed said nodes; there's also no need for additional imports in many cases.

NOTE The `mvcGroupInit()` method has a counterpart method that we'll discuss later in this chapter.

6.2.4 Advanced techniques

The basic conventions of the declaration and creation of the MVC groups can sometimes betray the subtleness of some of the more advanced techniques that can greatly enhance the usability of the group instantiation facilities. Two of the most powerful are preexisting member instances and multiple view components.

USING PREEXISTING MEMBER INSTANCES

When passing arguments to `buildMVCGroup()`, `withMVCGroup()`, or `createMVCGroup()`, what happens when you pass in an argument whose key turns out to be the same as a member name? In that case, the value provided is used as the member instance. The framework doesn't initialize a new instance of that member but instead uses the value declared by the user directly.

When would you do want to do this? Common scenarios are where a model object is reused across multiple MVC groups. For example, a model representing a weather forecast may be represented with numbers and images in one MVC group and may

also be represented via thermometers and colors in another view. Controllers may also be reused across multiple contexts where several UI buttons may have the same effect, such as in a tool bar view and a menu view.

This isn't without side effects. The preexisting member will still participate in the injection and initialization phases. So new values may be overwritten in the old object, and the `mvcGroupInit()` method will be called multiple times. This can be turned into a positive, however, if the `mvcGroupInit()` method serves to move the items injected into the affected properties into internal collections.

Multiple view components

Another technique that may improve code readability is creating multiple unlinked components in the view script. All the examples included in the Griffon SDK create a single root component that's composed of multiple children components, but nothing in the framework requires this. You could declare several related components that are driven by a single model and controller object.

An example of using multiple components is a master-detail view, where the master table and the detail panel are separate components. The master group defines a view that can display an aggregated snapshot of all elements; its controller most likely has a set of actions that allow you to navigate, edit, create, and delete such elements. When an element detail is required, the detail group comes into play; its job is to display each of the properties of the selected element. The detail's model can hold those properties, and the detail's view knows how to show them on the screen.

Another example is a complex graph and an associated control panel that's used to manipulate the graph parameters. The MVC groups that use the graph can place the two components in any location.

One more possibility is a series of components that represent the underlying data in a cohesive set of more basic components, such as a tree, a list, and a table. The declaring MVC group is free to pick any number of the components to display as it sees fit.

Removing MVC group members

The MVC Pattern in Griffon isn't an absolute requirement but a strongly worded suggestion. But sometimes you may wind up with empty portions. A widget that has no user interaction and merely reflects changes to its model may not need a controller component. The class can be deleted and the reference to the controller can be removed from `Applications.groovy`. Similarly, a group that reflects no data, such as a license dialog, or a group whose state is tracked by subgroups, may not need a model portion.

Creating additional groups

In addition to removing members, sometimes you may want to add members to the group. Two common examples are an action member and an animation member. The Greet example in the Griffon SDK adds an `Actions` member to the main `Greet` group and the login page group. The code for these members consists of Groovy scripts, so it's evaluated with the same builder that evaluates the views.

To create an additional member, you'll need to do three things, all of which follow the pattern set forth in section 6.1:

1. Create a directory to store the member's code: griffon-app/<member>.
2. Create the class or script for the member, and name it `<type><member>`.
3. Edit `Application.groovy` to refer to the new member.

The first step is to create a new directory to store the source code for the new member. This is typically added under griffon-app, and the name of the directory is the name of the member. For example, if you were adding an `actions` member, you would create a directory named `griffon-app/actions`. But the framework won't enforce this—as long as the named class can be found in the classpath at runtime, it will be used. The class could live under the `src/main` directory or under any directory under griffon-app. That's because almost any directory under griffon-app is compiled as though it were a source directory. The exceptions are i18n, resources, and most of conf.

The next step is to create the code file to represent the member. You have two choices: a Groovy script or a traditional class. The Groovy script will be executed by the builder for the member, so it will have access to the same set of methods that the view scripts have access to. It will also share the same binding context as the view script. This is handy when the additional member represents actions or animations for the view. The other option is to create a traditional class for the member. When you follow this path, the same injection and life cycle patterns that apply to models and controllers also apply to the new class. Either way, the class will be mutually injected like all other group members are.

To follow convention, you need to name the class with the member name as a suffix in camel case. For example, Greet[1] names its actions classes `LoginPanelActions` and `GreetActions`. Although the framework won't throw errors if you don't follow this convention, not doing so will create a problem in readability and maintenance of the application, so following the convention is strongly recommended. The following listing illustrates adding actions to the `FilePanel`.

Listing 6.3 Adding an `actions` member to `FilePanel`

```
mvcGroups {
    // MVC Group for "FilePanel"
    'filePanel' {
        model      = 'FilePanelModel'
        actions    = 'FilePanelActions'
        view       = 'FilePanelView'
        controller = 'FilePanelController'
    }
    //...
}
```

[1] Greet can be found in the samples directory of your Griffon installation.

Finally, you need to register the member in the Application.groovy class. If you were going to add an `actions` member to the `FilePanel` group of GroovyEdit, you would add an entry under the `mvcGroups.FilePanel` group referring the `name` actions to the `FilePanelActions` class.

There's one important item you must remember from earlier in this chapter: when declaring members, order matters. The order in which the members are listed is the order in which they will be initialized and executed. If the view is depending on objects declared in the actions script, then the `actions` member must come before the view script.

Now that you understand the basics of MVC groups, let's look at using and managing multiple MVC groups.

6.3 Using and managing MVC groups

Where are we so far? You know what the MVC pattern is, you know how Griffon declares the pattern, and you know how to create an MVC group. Now you're at perhaps the most pertinent part: what do you do with it? As far as patterns go, the MVC pattern requires a fair amount of setup and wiring. But the focus of the pattern is about how the pieces interact in a running environment. In this section, you'll see how to access multiple MVC groups and destroy MVC groups when the application is finished with them.

6.3.1 Accessing multiple MVC groups

How does an MVC group interact with the world? MVC groups, after all, are a lot like atoms. They can do some interesting stuff all by their lonesome, but the real fireworks occur at the molecular level when several atoms are combined. There are entire classes in upper-level chemistry devoted to interesting combinations of carbon, oxygen, hydrogen, and nitrogen. The trick is getting them to interact with each other in specific ways.

Interacting with other MVC groups is easy. You access properties and methods on the members (model, view, and controller) that form the other MVC groups you wish to interact with. The difficult part is getting hold of the portions from the other MVC groups. There are two ways to obtain the other group portions: access them by their MVC group name, or track the relevant members at the time the MVC group is created, either by having the creating group store the created group's members or by passing in the creating group's members as parameters on the call to `createMVCGroup()`, `withMVCGroup()` or `buildMVCGroup()`.

Accessing via references

Here's a snippet from the Greet sample application of the parent MVC group working with the contents of a child MVC group. The parent group owns the following snippet, and the `userPane` variable holds a reference to the child group:

```
def userPaneGroup = buildMVCGroup('userPane', mvcName,
    user:twitterService.userCache[username], closable:true);

view.tweetsTabbedPane.addTab("@$username", userPaneGroup.view.userPane)
```

Here, the controller is generating an MVC group for a user tab. After the MVC group is created, the controller takes the relevant widget from the new view and adds it to a tabbed pane in the parent's view.

You can also use this technique in reverse: the child group does the adding, and the parent fires and forgets. Here is how GroovyEdit does a similar action. First, in `GroovyEditController` you find

```
createMVCGroup('filePanel' mvcId,
    [file: file, tabGroup: view.tabGroup, tabName: file.name, mvcId: mvcId])
```

Next, in `FilePaneView` you encounter

```
tabbedPane(tabGroup, selectedIndex: tabGroup.tabCount) {
   panel(title: tabName, id: "tab") {
       //....
   }
}
```

The controller for the parent MVC group passes in an instance of a `TabbedPane` that it wants the child MVC group to add itself to. The child MVC group then uses the tabbed pane that was passed in as a value to its own `tabbedPane` widget. A new pane isn't created; instead, the node will use the existing widget. This technique works with most container types built by SwingBuilder.

ACCESSING VIA NAMES

Another method is to access groups via a symbolic name. Remember the second, optional parameter on `buildMVCGroup()`, `withMVCGroup()`, and `createMVCGroup()`? This is why it exists; it's an application-wide name that can be used to access the group's parts without having to hold a direct reference to them. This frees the MVC groups from having to store direct references to the other MVC groups.

The object that is used to access the portions is the `app` property. This property is unique from the other injectable properties in that it's always available in MVC group members because Griffon injects its value at the metaclass level. It also implements the `griffon.core.GriffonApplication` interface.

The relevant fields for MVC groups found in the aforementioned interface are the `groups` property and the `models`, `views`, and `controllers` properties. The `groups` property is a map containing all instantiated groups. Each group is keyed to the names of the MVC groups that have been created. Each MVC group member is keyed by name in its particular MVC group. The `models`, `views`, and `controllers` properties are quick-access properties that are keyed on the MVC group name and return the individual `model`, `view`, and `controller` of each group. If a member for the particular type doesn't exist, it isn't stored and a null is returned if the member key is accessed. These properties exist strictly as a convenience because they can be accessed from the `groups` property.

The Griffon SDK includes another sample application called WeatherWidget (look for it under `$GRIFFON_HOME/samples/WeatherWidget`). One of its controllers initializes four additional groups, as shown in the following listing.

Listing 6.4 `WeatherWidgetController` instantiating and using groups

```
void mvcGroupInit(Map args) {
    createMVCGroup('smallForecast', 'small1')
    createMVCGroup('smallForecast', 'small2')
    createMVCGroup('smallForecast', 'small3')
    createMVCGroup('smallForecast', 'small4')
}

 (1..4).each {
    def day = forecastData.simpleforecast.forecastday[it]
    def smallModel = app.models["small$it"]

    smallModel.day = day.date.weekday
    // ... other such updates
}
```

The parent `WeatherWidgetController` creates four instances of the `smallForecast` group and names them in a standard fashion. This explicit name is used in two places: in the update logic where you access model members by name, and in the view for the MVC group where you add the view widgets based on the explicit names given in the controller (see the following listing).

Listing 6.5 `WeatherWidgetView` wiring up views from the instantiated views

```
hbox {
    widget(app.views.small1.smallPanel)
    hstrut(6)
    widget(app.views.small2.smallPanel)
    hstrut(6)
    widget(app.views.small3.smallPanel)
    hstrut(6)
    widget(app.views.small4.smallPanel)
}
```

This example also touches on a relevant point: when interacting with other MVC groups, you won't always do so at startup. Some interactions need to cross the MVC group barriers. Often, the best place for building such interactions is the `mvcGroupInit()` method of an artifact such as a controller. You can also use the life cycle scripts to build such group relationships. Finally, there's the option for fine-grained event listeners, but we won't explore that subject until chapter 8.

Like all models with life cycles, the MVC group life cycle has to deal with death and destruction, as we'll discuss next.

6.3.2 Destroying MVC groups

A well-behaved application that runs for a long time will need to destroy at least some of the MVC groups it creates. These groups may represent things such as transient dialogs or documents that the user has closed. But if these MVC groups are kept around and none of their resources are reclaimed, eventually you'll run out of memory.

The `destroyMVCGroup()` method is used to destroy an MVC group. It takes only one argument: a `String` that is the name of the MVC group (this is why `mvcName` is one of the auto-injected variables). Figure 6.1 illustrates how this method works.

Calling `destroyMVCGroup()` will perform all the needed cleanup actions such as removing event listeners, invoking destroy callbacks, and disposing of Swing components. The first action that the destroy method takes ❶ is to remove any application event listeners that the controller object may have registered with the framework. When you're tearing the group apart, you don't want the framework asking it to respond to any actions it may not be prepared to deal with any more.

The next step in destruction is to call `mvcGroupDestroy()` on each of the MVC group members ❷. Dispatching these methods is done mostly in the same manner as calls to `mvcGroupInit()`. The only real difference is that there are no arguments to this method. As in the `init()` call, the members are inspected in the order in which they're declared in the Application.groovy file. Each instance is examined for a method named `mvcGroupDestroy()` that takes no arguments. Regardless of whether the method terminates normally or throws an exception, the destruction of the group will continue.

The next step is to call the `dispose()` method on the builder object ❸. This will cause the `dispose()` method to be called on all the registered builders as well. Two important cleanup activities for SwingBuilder occur during this process. First, any frame, dialog, or window that was created using the SwingBuilder APIs will have its

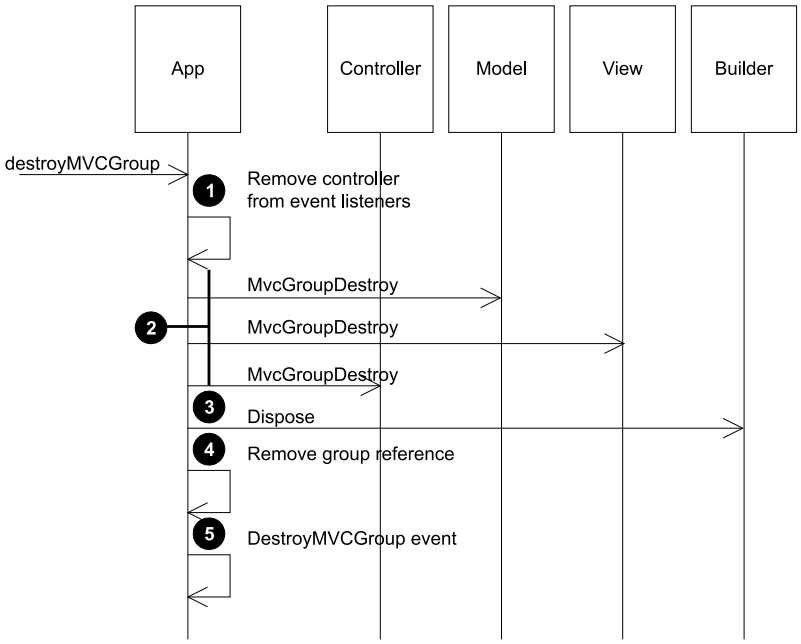


Figure 6.1 `destroyMVCGroup()` sequence

dispose method called. This not only closes those windows but also frees up any native resources that were consumed by them. Second, all bindings created with this builder are unbound via the `unbind()` call. Hence, any listeners that were registered to facilitate the binding will be removed.

Now you un-register the MVC group members from their registration in the `GriffonApplication` object ❹. Why is this step saved until near the end? To allow the `mvcGroupDestroy()` methods and disposal closures to reference the other portions by name if required. Only when all the other cleanup activity has been performed can the references to the members that make up the MVC group be removed.

The final step is to trigger an application-wide `DestroyMVCGroup` event notifying observers that the group has been destroyed ❺. This allows for additional cleanup that other components might desire to perform. Be aware that at this point, the group is no longer valid, and any references to its members have been purged from the application's cache. Any component that listens to this event should also purge any references it holds to the destroyed group or its members.

We've covered all runtime aspects of MVC group. Now it's up to you to experiment with the options at your disposal. Mixing the ability to create new groups with the introspection capabilities of the Artifacts API should give you enough ammo to build complex interactions with concise and readable code.

We're almost done with this chapter, but we can't close it without mentioning another useful feature of the Griffon framework. When it comes to creating MVC groups, custom artifact templates can play a significant role.

6.4 Creating custom artifact templates

Recall from section 6.1 that the files of each MVC member are initially created with a predetermined set of options. Well, each file is the result of a template being evaluated within the boundaries of a set of conventions.

Wouldn't it be great if it were possible to override the default template of a certain artifact before its corresponding file was created? It turns out, it's possible. Suppose you want to create a view that constructs a `TabbedPane` or a `Dialog`. If you have such a template ready, then you can use it instead of the default view template. This saves you the time of editing the freshly created view in order to replace the default code and paste what you need. Remember that Griffon favors convention over configuration, but that doesn't mean it lets go of configuration altogether.

Let's follow a pragmatic approach to learning about the artifact templating options by building a custom group whose main responsibility is to show an error dialog whenever an unknown error occurs. To give you an idea of what you'll end up with, glance at Figure 6.2.

The first order of business will be to build the template of each MVC member.

Figure 6.2 A dialog with custom title, icon, and message. The message's text should change according to the error that occurred.

6.4.1 Templates, templates, templates

A template in Griffon is a simple text file that may contain special placeholders for specific variables. You may have noticed that every time you create a controller, the class defined in the file matches the name of the file. A similar thing happens with the package name for the class. Here, for example, is the template used by default for a model class:

```
@artifact.package@import groovy.beans.Bindable

class @artifact.name@ {
   // @Bindable String propName
}
```

You can see two different placeholders: one for the package name and one for the class name. Table 6.1 lists all the available placeholders.

Table 6.1 Placeholders and their meaning

Placeholder	Meaning
`artifact.package`	Defines the name of the package to be used in the class. May be empty.
`artifact.name`	Name of the artifact, including suffix. Example: `BookController`.
`artifact.name.plain`	Name of the artifact, excluding suffix. Example: `Book`.
`artifact.superclass`	Name of the superclass, if any was defined (use `-super-class` command flag).
`griffon.version`	Griffon version currently being used. Example: `0.9.5`.
`griffon.project.name`	Name of the current project.
`griffon.project.key`	Same as the previous, but uses a dot (`.`) instead of a slash as a path separator.

OK, you know what placeholders can be used, but how do you name your templates? And where do you put them? The answer to the first question is found in the naming convention that you know by heart by now. A view class is constructed with a view suffix, and a model class is constructed with a model suffix. This probably means there are `View.groovy` and `Model.groovy` files lying somewhere in the Griffon SDK.

If you guessed this too, you're on the correct path. This leads to the second answer: Griffon is looking at a specific location for its own templates. That location could be configurable, or it could possibly allow for several locations to be specified. Well, it turns out the second guess is correct: there are several locations to be searched for templates. Any additional artifact templates can be placed under `src/templates/artifacts`—this includes both applications and plugins (we'll cover plugins in chapter 11).

Armed with this knowledge, you can build the templates for each member of your group. Let's call them `DialogView`, `DialogModel`, and `DialogController`; we'll present them in this order. The next listing presents the template for the view. It's sparse, yes, but you don't need much for this type of view

Listing 6.6 Template definition for DialogView.groovy

```
@artifact.package@import javax.swing.JOptionPane
optionPane(
    id: 'pane',
    messageType: JOptionPane.INFORMATION_MESSAGE,
    optionType: JOptionPane.DEFAULT_OPTION,
    icon: nuvolaIcon('core', category: 'apps', size: 64),
    message: bind {model.message})
```

The template sets up an `optionPane` node (it resolves to an instance of `JOptionPane`) with a couple of properties. Note the use of the `nuvolaIcon()` node. This node is supplied by a plugin and as such isn't available in a view unless you install the corresponding plugin in the target application that uses this template. If you're curious, this is done by invoking the following command:

```
$ griffon install-plugin nuvolaicons
```

This view template also sets up a binding with its model, which means you need to supply an observable property on the group's model as well. The next listing shows what the model template defines.

Listing 6.7 Template definition for DialogModel.groovy

```
@artifact.package@import groovy.beans.Bindable
class @artifact.name@ {
    @Bindable String message = ''
    @Bindable String title = 'Error'
}
```

The group you're building is simple, yet the model defines two properties that can be used to provide a more personal touch when put to work.

The third and last template we'll cover belongs to the controller member of the group. The controller is responsible for showing a dialog with the appropriate owner and setting the title on the dialog (see the following listing).

Listing 6.8 Template definition for DialogController.groovy

```
@artifact.package@ import java.awt.Window

class @artifact.name@ {
    def model
    def view

    def show = { Window window = null ->
        view.pane.createDialog(
            window ?: Window.windows.find{it.focused},
            model.title
        ).show()
    }
}
```

Once more you see the trick of defining a value for a parameter. This means the `show()` action of a controller created with this template can be called with a `Window` argument or with no argument at all. Either way, the dialog will be presented at the center of the currently focused `Window`.

Make sure you place these three files inside src/templates/artifacts at the root of your application. You're ready to bring this group to life.

6.4.2 It's alive!

Up to this point, we've covered the setup of the templates and their conventions. Now you can create your first MVC group from custom templates. In chapter 2, we discussed several of the command targets at your disposal, in particular the `create-*` command targets. This command target bootstraps a particular artifact using predefined template-naming conventions. The `create-model` command creates a model, the `create-view` command view creates a view, and `create-controller` creates a controller using standard templates. You can override those conventions and make the command targets do your bidding.

Every `create-*` command target requires an artifact type in order to work. Through this type, you can inject your custom templates and let them be resolved instead of the default ones. If a view requires a `View.groovy` template and has *view* as its suffix, it's likely that its type is `view` too. Follow the same train of thought for the other artifacts.

You now have a custom template that you want to use. How do you put this template together with the command line?

The `create-*` command targets accepts an optional parameter that matches the lowercase name of the type it can manage. For instance, if you want to target only views, then you define a `-view` flag and the name of the target template as its values.

Let's put this new knowledge to the test. The following command invocation is enough to create a group that uses all of your newly defined templates:

```
griffon create-mvc -view=DialogView \
                   -model=DialogModel \
                   -controller=DialogController sample
```

You should see output like this in your console after a few moments:

```
Running script /usr/local/griffon/scripts/CreateMvc.groovy
Environment set to development
Created DialogModel for Sample
Created DialogView for Sample
Created DialogController for Sample
Created IntegrationTests for Sample
```

Perfect! Now open each MVC member file in your favorite editor. Notice how every placeholder has been replaced with the correct value for each artifact.

Last, you need to instantiate this group in your code. Using the knowledge you've gained from previous sections of this chapter yields the following code:

```
def (m, v, c) = createMVCGroup('sample')
m.message = """
  Oops! An unexpected error has occurred :-(
  $exception
""".toString()
c.show()
destroyMVCGroup('sample')
```

The caller code construct an instance of the `sample` group, sets a value for the model's `message` property, and calls the `show()` action on its controller. Finally, it performs proper cleanup and deletes the group instance from the app's cache after the dialog has been dismissed.

One final remark about templates: you can also place them inside plugins. This means you can reuse the templates across several applications.

6.5 Summary

The MVC group facility is the means by which the Griffon framework encourages and rewards adherence to the Model-View-Controller pattern. It also makes using MVC groups the path of least resistance when it comes to creating GUIs—this resistance has often been the main reason the pattern was abandoned or ignored.

Griffon provides a bootstrapping mechanism to create the initial content of the MVC group artifacts. Each artifact follows both a naming and a place convention that can be exploited later to mix and match different values for each of its members.

An MVC group is formed from a model, a view, and a controller. You can create groups with fewer or more members, or reuse existing member instances between groups.

Griffon provides both a declarative means to create MVC groups and rich means to interact with the MVC groups once they're created. Groups can be created, initialized, and destroyed at will.

Finally, you caught a glimpse of the templating mechanism the framework uses to create each file. Because the mechanism is powered by conventions, it's possible to override the selection of the default templates and instruct the system to prefer a custom template of your choosing.

This concludes our journey through the basic building blocks of the Griffon framework. The next chapter will discuss the challenges of concurrency and threading, and how Griffon helps solve them.

7 Multithreaded applications

This chapter covers:

- Understanding the need for well-behaved multithreaded desktop applications
- Working with Griffon's multithreading facilities for Swing-based applications
- Additional UI toolkit-agnostic threading facilities for all components

We're halfway through our journey of discovering what Griffon has to offer to desktop application development. By now you should be familiar with the core concepts, such as the MVC pattern and the way Griffon implements it, its command utilities, and its configuration options, just to name a few. But there's more to Griffon than that. For example, you can extend the framework's capabilities to upgrade an application's looks. And you can certainly create an application that's prepared to deal with the terrible beast that is concurrent programming in the JVM.

Handling threading in desktop applications is a crucial task. Do it carelessly, and your users will walk away. Continue reading to find out the secrets for mastering multithreaded applications with Griffon.

7.1 The bane of Swing development

As noted back in chapter 1, the JVM is a great place to develop applications for many reasons. It's an amazing piece of technology that harbors a huge ecosystem of libraries, tools, and languages. The designers of the JVM and the Java language made a conscious decision to include multithreading support since the beginning, thus allowing a wide range of applications to be built. Although many agree that building an application with concurrency in mind isn't an easy job, it beats what we had before the JVM came forward.

But writing desktop applications with Swing while taking concurrency into account is a different game altogether, and it can get ugly pretty fast. It's easy to make the wrong assumptions when working with multithreaded applications. The threading support found in the JVM and the Java programming language is certainly welcome, but it's not enough by itself.

Swing is a powerful UI toolkit capable of handling a multitude of events at a time. It handles events generated by user interaction, such as the push of a button, the movement of the mouse over a component, or keyboard input. Swing also handles internal events generated by the application and by its own internals, such as a repaint request of a particular section of the UI. Swing handles the load by placing all events in a queue and later dispatching them in the same order they entered the queue. The responsibility of dispatching events falls to `java.awt.EventQueue`, and event dispatching must be performed in a special thread: the event dispatch thread (EDT).

It so happens that the majority of events posted to the event queue are paint requests; the rest deal with updates that may affect the UI state, possibly triggering another paint request as a byproduct. This means the `EventQueue` is busy properly updating the visuals of your application most of the time.

Can you imagine what happens when an operation that isn't UI-related, such as reading a big file or querying a database, is executed in the EDT? In no particular order: disaster, despair, frustration, and a lot of stress, mainly because the EDT should be concerned with UI-related operations. Everything else should be run outside of the EDT.

Without further ado, let's review what plain Java and Swing offer in terms of threading support. We'll use this basic understanding to showcase the advantages of handling threading concerns with Groovy and SwingBuilder. Later we'll show you how Griffon uses these new building blocks and adds a few of its own.

7.1.1 Java Swing without threading

Let's review a typical Swing example, as shown in listing 7.1. Chances are you've encountered a similar piece of code around the web when searching for information about Swing. This application is a simple file viewer. We could have chosen a network- or database-aware application, but accessing the file system through I/O is a common task and requires less setup. The implementation is straightforward, but we'll explain a little about the code.

Listing 7.1 Java version of `SimpleFileViewer`

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

public class SimpleFileViewer extends JFrame {
  public static void main(String[] args) {
    SimpleFileViewer viewer = new SimpleFileViewer();
  }

  private JTextArea textArea;
  private static final String EOL = System.getProperty("line.separator");

  public SimpleFileViewer() {
    super("SimpleFileViewer");
    buildUI();
    setVisible(true);
  }

  private void buildUI() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JButton button = new JButton("Click to select a file");
    textArea = new JTextArea();
    textArea.setEditable(false);
    textArea.setLineWrap(true);
    button.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event) {
        selectFile();
      }
    });
    getContentPane().setLayout( new BorderLayout() );
    getContentPane().add(button, BorderLayout.NORTH);
    getContentPane().add(new JScrollPane(textArea), BorderLayout.CENTER);
    pack();
    setSize( new Dimension(320, 240) );
  }

  private void selectFile() {
    JFileChooser fileChooser = new JFileChooser();
    int answer = fileChooser.showOpenDialog(this);
    if( answer == JFileChooser.APPROVE_OPTION ) {
      readFile(fileChooser.getSelectedFile());
    }
  }

  private void readFile( File file ) {
    try {
      StringBuilder text = new StringBuilder();
      BufferedReader in = new BufferedReader(new FileReader(file));
      String line = null;
      while( (line = in.readLine()) != null ) {
        text.append(line).append(EOL);
      }
      textArea.setText(text.toString());
      textArea.setCaretPosition(0);
```

1 Create subclass

2 Instantiated outside EDT

3 Define event handler

4 Executed in EDT

5 Doesn't care about EDT

```
    } catch( IOException ioe ) {
      ioe.printStackTrace();
    }
  }
}
```

The code starts by defining a subclass of `javax.swing.JFrame` ❶. In the good old days, inheritance was regarded as the way to go most of the time; these days, people tend to favor composition. Although this particular aspect doesn't hinder you in creating a multithreaded Swing application, it imposes a design constraint that might affect you later. For example, any object that interacts with an instance of `SimpleFileViewer` could potentially change the layout, add or remove components from it, or even alter the button's behavior; all these are possible because the contract of a UI container (such as `JFrame`) is exposed via the inheritance mechanism.

Another recurring idiom in Swing applications is the usage of inner classes to quickly define precise event handlers ❸. This is a powerful mechanism that tries its best to be as friendly as anonymous functions are in other languages. They pretty much behave like anonymous functions, other than their verbosity and some limitations to enclosing scope and visibility. But we're not about to discuss the merits of Java's inner-class design versus real anonymous functions.

The catch is that while dealing with the amount of verbosity needed to wire up these handy classes, you may forget about the intricacies of Swing threading. In case you were wondering, event listeners receive a notification in the same thread that sent them the message—that would be the EDT. This means you have to plan for `selectFile()` ❹ to be called in the EDT. As listing 7.1 shows, there's no indication that the code is aware of this fact.

Because the implementation of `selectFile()` doesn't display any special thread-handling code, it's safe to assume that it will be executed in the same thread that called it. This method will call `readFile()` ❺ if the user has selected a file to read. If there were any special code to properly handle threading, this would be it. Sadly, that code isn't there. The implementation will read the file's contents in a buffer and update the `textArea`, `text`, and `caretPosition` properties.

Finally, at the application's main entry point ❷, an instance of the `SimpleFileViewer` class is created in the main thread. But there's another catch: it's recommended that Swing components be initialized in the EDT. In this case, initialization occurs in the main thread.

Now that you know some of the pitfalls you need to avoid, the next section will show how you can sort them out with plain Java.

7.1.2 Java Swing with threading

Let's look now at a revised version that addresses the previously outlined problems: you use composition instead of inheritance, you make sure `readFile()` is protected against reading a file in the EDT, and you initialize all Swing components off the main thread (see the following listing).

Listing 7.2 `RevisedSimpleFileViewer` with threading taken into account

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

public class RevisedSimpleFileViewer {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        RevisedSimpleFileViewer viewer = new RevisedSimpleFileViewer();
      }
    });
  }

  private JTextArea textArea;
  private JFrame frame;
  private static final String EOL = System.getProperty("line.separator");

  public RevisedSimpleFileViewer() {
    frame = new JFrame("RevisedSimpleFileViewer");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(new BorderLayout());
    frame.getContentPane().add(buildUI(), BorderLayout.CENTER);
    frame.pack();
    frame.setSize( new Dimension(320, 240) );
    frame.setVisible(true);
  }

  private JPanel buildUI() {
    JPanel panel = new JPanel(new BorderLayout());
    JButton button = new JButton("Click to select a file");
    textArea = new JTextArea();
    textArea.setEditable(false);
    textArea.setLineWrap(true);
    button.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event) {
        selectFile();
      }
    });
    panel.add(button, BorderLayout.NORTH);
    panel.add(new JScrollPane(textArea), BorderLayout.CENTER);
    return panel;
  }

  private void selectFile() {
    JFileChooser fileChooser = new JFileChooser();
    int answer = fileChooser.showOpenDialog(frame);
    if( answer == JFileChooser.APPROVE_OPTION ) {
      readFile(fileChooser.getSelectedFile());
    }
  }

  private void readFile( final File file ) {
    new Thread(new Runnable(){
      public void run() {
```

1 **Run code in EDT**

2 **Use composed JFrame**

3 **Executed in EDT**

4 **Read file outside EDT**

```
        try {
          final StringBuilder text = new StringBuilder();
          BufferedReader in = new BufferedReader(new FileReader(file));
          String line = null;
          while( (line = in.readLine()) != null ) {
           text.append(line).append(EOL);
          }
          SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              textArea.setText(text.toString());
              textArea.setCaretPosition(0);
            }
          });
        } catch( IOException ioe ) {
          ioe.printStackTrace();
        }
      }
    }).start();
  }
}
```

Although it's not our intent to scare you off by incrementing the verbosity level, this is how inner classes can be used to cope with threading problems. You may have noticed that 15 lines have been added in the process of converting `SimpleFileViewer` to a well-behaved Swing application.

From top to bottom this time: A Swing utility class ❶ is used to explicitly invoke a piece of code in the EDT. You've now ensured that the UI is built in the correct thread. A `JFrame` internal variable is used because `RevisedSimpleFileViewer` doesn't inherit from `JFrame` ❷. Notice that the button's event handler ❸ has been left untouched; you'll make sure the helper methods receive proper threading updates as shown by ❹ and ❺. A new thread is spun off ❹, effectively executing the file-read code outside of the EDT; but once the contents have been finished, you need to update the `textArea`. It's back then into the EDT by using `SwingUtilities.invokeLater()` again. This particular Swing facility posts a new event into the event queue. On the other hand, `SwingUtilities.invokeAndWait()` executes code in the EDT and waits until code execution has finished. In other words, `invokeLater()` is an asynchronous call to the EDT, whereas `invokeAndWait()` is a synchronous one.

As you've seen, the EDT is both a blessing and a curse. On one hand, it makes the job of dispatching events in a serialized way a reality. On the other hand, it spells disaster if developers don't take special care to avoid executing long-running operations on it.

More information about threading

Oracle keeps tutorials online if you'd like to learn more about Java technologies. One of these tutorials is "Concurrency in Swing" (http://mng.bz/dKpA). This particular document tells you everything you need to know about threading and Swing. We did our best to sum up the contents and show you the common pitfalls.

You're here to learn about making the job of building Swing applications easier and fun, so let's look at the alternatives that Groovy offers.

7.2 SwingBuilder alternatives

Up to this point, you've seen numerous examples of Groovy's SwingBuilder. You know it reduces visual clutter and increases readability, but that doesn't mean you're safe from shooting yourself in the foot just by relying on SwingBuilder alone. Groovy can work its magic, but you have to give it a few nudges in the right direction from time to time.

In this section, we'll revisit SwingBuilder for its Swing DSL capabilities, but this time we'll journey a bit deeper into the threading capabilities it exposes.

7.2.1 Groovy Swing without threading

Let's revisit the first version of `SimpleFileViewer` (see listing 7.1), but switch to Groovy. As shown in the following listing, the code's design is pretty much the same apart from using composition instead of inheritance from the get-go.

Listing 7.3 Groovy version of `SimpleFileReader`

```
import groovy.swing.SwingBuilder
import javax.swing.JFrame
import javax.swing.JFileChooser

public class GroovyFileViewer {
  static void main(String[] args) {
    GroovyFileViewer viewer = new GroovyFileViewer()
  }

  private SwingBuilder swing

  public GroovyFileViewer() {
    swing = new SwingBuilder()
    swing.fileChooser(id: "fileChooser")
    swing.frame( title: "GroovyFileViewer",
                 defaultCloseOperation: JFrame.EXIT_ON_CLOSE,
                 pack: true, visible: true, id: "frame" ) {
      borderLayout()
      button("Click to select a file", constraints: context.NORTH,
             actionPerformed: this.&selectFile)
      scrollPane( constraints: context.CENTER ) {
        textArea( id: "textArea", editable: false, lineWrap: true )
      }
    }
    swing.frame.size = [320,240]
  }

  private void selectFile( event = null ) {
    int answer = swing.fileChooser.showOpenDialog(swing.frame)
    if( answer == JFileChooser.APPROVE_OPTION ) {
      readFile(swing.fileChooser.selectedFile)
    }
  }
```

1 Instantiated outside EDT

2 Converts method to closure

```
  private void readFile( File file ) {
    swing.textArea.text = file.text
    swing.textArea.caretPosition = 0
  }
}
```

❸ Executed in EDT

Comparing lines of code, listing 7.3 is 20 lines shorter than listing 7.1, but it still lacks proper threading support. The application is still being instantiated in the main thread ❶, which means the UI components are also being instantiated and configured in the main thread. Next a Groovy shortcut ❷ obtains a closure from an existing method, a handy alternative to declaring an inlined closure. Back to the application's business end ❸, where the file is read in the current executing thread (that would be the EDT again), and the `textArea` is updated in the same thread. In summary, you made the code more readable but didn't rid yourself of the threading problems.

You could make a Groovy version of listing 7.2 too, using SwingUtilities and creating new threads, but Groovy has a tendency to simplify things. We'd like to show you what the language has to offer before we get into SwingBuilder threading proper.

7.2.2 Groovy Swing with threading

The ability to use closures in Groovy is a big selling point to many developers, but the ability to use Groovy closures with plain Java classes is perhaps the most compelling reason to switch.

Take a moment to look at the Java implementation of `RevisedSimpleFileViewer.readFile()` in listing 7.2. Try to visualize its behavior behind the verbosity of its implementation. Now look at the Groovy version of the same code, as shown in the next listing; it's a rather different picture.

Listing 7.4 Groovy enhanced `readFile()`

```
private void readFile( File file ) {
  Thread.start {
    String text = file.text
    SwingUtilities.invokeLater {
      swing.textArea.text = text
      swing.textArea.caretPosition = 0
    }
  }
}
```

We can almost hear you shouting with anger and disbelief, "Not fair!" (that is, if you're one of the countless developers who have been bitten by Swing threading problems in the past). Otherwise it's probably something like, "I didn't know I could do that!"

Among the several tricks Groovy has in its arsenal is one that is helpful with threading. You see, Groovy can translate a closure into an implementation of a Java interface that defines a single method. In other words, Groovy can create an object that implements the `Runnable` interface: the behavior of its `run()` method is determined by the closure's contents. In listing 7.4, a new thread is created, the file's contents are read in that thread, and then you're back into the EDT to update the `textArea`'s properties.

This chapter would be over right now if these were the only things Groovy had to offer regarding threading. Luckily, that isn't the case. SwingBuilder also has some tricks up its sleeve. The next listing shows the full rewrite of `GroovyFileReader` using SwingBuilder's threading facilities.

Listing 7.5 SwingBuilder threading applied to `GroovyFileReader`

```
import groovy.swing.SwingBuilder
import javax.swing.JFrame
import javax.swing.JFileChooser

public class RevisedGroovyFileViewer {
  static void main(String[] args) {
    def viewer = new RevisedGroovyFileViewer()
  }

  private SwingBuilder swingBuilder

  public RevisedGroovyFileViewer() {
    swingBuilder = new SwingBuilder()
    swingBuilder.edt {
      fileChooser = fileChooser()
      frame( title: "RevisedGroovyFileViewer",
             defaultCloseOperation: JFrame.EXIT_ON_CLOSE,
             preferredSize: [320, 240],
             pack: true, visible: true, id: "frame" ) {
        borderLayout()
        button("Click to select a file", constraints: context.NORTH,
               actionPerformed: this.&selectFile)
        scrollPane( constraints: context.CENTER ) {
          textArea( id: "textArea", editable: false, lineWrap: true )
        }
      }
    }
  }

  private void selectFile( event = null ) {
    def fileChooser = swingBuilder.fileChooser
    int answer = fileChooser.showOpenDialog(swingBuilder.frame)
    if(answer == JFileChooser.APPROVE_OPTION) {
      readFile(swingBuilder.fileChooser.selectedFile)
    }
  }

  private void readFile( File file ) {
    swingBuilder.doOutside {
      String text = file.text
      doLater {
        textArea.text = text
        textArea.caretPosition = 0
      }
    }
  }
}
```

1 Executed outside EDT

2 Convert method into closure

3 Executed in EDT

4 Frame set in builder's context

5 FileChooser set on builder's context

1 Executed outside EDT

3 Executed in EDT

Judging by ❶, the application is still being instantiated on the main thread; this would mean UI components are also being instantiated in that thread, but ❷ says otherwise. At that line, SwingBuilder is being instructed to run the code by making a synchronous call to the EDT, thus ensuring that UI building is done in the correct thread. Notice at ❸ and ❹ that because `fileChooser` and frame variables are tied to the builder's context, there's no longer a need to define external variables to access those components if you keep a reference to the builder (which you do). This design choice is taken into account where the builder's instance is used to get a reference to both `fileChooser` and `frame` ❺.

The more interesting bits can be found at ❶ and ❸. Calling `doOutside{}` on SwingBuilder has pretty much the same effect as calling `Thread.start{}` except there is a slight but important difference: the SwingBuilder instance is used as the closure's delegate. This means you can access any variables tied to that particular instance and any SwingBuilder methods in the closure's scope. That's why the next call to the SwingBuilder threading method ❸ `doLater{}` doesn't need to be prefixed with the `swing` variable.

But you can go further by combining `selectFile()` and `readFile()` in a single method, and by encapsulating the method body with a special Groovy construct, as shown in the next listing.

Listing 7.6 Simplified version of `selectFile()` and `readFile()`

```
private void selectFile( event = null ) {
  swingBuilder.with {
    int answer = fileChooser.showOpenDialog(frame)
    if(answer == JFileChooser.APPROVE_OPTION) {
      doOutside {
        String text = fileChooser.selectedFile.text
        doLater {
          textArea.text = text
          textArea.caretPosition = 0
        }
      }
    }
  }
}
```

This is quite the trick. By using the `with{}` construct, you're instructing Groovy to override the closure's delegate. In this case, its value will be the SwingBuilder instance your application is holding. This means the closure will attempt to resolve all methods and properties not found in `RevisedGroovyFileViewer` against the SwingBuilder instance. That's why the references to `fileChooser` and `name`, as well as method calls to `doOutside{}` and `doLater{}`, need not be qualified with the SwingBuilder instance. Sweet!

> **TIP** The trick related to the use of the `with{}` construct can be applied to any object—it isn't SwingBuilder specific. This can greatly simplify the code you place on a controller or service, for example.

Now let's take a moment to further inspect SwingBuilder's threading facilities.

7.2.3 Synchronous calls with edt

We have already established that code put in the `edt{}` block is executed directly in the EDT—that is, the EDT thread will block until the code has completed its execution. This appears to be no different from explicitly calling `SwingUtilities.invokeAndWait{}`, but in reality there are three important things to consider.

The first improvement found in `edt{}` is that the current SwingBuilder instance is set as the closure's delegate, so you can call any SwingBuilder methods and nodes directly without needing to qualify them with an instance variable.

The second convenience has to do with a particular rule of executing code in the EDT. Once you're executing code in the EDT, you can't make an explicit call to `SwingUtilities.invokeAndWait{}` again. A nasty exception will be thrown if you do make the call. Let's see what happens when you naively make a synchronous call in the EDT when you're already executing code in that thread. The following listing displays this example. Remember that the button's `ActionListeners` are processed in the EDT.

Listing 7.7 Violating EDT restrictions

```
import groovy.swing.SwingBuilder
import javax.swing.SwingUtilities

def swing = new SwingBuilder()
swing.edt {
  frame(title: "Synchronous calls #1", size: [200,100], visible: true) {
    gridLayout(cols: 1, rows:2)
    label(id: "status")
    button("Click me!", actionPerformed: {e ->
       status.text = "attempt #1"
       SwingUtilities.invokeAndWait{ status.text = "attempt #2" }
    })
  }
}
```

Synchronous call in EDT ❶

You were so infatuated with Groovy threading that you forgot for a moment that `SwingUtilities.invokeAndWait{}` can't be called in the EDT! If you run the application and click the button, an ugly exception like the following is thrown:

```
Caused by: java.lang.Error: Cannot call invokeAndWait from the event
     dispatcher thread
        at java.awt.EventQueue.invokeAndWait(EventQueue.java:980)
        at javax.swing.SwingUtilities.invokeAndWait(SwingUtilities.java:1323)
        at javax.swing.SwingUtilities$invokeAndWait.call(Unknown Source)
```

But if you rely on `SwingBuilder.edt{}` to make the synchronous call at ❶ as shown in the following listing, you get a different result: a working, bug-free application!

Listing 7.8 EDT restriction no longer violated

```
import groovy.swing.SwingBuilder

def swing = new SwingBuilder()
swing.edt {
```

```
  frame(title: "Synchronous calls #2", size: [200,100], visible: true) {
    gridLayout(cols: 1, rows:2)
    label(id: "status")
    button("Click me!", actionPerformed: {e ->
       status.text = "attempt #1"
       edt{ status.text = "attempt #2" }
    })
  }
}
```

Much better. The name change isn't that hard to remember, is it?

The third and final difference is that `edt{}` is smart enough to figure out whether it needs to make a call to `SwingUtilities.invokeAndWait{}`. If the currently executing code is already on the EDT, then it will continue to be executed in the EDT; `edt{}` makes a call to `SwingUtilities.isEventDispatchThread()` to figure that out.

Making asynchronous calls to the EDT is your next goal.

7.2.4 Asynchronous calls with doLater

Posting new events to the EventQueue can be done by calling `SwingUtilities.invokeLater{}`. These events will be processed by the EDT the next time it gets a chance. That's why these calls are called *asynchronous;* code posted this way may take a few cycles to be serviced depending on the currently executing code in the EDT and the `EventQueue`'s state.

Parallel to what we described in the previous section, a call to `doLater{}` is like a call to `SwingUtilities.invokeLater{}`. But you can guess the difference: just as with `edt{}`, this method makes sure the current SwingBuilder instance is set as the closure's delegate. Again, you don't need to qualify SwingBuilder methods and properties; they'll be already in scope.

Because this threading facility always posts a new event to the `EventQueue`, there's nothing much else to see here. There are no calling-EDT violations to worry about as in the previous section.

One aspect of SwingBuilder threading remains for review: executing code outside of the EDT.

7.2.5 Outside calls with doOutside

You may be detecting a trend. For every Groovy threading option, there's a SwingBuilder-based alternative that adds a bit of spice: the ability to register the current SwingBuilder instance as the closure's delegate, saving you from typing a lot of repeated identifiers. The trend continues in this case, which means that calling `doOutside{}` has the same effect as `Thread.start{}` with the added benefit of a proper delegate set on the closure.

This threading facility mirrors `edt{}` in the sense that when `doOutside{}` is invoked, it spawns a new thread if and only if the current thread is the EDT; otherwise it calls the code in the currently executing thread. This behavior was introduced in Groovy 1.6.3; previous versions spawn a new thread regardless of the currently executing thread.

It appears this is all that SwingBuilder has to offer in terms of threading goodness—or is it? We'll revisit threading facilities in section 7.4; but before we get there, let's see how Griffon enables all these goodies. After all, that's what you're here to learn.

7.3 Multithreaded applications with Griffon

Griffon goes to great lengths to make your life more comfortable when dealing with Swing threading. It's as if the Swing threading problems don't exist in the first place! Well, not quite, but you get the point. Let's start with what happens when the application bootstraps itself.

7.3.1 Threading and the application life cycle

Back in chapter 2, we outlined the basic life cycle of each and every Griffon application. Here's a quick reminder.

The Griffon launcher sets up the appropriate classpath and loads the application's main class. Then it proceeds to read all of the application's configuration files located at `griffon-app/conf`. These operations are carried out in the main thread.

The initialization phase kicks in, giving you the choice to call custom code defined at `griffon-app/lifecycle/Initialize.groovy`. The contents of this script are guaranteed to be executed in the EDT. As a matter of fact, all scripts under that directory will be executed in the EDT—no exceptions.

Between calling the initialization life-cycle script and the next phase, something amazing happens—each MVC group configured to be initialized at startup comes to life! View members are initialized in the EDT too, which ensures that all view scripts are given proper threading handling. It's like each one was wrapped with an implicit call to `edt{}`.

Once initialization is finished, it's time for the next phase, Startup, which offloads any custom code to its corresponding life-cycle script. Then comes the Ready phase, which is guaranteed to be called after all pending events posted to the `EventQueue` have been processed. After clearing this phase, the application is fully initialized and configured, and it's time to display its main window (if any). This operation is also guaranteed to run in the EDT.

As you may recall from chapters 4 and 6, Griffon makes sure a SwingBuilder instance is available per view at all times. This facilitates the job of keeping references to UI components in the view. Given that the other two members of the MVC triad can read the view whenever they desire, all the dots are connected. Of course, this results in the ability to call `edt{}`, `doLater{}`, and `doOutside{}` from any view script, but as you saw in chapter 5, it's the responsibility of controllers to provide the required behavior. They're the ones that have to deal with threading explicitly more than any other MVC member.

You might be thinking that it would be simpler to create an instance of SwingBuilder during a controller's initialization or perhaps tap into the MVC group's associated builder by inspecting `app.builders`. But there's a better way: the Griffon way.

7.3.2 Threading support the Griffon way

We've stressed on previous occasions that one of Griffon's key components is the `CompositeBuilder`; this remarkable component is capable of creating a mish-mash of builders (albeit coherent and rather tasty). Every Griffon application sports a configuration file for its CompositeBuilder, located at `griffon-app/conf/Builder.groovy`. If that sounds familiar, it's because this was explained in detail back in chapter 2. The following listing reproduces the file's contents upon creating an application

Listing 7.9 Contents of griffon-app/conf/Builder.groovy

```
root {
    'groovy.swing.SwingBuilder' {
        controller = ['Threading']
        view = '*'
    }
}
```

Notice the `Threading` group being assigned to a `controller` property. That's the secret! SwingBuilder nodes can be found in groups: text component groups, button and action groups, windows and containers, and so on. Of course, there's a threading group too. When the `CompositeBuilder` is processing its configuration file, it pays attention to the group mechanism, assigning groups to views, controllers, or both.

To recapitulate what listing 7.9 describes, all nodes that are contributed by SwingBuilder are assigned to views; all nodes pertaining to SwingBuilder's `Threading` group are assigned to controllers as well. Finally, all nodes contributed by `ApplicationBuilder` are made available to views alone. With this configuration, you should be able to call `edt{}`, `doLater{}`, and `doOutside{}` from any controller.

Speaking of controllers, as you may recall from chapter 5, the main job of a controller action is to react to view events and signal views of new data availability. This means threading concerns are key when you're implementing actions in controllers. For this reason, Griffon applies special policies to controller actions, which we'll discuss next.

7.3.3 Controller actions and multithreading: a quick guide

We all know the end result when a long computation is run in the UI thread: unhappy users. We also know that to avoid this outcome, we must pay attention to proper threading etiquette. Griffon simplifies the task of keeping tabs on the thread in which a particular piece of code is being executed. Because controller actions hold a special place between views and models, it's also true that they receive additional care: Griffon assumes that a controller action should be executed outside of the UI thread unless otherwise specified.

We're back into the realm of convention over configuration. Most of the time, the actions that a developer wires up with the view are set up to execute a long-running computation or some logic that doesn't affect the UI elements directly, and binding is used for the rest. If that's the common case, then it makes no sense to force a developer

to wrap an action with an explicit call to `doOutside{}`—let the framework do it! This is a great feature that removes some of the pain of constantly thinking about threading concerns. But it comes with a price: when you do need an action to be executed in a different threading mode, you must configure it that way. A typical scenario for such a case is an action that performs navigation on UI components. But don't be afraid, the configuration is simple, as you'll see in a moment.

Remember the different configuration files available at your disposal in `griffon-app/conf`? One in particular affects the build system. Yes, it's `BuildConfig.groovy`. In this file, you can change the configuration read by the Griffon compiler, because as it turns out, it's the compiler's job to inject appropriate threading code into each action. This means you must instruct the compiler to skip certain actions when you don't want the default threading wrapper code to be applied to them.

The following snippet shows the typical setup for marking a set of actions to be skipped from this special threading injection-handling mechanism:

```
compiler {
    threading {
        sample {
            SampleController {
                readAction = false
            }
            NoThreadingController = false
        }
        com {
            acme = false
        }
    }
}
```

Here's a quick roundup of what's happening. No threading code is injected into the action named `readAction` on controllers of type `sample.SampleController`. If this controller happens to define additional actions, they see their code updated with threading wrapper injections. Only `readAction` isn't affected by the automatic wrapping code. In other words, this method disables injection with fine-grained control per action.

Next is `sample.NoThreadingController`. There's no action name related to this type, which instructs the compiler to skip injection code to all actions found in that controller. This is how you disable threading injection for a group of actions belonging to the same type.

Finally, all controllers found in the package `com.acme` or any of its subpackages have their actions skipped from injection. This is how you can disable a whole set of controllers that belong to the same hierarchy.

Given that these settings affect the compilation process, you must recompile your code right after making any adjustments to this configuration. Otherwise the changes won't be applied to the bytecode.

Let's pause for a moment and recap what you now know. All controller actions are automatically executed outside of the UI thread because the compiler injects wrapping

code into each one. The wrapping code is basically a call to `doOutside{}` as if you typed it explicitly. It's also possible to disable threading injection by configuring compile-time settings. Actions marked this way will be executed in the same thread as the caller, which may be the UI thread. Is it possible to explicitly mark an action to be executed in the UI thread? Yes, of course. You can do so with explicit calls to `edt{}` and `doLater{}` as you saw before. But there's another option in Griffon's bag of tricks—one that can be applied to any Groovy class, not just controllers.

7.3.4 Fine-tuning threading injection

Threading concerns are most important to controllers, but other components may require special handling of threads too. It's likely that those components (such as services or plain beans) don't participate in the threading injections done by the builder, like controllers. But you can instruct the compiler to do the legwork for you, if you give it a few hints in the correct places.

Remember that `@Bindable` is a handy way to generate observable properties on a bean. It works by hooking into the AST transformation framework. The feature we'll show you next also hooks into the compiler through the same mechanism. The following snippet demonstrates it in all its glory:

```
import griffon.transform.Threading
class SomeBean {
    @Threading(Threading.Policy.OUTSIDE_UITHREAD)
    void fetchData(URL endpoint) { ... }
}
```

The `@Threading` annotation is responsible for advising the compiler about the threading hints we spoke of. You can apply this annotation either to methods or closure properties; its value is the type of injection you want to see applied to the code. This type has four possible settings, as explained in table 7.1.

Table 7.1 Values for all types of injections that can be performed with `@Threading`

Setting	Meaning
`Policy.OUTSIDE_UITHREAD`	The code will be executed outside of the UI thread as if it were wrapped with `doOutside`. This is the default value for the annotation.
`Policy.INSIDE_UITHREAD_SYNC`	The code will be executed synchronously in the UI thread, as if it were wrapped with `edt`.
`Policy.INSIDE_UITHREAD_ASYNC`	The code will be executed asynchronously in the UI thread, as if it were wrapped with `doLater`.
`Policy.SKIP`	No injection will be performed. The code will be executed in the same thread as the caller.

Phew! That's a lot to digest, and we still have a few more things to discuss. Don't worry—the next sections build on what you already know.

7.3.5 What about binding?

If controller actions are automatically executed outside of the EDT by default, then the following snippet might be doing the wrong thing:

```
class SampleController {
    def model
    def someAction {
        String input = model.input
        model.output = input * 2
    }
}
```

This particular controller has an action that reads a value from the model and sends it back transformed. If the model properties are bound to UI components, this means the value written back to the model will be sent in the same thread executing the action, which you know is not the EDT. And that's a problem.

You could fix the code by wrapping the write back with a call to `doLater{}`. Wouldn't it be cool if Griffon knew about this case and automatically wrapped the call for you? Well, it turns out it can. The code you previously thought was unsafe *is* safe. We weren't kidding about the great lengths Griffon goes to in order to make your life easier. This behavior is also configurable, should you encounter the need to change the default. The `bind()` node accepts the values listed in table 7.2 for a property named `update`.

Table 7.2 Update strategies applicable to bindings, and their meanings

Value	Meaning
`mixed`	Proceeds if the current thread is the UI thread. Otherwise, delivers the update using `doLater{}`. This is the default for bound UI components.
`async`	Posts the update with `doLater{}`.
`sync`	Proceeds if the current thread is the UI thread. Otherwise, calls `edt{}`.
`outside`	Proceeds if the current thread isn't the UI thread. Otherwise, submits the update to an `ExecutorService`.
`same`	Proceeds in the current thread. This is the default for non-UI components.
`defer`	Submits the update to an `ExecutorService`.

For example, if for some reason you want to force an update to always happen outside of the EDT, you can use the `defer` setting when creating the binding, like this:

```
textField(text: bind(source: model,
    sourceProperty: 'output',
    update: 'defer'))
```

A demonstration of the threading facilities on controllers is in order. But before we dive into it, we want to discuss one additional threading option.

7.4 SwingXBuilder and threading support

As you may be aware, the JDK provides many UI components, but it's in no way complete. There are myriad components out there, available for free or by paying a fee. At some point, Sun Microsystems decided it was a good idea to create a repository that could serve as an incubator for new ideas and components that, given the proper time and maturity, could find their way into the JDK. And thus the SwingX project was born.

Sun engineers spotted early on that dealing with threads in Swing applications was a hard task, and they decided to alleviate the problem by creating a powerful threading-aware component: the `SwingWorker`. This component was fostered by the SwingX project and eventually found its way into the JDK when Java 6 was released in 2006. The powers that be saw that the enhancements brought by SwingWorker were too good to be left alone and decided to port it back to Java 5, but the code never made it to the JDK 5 proper; it stayed at the SwingX project as a separate download.

All this talk leads to the following: if your target platform is JDK 5, you'll need a SwingX jar to harness `SwingWorker`'s power. On the other hand, if your target is JDK 6 or beyond, there's no additional library to set up. You can bet Griffon has something up its sleeve, because it's been good at facilitating things for you so far.

In this case, plain SwingBuilder comes to your rescue. You see, SwingBuilder isn't the only builder capable of dealing with Swing components; a few others have ties to SwingBuilder and the Griffon project, as we'll discuss in chapter 12. Right now, we'll introduce SwingXBuilder, which is an official extension to SwingBuilder carried out as one of Google's Summer of Code 2007 entries.

SwingXBuilder is capable of dealing with the complexity of setting up the proper `SwingWorker` class depending on your development and deployment targets. Excited? Let's configure SwingXBuilder and explore its threading offerings.

7.4.1 Installing SwingXBuilder

Installing SwingXBuilder is as easy as fetching its latest release from http://griffon.codehaus.org/SwingXBuilder, along with its dependencies from the SwingX project. But there's a quicker and painless option: you can install the swingx-builder plugin. We might be getting a few steps ahead of ourselves, because plugins will be discussed in full detail in chapter 11; nevertheless, here's how you would install this plugin. Open a command prompt or shell, and type in the following, making sure the command is executed in your application's main directory:

```
$ griffon install-plugin swingx-builder
```

If you have access to the network, the command downloads the latest version of the swingx-builder plugin and installs it on your application. That wasn't so bad, was it? The installation procedure does more than just set up the proper classpath configuration by placing all required jars in your application's lib directory; it also modifies `CompositeBuilder`'s configuration script. Take a peek at `griffon-app/conf/Builder.groovy` again, and you should see something like the following listing.

Listing 7.10 Updated configuration found on Builder.groovy

```
root {
    'groovy.swing.SwingBuilder' {
        controller = ['Threading']
        view = '*'
    }
}
jx {
    'groovy.swing.SwingXBuilder' {
        controller = ['withWorker']
        view = '*'
    }
}
```

There's a new node, `jx`, which contains the definitions related to SwingXBuilder. This node exposes a `Threading` group too, and in that group you'll find what you're looking for: a `withworker()` node. What makes this node so special? Keep reading to find out.

7.4.2 The withWorker() node

As it turns out, `withWorker()` exposes the same public contract regardless of what version of `SwingWorker` you're currently dealing with (JDK 5 or JDK 6). This ensures that no matter what platform your application is deployed to, it will behave the same.

Let's get down to business. This node relies on the builder pattern to configure the underlying `SwingWorker` instance. There are four configurable nested nodes at your disposal: `onInit()`, `work()`, `onUpdate()`, and `onDone()`. Of these, only two are required to be defined by your code: `work()` and `onDone()`. This is how a typical usage looks:

```
jxwithWorker(start: true) {
   work {
     // work, work, and more work
   }
   onDone {
     // we're done!
   }
}
```

Let's break down each pseudo-node, starting with the one responsible for initializing the worker.

INITIALIZING THE WORKER

Because you're being shielded from the actual `SwingWorker` implementation chosen at runtime, there's no way for you to provide a constructor for it. But you can run initialization code before the worker gets on with its job; this is the responsibility of `onInit()` pseudo-node. This node requires a closure that takes no parameters, so be aware that the code defined in the closure will be executed in the current thread.

MAKING THE WORKER DO SOMETHING

When the worker is ready to be executed, either by defining a true value for its start property or by calling its `start()` method directly, it will call the code defined by the

`work()` pseudo-node. But as opposed to `onInit()`, the code defined for `work()` is guaranteed to run outside the EDT. The `onDone()` pseudo-node will be called once this one has finished doing whatever you instructed it to do.

Occasionally, you'll want to refresh the UI contents with the partial results, and you need a way to get back into the EDT. Before you glance at the previous threading facilities, you should know that the `SwingWorker`'s designers took this scenario into account; thus there is a handy `publish()` method available. It signals the worker to process whatever parameters you send to the `publish()` method in the EDT; that's the job of the third pseudo-node.

One more thing: the last evaluated expression on this pseudo node will serve as the computed value for the whole operation. You can access that value by calling `get()`, typically in `onDone()`.

Updating the UI as you go

If `publish()` is responsible for sending data back into the EDT to be processed, it's the job of `onUpdate()` to do the actual processing. This node's closure takes a single parameter of type `List`; because `publish()` is a method that takes variable arguments, all of them are collected in a `List` before being sent to `onUpdate()`.

Finishing the job

It's time up to tie any loose ends and process the results when the worker has finished its business. This is where the `onDone()` pseudo-node steps up to handle the situation. The closure you set to this node is executed back in the EDT. To obtain the value of the computation made by `work()`, you call `get()`.

Enough theory. Let's see all this in action!

7.5 Putting it all together

For demonstration purposes, and to draw a parallel to the discussion made throughout this chapter, you'll develop yet another FileViewer application. The difference is that you'll enable four different loading techniques using each of the threading options at your disposal. You'll be constantly updating a UI element on screen, too; that way you'll know right away if the application becomes unresponsive due to a threading problem. Figure 7.1 shows the application.

Each test executes a different loading technique. You'll develop your threading-aware FileViewer application in the following steps:

1. Outline the application.
2. Set up the UI.
3. Define the tabs.
4. Code the four file-loading techniques.

Time to flex those thinking and writing muscles: you have an application to build.

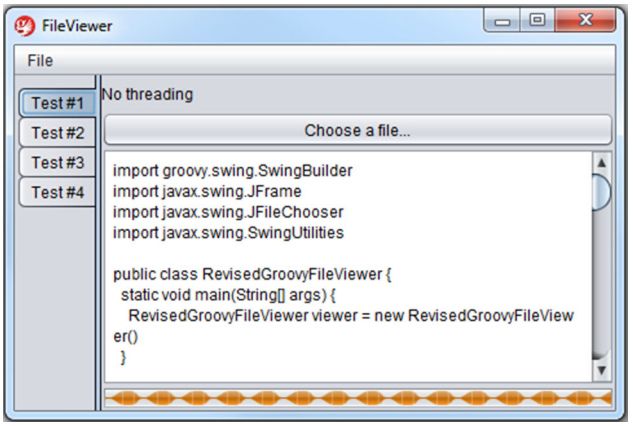


Figure 7.1 A threading-aware FileViewer application displaying a file using the first technique (Test #1) for reading a file's contents. The other file-loading techniques (Test #2, #3, and Test #4) depend on the threading options employed.

7.5.1 Defining the application's outline

Let's ponder for a moment what you want to accomplish with the FileViewer application. The following actions are common to all file-loading techniques, regardless of threading approach:

- Display a button that lets you choose the file to load.
- Display a `textArea` with the file's contents.
- Display a `progressBar` that is updated constantly.
- Provide each file-loading technique with its own tab and title.

You could pack each loading technique in its own MVC group, but that might lead to repeated code. Another option would be creating a second MVC group, similar to what you did with GroovyEdit back in chapter 1, but you would need a way to explicitly instantiate each copy and wire it up to the `tabbedPane` (here's a hint: you could do it with one of the life-cycle scripts). Or you could keep it simple with a single MVC group, and have a looping construct instantiating and wiring up each tab, all in the view script. Let's go with the third option, because it's the simplest.

First, let's create the application. Drop to your command prompt, and type the following:

```
$ griffon create-app FileViewer
```

Now change to the directory of the newly created application. To add the dependencies you'll need for this application by installing the SwingXBuilder plugin, drop into your command prompt again and type

```
griffon install-plugin swingx-builder
```

Make sure SwingXBuilder's `Threading` group is contributed to controller classes. Your `griffon-app/conf/Builder.groovy` should contain something like this:

```
jx {
    'groovy.swing.SwingXBuilder' {
```

```
        controller = ['withWorker']
        view = '*'
    }
}
```

You're done setting up the application—the only thing left is adding the fun bits to it.

7.5.2 Setting up the UI elements

The next step for evaluating each loading technique with threading on this application is building the common UI elements. Open `FileViewerView.groovy` with your favorite text editor or IDE, and replace its contents with the code shown in the following listing.

Listing 7.11 Minimal implementation for `FileViewerView.groovy` without tabs

```
import javax.swing.JTabbedPane

actions {
   action(id: "quitAction",
      name: "Quit",
      mnemonic: "Q",
      accelerator: shortcut("Q"),
      closure: controller.quit)
}

fileChooser = fileChooser()
mainWindow = application(title:'FileViewer', size: [480,320],
     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 {
      menu("File") {
         menuItem(quitAction)
      }
   }
   borderLayout()
   tabbedPane(constraints: CENTER, tabPlacement: JTabbedPane.LEFT)
}
```

There's one more thing to do before you try the application for the first time. `quitAction` refers to an action named `quit` found in the controller, so let's define that action in FileViewerController.groovy as shown here:

```
class FileViewerController {
   def model
   def view

   def quit = { evt = null ->
      app.shutdown()
   }
}
```

At this point the application should be functional, although it only displays a menu bar at this moment. You'll fix that shortly by adding a tab for each of the loading techniques you'd like to explore.

7.5.3 Defining a tab per loading technique

You're ready to create the tabs, and you'll handle this operation as a four-step procedure:

1. Define the common code for each tab.
2. Graft each tab into the `tabbedPane`.
3. Make sure the model is updated with the proper properties.
4. Add skeleton implementations for each of the required controller actions.

Return to the view script, and paste the code shown in listing 7.12 before anything else. You can place this code anywhere in the script before the `tabbedPane` is defined, but putting it at the top makes the script look cleaner.

Listing 7.12 Parameterized tab creation

```
def makeTab = { heading, loadTechnique, technique ->
  panel {
    borderLayout()
    panel(constraints: NORTH) {
    gridLayout(cols: 1, rows: 2)
      label(heading)
      button("Choose a file...",
        enabled: bind{ technique.enabled },
        actionPerformed: loadTechnique)
    }
    scrollPane(constraints: CENTER) {
      textArea(id: "editor",
        editable: false,
        lineWrap: true,
        text: bind{ technique.text },
        caretPosition: bind(source: technique,
              sourceProperty: "text", converter: {0}))
    }
    technique.progress = progressBar(indeterminate: true,
      minimum:0, maximum: 100,
      constraints: SOUTH)
  }
}
```

❶ Label with heading text

❷ Reference loading technique

❸ Text source

Let's take a moment to review what's going on with this piece of code. Each tab requires the following:

- Some heading text ❶, such as "No threading" as shown in figure 7.1.
- A reference to a `loadingTechnique` ❷. This is a direct link to the appropriate controller action.
- An additional parameter for the closure that defines a tab: `technique` ❸.

Each loading technique is supposed to be independent from the rest. You don't want them to step on each other's toes, which is why there will be a separate space on the model. You'll see how to do that when you get to the third step of this procedure.

First, though, you must wire up each tab into the `tabbedPane`. Scroll down in your editor, locate the `tabbedPane` definition, and replace it with the code shown in the next listing.

Listing 7.13 Adding parameterized tabs to the `tabbedPane`

```
tabbedPane(constraints: CENTER, tabPlacement: JTabbedPane.LEFT) {
 [[heading: "No threading", action: "readFileNoThreading"],
 [heading: "Threading - doOutside/doLater", action:
"readFileWithThreading"],
 [heading: "Threading - withWorker", action: "readFileWithWorker"],
 [heading: "Threading - progress update", action: "readFileWithUpdates"]]
     .eachWithIndex { entry, index ->
      index += 1
      def loadTechnique = controller."${entry.action}"
      def technique = model."technique$index"
      widget(title: "Test #"+index,makeTab(entry.heading, loadTechnique,
technique))
   }
}
```

You have a list of maps. A looping construct iterates over each entry, and the real values for `loadTechnique` and `technique` are calculated as you expect them to be.

The variable `loadTechnique` is indeed a reference to a controller action. Judging by the names on the maps, the first action will be resolved to `controller.readFileNoThreading`. Here you're taking advantage of Groovy's ability to let a property access call be resolved in a dynamic way, because you're using a parameterized `String`.

The same trick is applied to the `model` property. Judging by that usage, there should be four observable beans on the model whose names are `technique1`, `technique2`, `technique3`, and `technique4`. That's precisely what you're about to implement.

Open FileViewerModel.groovy in your editor, and write the following:

```
class FileViewerModel {
  Map technique1 = [text: "", enabled: true] as ObservableMap
  Map technique2 = [text: "", enabled: true] as ObservableMap
  Map technique3 = [text: "", enabled: true] as ObservableMap
  Map technique4 = [text: "", enabled: true] as ObservableMap
}
```

You're using a quick prototyping technique here. Do you remember `ObservableMap` and `ObservableList` from chapter 3? Usually you would create a simple class as a holder for a few properties like the ones you need, and those properties need to be made observable. Thanks to `@Bindable`, the code would've been shorter; but you went with `ObservableMap` instead, which works like any other observable bean with the added benefit that properties can be added at runtime (a fact you rely on). Look back to listing 7.12, where the `progressBar` is defined. A new property with name `progress`

is added to the technique, and this means every `ObservableMap` in your model will hold three properties once the application has processed the view script. If you're wondering why you need to keep a reference to the `progressBar`, the answer lies in the final implementation of the controller's actions. Speaking of which, let's finish up this procedure with the fourth step.

The following code adds a skeleton implementation to each action found on the controller:

```
class FileViewerController {
   def model
   def view

   def quit = { evt = null ->
      app.shutdown()
   }

   def readFileNoThreading = { evt = null -> }
   def readFileWithThreading = { evt = null -> }
   def readFileWithWorker = { evt = null -> }
   def readFileWithUpdates = { evt = null -> }
}
```

There you have it; this should be enough to launch the application and inspect its visuals. Feel free to make any adjustments you like. You're at the final phase of making a multithreaded application: wiring up the different loading techniques using the various threading facilities provided by Griffon.

7.5.4 Adding the loading techniques

Here's the attack plan for the four loading techniques you'll implement:

- Technique 1, `readFileNoThreading`, will read the file and update the `textArea` in the current executing thread—in other words, you won't use a threading facility. This will demonstrate what will happen if threading isn't taken into account. Considering that each controller action is executed outside the UI thread by default, you must explicitly mark this one with `@Threading.Policy.SKIP`.
- Technique 2, `readFileWithThreading`, will use `doOutside{}` to read the file's contents in a different thread other than the EDT. Then it's back into the EDT with `doLater{}` to update the `textArea`.
- Technique 3, `readFileWithWorker`, is similar to the second, but it will perform its job by using `SwingWorker`.
- Technique 4, `readFileWithUpdates`, is a refined version of the previous one. It will use `SwingWorker` to read the file's contents, and it will also publish timely updates as it reads the file. This is why you needed a reference to the `progressBar`; you'll use it as a status display.

Another tidbit before you get into the code, given that you'll read the file's contents on a different thread than the EDT, is that the button that pops up the `fileChooser` can be clicked several times before the first request has been processed fully. You need

to find a way to avoid this kind of situation. Fortunately, the button's enable state is bound to a property on the model; you just have to make sure the value for that property is toggled at the appropriate time.

Without further ado, let's open FileViewerController.groovy and add the code for the four loading techniques.

READ FILE, NO THREADING

To replace the first threading action, paste in the code shown in the following listing.

Listing 7.14 Full implementation of the `readFileNoThreading` action

```
private doWithSelectedFile = { Map technique, Closure codeBlock ->
  def openResult = view.fileChooser.showOpenDialog(view.mainWindow)
  if( JFileChooser.APPROVE_OPTION == openResult ) {
    File file = new File(view.fileChooser.selectedFile.toString())
    technique.text = ""
    technique.enabled = false
    codeBlock(file)
  }
}

@Threading(Threading.policy.SKIP)
def readFileNoThreading = { evt = null ->
  def technique = model.technique1
  doWithSelectedFile(technique) { file ->
    technique.text = file.text
    technique.enabled = true
  }
}
```

Remember to include an import for `javax.swing.JFileChooser` and `griffon.transform.Threading` at the top of the file, or you'll get compilation errors. That's the minimal implementation for this action. The code isn't protected against any I/O errors that might occur, such as no read permissions or something similar; error handling is left out to keep the code simple and on topic.

You take the precaution of factoring out the common code that each technique requires for selecting a target file and toggling the model's enable property into a private method—recall from chapter 5 that public methods will be seen as actions too. This method takes two arguments; the first is the composed model space the technique requires (defined as an `ObservableMap`), and the second is a closure that defines the code that puts the technique to work.

READ FILE WITH THREADING

Defining the second technique is pretty straight forward, as shown in the next listing.

Listing 7.15 Full implementation of the `readFileWithThreading` action

```
def readFileWithThreading = { evt = null ->
  def technique = model.technique2
  doWithSelectedFile(technique) { file ->
    doOutside {
```

```
      technique.text = file.text
      technique.enabled = true
    }
  }
}
```

The second technique builds on the first. It makes sure that reading the file's contents takes place on a thread that isn't the EDT, and then it proceeds to update the UI after the text has been read. Of course, it does so back in the EDT because bindings will update UI components in the EDT by default.

READ FILE WITH WORKER

The following listing shows the code for the third technique.

Listing 7.16 Full implementation of the `readFileWithWorker` action

```
def readFileWithWorker = { evt = null ->
  def technique = model.technique3
  doWithSelectedFile(technique) { file ->
    jxwithWorker(start: true) {
      work { file.text }
      onDone {
        technique.text = get()
        technique.enabled = true
      }
    }
  }
}
```

Notice how similar listings 7.15 and 7.16 are. The most relevant change is how to obtain the computed value once the calculation is finished, which is unique to the `withWorker()` node.

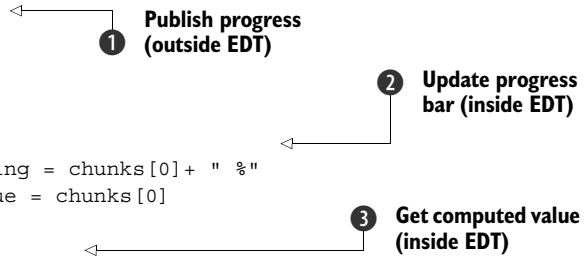
READ FILE WITH UPDATES

There's one final technique to try. As we mentioned before, you use `SwingWorker` again, and this time the `progressBar` will serve as an indicator of how much progress you have made when reading the file (see the following listing).

Listing 7.17 Full implementation of the `readFileWithUpdates` action

```
def readFileWithUpdates = { evt = null ->
  def technique = model.technique4
  doWithSelectedFile(technique) { file ->
    jxwithWorker(start: true) {
      onInit {
        technique.progress.with {
          setIndeterminate(false)
          setStringPainted(true)
          setString("0 %")
        }
      }
      work {
        int max = file.size()
```

```
        def buffer = new char[max/10]
        def text = new StringBuffer()
        file.withReader { reader ->
          (1..10).each { i ->
            reader.read(buffer, 0, buffer.size())
            text.append(buffer)
            Thread.sleep(300)
            publish(i*10)
          }
        }
        text.toString()
      }
      onUpdate { chunks ->
        technique.progress.string = chunks[0]+ " %"
        technique.progress.value = chunks[0]
      }
      onDone {
        technique.text = get()
        technique.progress.stringPainted = false
        technique.progress.indeterminate = true
        technique.enabled = true
      }
    }
  }
}
```



Here is what's happening. As you may recall from listing 7.12, the `progressBar` is set to indeterminate mode. You need to change it to deterministic mode, and you accomplish that during the worker's initialization. Notice you're reusing the `with{}` trick to change the closure's delegate. All three methods will be executed on the `progressBar` associated with this technique.

The worker reads the file in 10 chunks, one at a time, pausing briefly to simulate slow I/O, and finally publishing the current chunk's index ❶. This index will serve to calculate the amount of progress the worker has made so far. Remember that all this happens outside the EDT.

Back in the EDT, the worker updates the UI state safely ❷, mainly by changing the current value of the `progressBar` to a percentage of the total work so far. It updates both the text and the value.

When the worker has finished its job ❸, you're again in the EDT, which is perfect because the only things left to do are update the `textArea`'s contents and revert the `progressBar` back to its previous state.

7.5.5 FileViewer: the aftermath

It took you a few iterations, but you've finished the job you set out to do: demonstrate how the different threading facilities provided by SwingBuilder and Griffon can be used in a typical scenario. We tried to keep the code's verbosity at a minimum. Just remember that the application isn't completely safe from I/O errors occurring at unexpected times; but that job shouldn't be difficult, especially if Groovy techniques

such as closure currying and metaprogramming are added to the mix. We hope you enjoyed making this little application, and as a reward, here are the app's stats:

```
+----------------------+-------+-------+
| Name                 | Files |  LOC  |
+----------------------+-------+-------+
| Models               |     1 |     7 |
| Views                |     1 |    55 |
| Controllers          |     1 |    85 |
| Lifecycle            |     5 |     3 |
| Integration Tests    |     1 |    14 |
+----------------------+-------+-------+
| Totals               |     9 |   164 |
+----------------------+-------+-------+
```

Can you believe you accomplished so much with so little? Only 147 of those 164 lines of code were explicitly written for this application.

7.6 Additional threading options

We've discussed all threading options related to Java Swing that Griffon has to offer. But Swing isn't the only toolkit that can be used in the JVM.

There's the Standard Widget Toolkit (SWT), for example. If you're familiar with the Eclipse IDE, then you've seen SWT in action. Its driving goal is to provide a seamless integration with the native environment, allowing the running platform to draw the widgets by its own means; in Swing, on the other hand, an abstraction layer (implemented with Java 2D by the release of JDK 6) draws all the bits.

You've probably heard about JavaFX. It too is a UI toolkit that provides a modern set of features. Qt from Nokia (originally from Trolltech), Apache Pivot from the Apache Foundation, and GTK (on Linux) are other popular choices.

Regardless of which one you pick to develop your next application, you'll soon face the problem of multithreading. Each toolkit deals with the problem in its own way, often differently from the others. What if there was a common way to handle multithreading regardless of the toolkit? Better yet, what if it was close to what you already know with Swing?

This is precisely what the following Griffon threading facilities do for you. Let's start by finding out how you can call code in the UI thread. When dealing with Swing, the UI thread is the EDT, but other toolkits call it something different.

7.6.1 Synchronous calls in the UI thread

In section 7.2.3, we explained the concept of the `edt{}` block. Its toolkit-agnostic counterpart is `execInsideUISync{}`. This block guarantees that code passed to it is executed synchronously in the UI thread. You can be sure this block performs exactly the same motions as `edt{}` when running in Swing.

It's safe to change all calls from `edt{}` to `execInsideUISync{}` in all parts of a Griffon application.

7.6.2 Asynchronous calls in the UI thread

Following the steps of the previous block, the next one mimics closely what you can do with `doLater{}`. Executing code in the UI thread in an asynchronous way can be achieved by means of the `execInsideUIAsync{}` block.

Similar to what Swing's EDT does, all other toolkits have a way to post an event to be processed in the UI thread at a later point in time. This threading block exploits that option for each of the supported UI toolkits.

Next in the list is executing code outside of the UI thread, but we're pretty sure you've guessed the name already.

7.6.3 Executing code outside of the UI thread

The name, as you probably guessed, is `execOutsideUI{}`, inspired by the already-discussed `doOutside{}`. This block also ensures that all code passed to it is executed in a thread that isn't the UI thread.

On more recent versions of Griffon (since 0.9), you'll discover a few hints here and there about the usage of these threading facilities. Their usage is preferred over the Swing facilities, not just because they're newer but because they make your code less fragile to a sudden UI toolkit change if it's required by your application. They also make the code look more consistent across different toolkits. That way you can read code of an SWT application and understand it more easily, even if the only toolkit you know is Swing.

But you can use a few additional methods besides the three we just discussed. They complete the set of threading options that Griffon has to offer.

7.6.4 Is this the UI thread?

In Swing, how do you find if the current thread is the EDT? That's right: you ask the JVM using `SwingUtilities.isEventDispatchThread()`. But as we said before, other UI toolkits have decided to use their own conventions that may or may not match Swing's, so you can be sure `SwingUtilites` won't work for all cases. You need a way to query whether a thread is the UI thread, and that's the job of the `isUIThread()` method.

This method, as well as the other threading methods introduced in section 7.6, are available to all MVC artifacts and the application instance; you can call them pretty much from anywhere.

7.6.5 Executing code asynchronously

Wait, didn't we talk about this one before? Yes, we did, but in the context of executing code asynchronously in the UI thread. This case is for executing code asynchronously outside of the UI thread, as a promise for a future execution. And with that, we gave away the name of the method: `execFuture()`. It may take either a `Closure` or a `Callable` as argument and returns a `Future` instance. You're free to do with the `Future` object as you please.

By default, this method is invoked using an `ExecutorService` that should be configured to the number of cores available in the running platform. A variant of this

method accepts an `ExecutorService` instance. In this way, you can configure all the details of the execution.

7.7 Summary

This chapter has shown that you don't need to be driven to the verge of despair while making sure your Swing applications take concurrency and threading into account. Thanks to the power of Groovy SwingBuilder and Griffon's additional threading facilities, this process doesn't need to be a crazy ride.

You learned that Groovy provides an advantage for threading by allowing any closure to be used as the body of a `Runnable`. You have two threading options: create a closure and set it either as the parameter for a new `Thread` or as the parameter for the helpful `SwingUtilities.invokeAndWait()` and `SwingUtilities.invokeLater()`. As a third option, Groovy's SwingBuilder makes the job of properly handling threading in a Swing application much easier. A fourth threading option is based on JDK 6's `SwingWorker` or SwingX's `SwingWorker`, a handy class that facilitates the job of offloading work from the EDT.

We also explored how Griffon makes sure the application life cycle is executed in the correct threads depending on the particular phase. And you worked through a full example of all of Griffon's threading techniques. We're confident it will inspire you to thread into concurrency with more confidence (pun intended).

Finally, we surveyed additional, UI-toolkit-agnostic threading options that Griffon exposes.

Now that you know how to deal with threading concerns, we're ready to address the subject of notifications and events. Given that many of them will occur at any point of the execution of your application, you'll have to either deal with them immediately or delay their handling.

8 Listening to notifications

This chapter covers

- Setting up event handlers on your build
- Setting up application event handlers
- Adding listeners to model updates
- Transforming any class into an event publisher

Think for a moment how your typical work day starts. You get up on time (most likely) because your alarm clock rings. You grab some warm toast for breakfast because you hear it popped from the toaster. On your way to work, you know to stop on the street because the traffic light changes to red.

What is common to all these cases is that you take action as a response to some stimulus, whether auditory (your alarm clock ringing) or visual (the traffic light). It's as if you're reacting to signals sent by different agents. Could you imagine your morning routine without all those signals? It could turn out to be a bit chaotic. It might also be tedious, because you'd constantly have to check for a particular condition to see if you could proceed. Sometimes it's better to react to a signal rather than poll for it.

The same principle applies to applications. You know that during a build, a predefined set of steps must take place. After all, the build must be reproducible every time. But a build should also be extensible: not every shoe fits every foot. Sometimes

you'll need to tweak the build. Signals, or *events* as we'll call them from now on, are a perfect fit to make this happen.

In this chapter you'll see how to use build events to change and/or extend the build process. Next, you'll take the information you learned about using events during build time and see how the same idea applies to runtime using application events. Finally, you'll learn how easy it is to add event publishing to any class using the `@EventPublisher` annotation.

Let's start by looking at how events can be used to change and/or extend the build process.

8.1 Working with build events

We're sure you've become accustomed to Griffon's command-line tools by now. As you may recall, the scripts rely on Gant (http://gant.codehaus.org), a Groovier version of Ant (http://ant.apache.org), to work. Each script has a predefined set of responsibilities. Some scripts even piggy-back on others to get the job done. What you may not know is that you can interact with the build process, and even change it, by means of build events.

In this section, you'll see that the build process uses build events to inform listeners which step the build process is in. Using this information, listeners can take specific actions, such as post-processing resource files after the compile step has finished.

To interact with events, an event handler must be registered. You use a special script to register handlers, and the build system locates the script by means of the naming convention. Before we get into the details of using this script, let's see how to create scripts with ease.

8.1.1 Creating a simple script

Scripts are just one of many of the Griffon artifacts available to you. You might have noticed the scripts directory located under your application's root directory; it's empty by default.

Scripts are simple to create, but just to be on the safe side this first time, let's rely on the command line. Type the following commands at your prompt:

```
$ griffon create-app buildEvents
$ cd buildEvents
$ griffon create-script First
```

This creates a file named `First.groovy` inside the `scripts` directory. If you omit the script name, the command will prompt you for one.

Let's peek into the file you just created:

```
target(name: 'first',
        description: "The description of the script goes here!",
        prehook: null, posthook: null) {
    // TODO: Implement script here
}

setDefaultTarget('first')
```

There isn't much to see, other than the definition of the script's default target.

Working with Gant

If you're familiar with Ant you're more than ready to start hacking build scripts. If you're not, don't worry; here are a few hints to get you up to speed.

Direct your browser to http://ant.apache.org/manual, and read a few pages, especially those that show how to set up a sample build file. You may notice an example with a snippet similar to this:

```
<target name="init">
  <mkdir dir="build"/>
</target>
```

Armed with this knowledge, edit your First script to make it look like this:

```
target(name: 'first',
        description: "The description of the script goes here!",
        prehook: null, posthook: null) {
    ant.mkdir(dir: "build")
}
setDefaultTarget(first)
```

Do you see the trick? Basically, you have to transform XML code into Groovy code. You should be able to call any Ant target from within your scripts; just remember to qualify it with the `ant.` prefix.

Gant scripts are not only a groovier version of Ant's XML-based scripts, they're also valid Groovy scripts. We know what you're thinking right now, and you're correct: you can use any of Groovy's features and mix them with Ant targets. Yes, you can mix closures, lists, maps, iterators, and pretty much everything you've learned so far pertaining to Groovy. Isn't that great? Not only do you get rid of XML's visual clutter, but you also gain a compile time checked script with powerful programming features.

It's time to continue with build events and their handlers now that you've got the basics of scripts.

8.1.2 Handling an event with the events script

To interact with events, you must register an event handler using the specially named script `_Events.groovy`. You can create it by hand or use the `create-script` command.

Given that this is a special script, let's start with a blank file. Fire up your favorite editor, create a text file, paste the following code into it, and save it under scripts as `_Events.groovy`.

Listing 8.1 Barebones `_Events.groovy` script listening to a single event

```
eventCompileStart = {
    println "Griffon is compiling sources"
    ant.echo message: "This message written by an Ant target"
}
```

Go back to your command prompt, and compile the `buildEvents` application. You should see the following output:

```
Welcome to Griffon 0.9.5 - http://griffon.codehaus.org/
Licensed under Apache Standard License 2.0
Griffon home is set to: /usr/local/griffon

Base Directory: /tmp/buildEvents
Running script /usr/local/griffon/scripts/Compile.groovy
Resolving dependencies...
Dependencies resolved in 702ms.
Environment set to development
Resolving plugin dependencies ...
Plugin dependencies resolved in 763 ms.
Griffon is compiling sources
     [echo] This message written by an Ant target
...
```

Interesting, isn't it? Toward the bottom, Griffon reassures you that it found your script file as you get the printouts specified in the script itself. This is great; in the event (no pun intended) you didn't set the correct event handler (more on that in just a moment), you still know your event script was located. Next, Griffon compiles sources, which verifies that a regular Groovy statement can be used on the script. Finally, you use one of Ant's basic targets, the `echo` target.

This is all fine and dandy; the script is working correctly because you followed the convention for naming and writing an event handler. Perhaps it's time to explain that convention, don't you think? Look again at your `_Events.groovy` script, and maybe you can spot it.

Event handlers are actually closure properties set on the script's binding. They're of the form

```
event<EventName> = { args -> /* your code*/}
```

You can infer from `_Events.groovy` that the Griffon compile script fires a `CompileStart` event. Table 8.1 summarizes the most common events and the targets that fire them.

Table 8.1 Events you'll often see in a Griffon script

Target	Event	Fired when?
`clean`	`CleanStart`	Before cleaning the application's artifacts
`clean`	`CleanEnd`	After cleaning has been completed
`compile`	`CompileStart`	When compilation starts
`compile`	`CompileEnd`	When compilation ends
`packageApp`	`PackagingStart`	Just before collecting jar files
`packageApp`	`PackagingEnd`	Just after collecting jar files
`runApp`	`RunAppStart`	Just before launching the application

Table 8.1 Events you'll often see in a Griffon script *(continued)*

Target	Event	Fired when?
runApp	RunAppEnd	When the application has shut down
runApplet	RunAppletStart	Just before launching the application in applet mode
runApplet	RunAppletEnd	When the applet has shut down
runWebstart	RunWebstartStart	Just before launching the application in webstart mode
runWebstart	RunWebstartEnd	When the application has shut down
several	StatusFinal	Just before a script finishes or an error happens

There are of course more events than these. You'll find a comprehensive list in the Griffon Guide (http://griffon.codehaus.org/guide/latest/), or you can glance at the scripts source code (`$GRIFFON_HOME/scripts`) if you're feeling adventurous.

You now know how to handle an event, but what about publishing one? That's the topic we'll cover in the next section.

8.1.3 Publishing build events

If you think writing an event handler was easy, just wait until you see how you publish an event! Every script you write will have the ability to publish events as long as you include the `Init` script. Luckily, that's what `create-script` generates. Let's write a simple example using the Ping concept.

Create a new script named `Ping.groovy`. Use `create-script` to make it easier:

```
$ griffon create-script Ping
```

Open it in your editor, and make sure it looks like this:

```
target(name: 'ping',
        description: "The description of the script goes here!",
        prehook: null, posthook: null) {
    event("Ping", ["Howdy!"])
}
setDefaultTarget('ping')
```

To add an event handler for the Ping event, open `_Events.groovy` (or create it in the scripts directory if you don't have it already). Remember the conventions? Do this:

```
eventPing = { msg ->
    println "Got '${msg}' from ping"
}
```

The only thing left to do is test it. Your Ping script is like any other Griffon script, which means you can launch it using the `griffon` command. Scripts are not only useful for handling events, but also work as an extension mechanism, at least at build time.

> **TIP** We'll explain more about Griffon's extension mechanism, both at build time and runtime, in chapter 11.

To launch the `Ping.groovy` script, go to your command prompt and type

```
$ griffon ping
```

Upon invoking that command, you should see something like this as output:

```
Welcome to Griffon 0.9.5 - http://griffon.codehaus.org/
Licensed under Apache Standard License 2.0
Griffon home is set to: /usr/local/griffon

Base Directory: /private/tmp/buildEvents
Running script /private/tmp/buildEvents/scripts/Ping.groovy
Resolving dependencies...
Dependencies resolved in 776ms.
Environment set to development
Resolving plugin dependencies ...
Plugin dependencies resolved in 796 ms.
Got 'Howdy!' from ping
```

Excellent! You just wrote your first custom event paired with an event handler. And it only took a few lines of code.

As you can see in `Ping.groovy`, publishing an event is only a matter of calling what appears to be a method named `event()`. This method (which is actually a closure provided by the `Init` script, if you want to get technical about it) takes two parameters: a `String` that identifies the type of event to fire and a `List` of arguments. In this case, the Ping event sends a predefined message as a single argument. Assuming you're working within the boundaries of a Groovy script, you can send any valid Groovy object as an argument—even closures, if that makes sense for the particular problem at hand.

Now you know what to look for in a Griffon script when determining which events can be fired at a specific point. Just do a search on `event()`, and you're in business.

The mechanism to build events is simple yet powerful. Wouldn't it be great if there was a similar mechanism at runtime? It turns out you can have that too.

8.2 Working with application events

An application can fire events in much the same way as build events are fired. That is, the syntax is the same, but the source is a bit different. Don't worry, we'll cover the details. Event handlers, on the other hand, come in several flavors. Given these facts, let's cover event handlers first. As you might expect, there are some default events published by your application that you can try.

8.2.1 E is for events

We trust you remember the mnemonic rule related to Griffon's configuration files (see chapter 2 for a refresher). We're talking about your ABCs: *A* for *application*, *B* for *builder*, and *C* for *config*. Those are the standard (and required) files, but a few more are optional. That's the case with *E* for *events*.

Let's start with a fresh application. Go to your command prompt, and type

```
$ griffon create-app appEvents
```

The simplest way to register an application event handler is to create a file named Events.groovy (be mindful of the missing underscore at the beginning of the filename) and place it under griffon-app/conf. Open your editor and enter the following:

```
onBootstrapEnd = { app ->
   println "Application bootstrap finished"
}
```

Save the Events.groovy file, and run your `appEvents` application. A few lines after the application's jar has been packed and the application has been launched, you should see this on your output:

```
Application bootstrap finished
```

Your event handler is open for business—congratulations! You'll notice a slight difference from build event handlers. The convention here uses `on` instead of `event` as a prefix. This is by design; it marks a clear distinction between build-time and runtime event handlers.

Table 8.2 summarizes all the events fired by every application by default. They are listed in the order you will encounter them as the application loads. All of these events have a single argument: the `app` instance.

Table 8.2 Application events tied to life-cycle phases

Event	Fired when?
`Log4jConfigStart`	After the application's logging configuration has been read.
`BootstrapStart`	After the application's config has been read and before anything else is initialized.
`LoadAddonsStart`	Before any addons have been initialized.
`LoadAddonStart`	Before a specific addon is initialized.
`LoadAddonEnd`	After an addon has been initialized.
`LoadAddonsEnd`	After all addons have been initialized.
`BootstrapEnd`	At the end of the Initialize phase.
`StartupStart`	Before any MVC group is created. Coincides with the Startup life-cycle phase.
`StartupEnd`	After the Startup life-cycle phase has finished.
`ReadyStart`	Before the Ready life-cycle phase starts.
`ReadyEnd`	After the Ready life-cycle phase has finished.
`ShutdownRequested`	When a component calls the `shutdown()` method on the application.
`ShutdownAborted`	When the Shutdown sequence is aborted by a `ShutdownHandler`.
`ShutdownStart`	Before the Shutdown life-cycle phase starts.

NOTE If you're wondering what addons are, skip to chapter 11. But come back to continue learning about application events!

Table 8.3 lists the three other important events fired by an application, which are related to MVC groups.

Table 8.3 Application events launched by `createMVCGroup()` and `destroyMVCGroup()`

Event	Fired when?
`NewInstance`	An object is created via `app.newInstance()` Arguments: `Class` class, `String` type, `Object` instance Example: `event("NewInstance", [FooModel, "model", fooModel])`
`InitializeMVCGroup`	Before group members are initialized—that is, before `mvcGroupInit()` is called Arguments: `MVCGroupConfiguration` config, `MVCGroup` group Example: `event("InitializeMVCGroup", config, group])`
`CreateMVCGroup`	After an MVC group is created Argument: `MVCGroup` group Example: `event("CreateMVCGroup", [group])`
`DestroyMVCGroup`	After an MVC group is destroyed Argument: `MVCGroup` group Example: `event("DestroyMVCGroup", [group])`

We hope that tables 8.2 and 8.3 give you enough information to keep you busy with default events. But the `Events.groovy` script isn't the only way to register application event handlers. There are a few more techniques, which we'll discuss next.

8.2.2 Additional application event handlers

You might have glanced at the base interface of all Griffon applications, if you have a curious nature and a thing for reading source code. `griffon.core.GriffonApplication` defines the following contract related to events:

```
void addApplicationEventListener(Object handler);
void addApplicationEventListener(String name, Closure handler);
void addApplicationEventListener(String name, RunnableWithArgs handler);

void removeApplicationEventListener(Object handler);
void removeApplicationEventListener(String name, Closure handler);
void removeApplicationEventListener(String name, RunnableWithArgs
  handler);
```

You can gather from this that

- Any object may become an application event listener.
- Closures and instances of `RunnableWithArgs` may be registered as application event listeners.

The first statement is true as long as the object follows a specific set of conventions. The conventions are similar to the ones we just laid out for `Events.groovy`. First, a listener may be a map, a script, or an object. Next, depending on the type of object, you have the following options:

- *Map*—Keys must match event names. Values must be closures or instances of `RunnableWithArgs` that take the same number of arguments.
- *Script*—The same rules as `Events.groovy` apply: each event handler is of the form `on<EventName> = { args -> /* code */ }`.
- *Object*—You must define either a method or a closure property that follows the naming convention. Either of the following will work:

```
def onEventName = { args -> /* code }
void onEventName(args) { /* code */
```

Let's look at an example.

EVENT HANDLER OPTIONS IN ACTION

Here's a tip: all controllers are registered as application event listeners automatically. You'll use that fact in this small example.

Once again, let's start with a fresh application. Go to your command prompt and type

```
$ griffon create-app events
```

Open the `EventsController.groovy` file in your editor, and paste in the code found in the following listing.

Listing 8.2 `EventsController` displaying all event-handler options at its disposal

```
class EventsController {
    void mvcGroupInit(Map params) {
        app.addApplicationEventListener([
            ReadyStart: { a ->
                println "ReadyStart (via Map closure event handler)"
            },
            ReadyEnd: new RunnableWithArgs() {
                public void run(Object[] args) {
                    println "ReadyEnd (via Map runnable event handler)"
                }
            }
        ])
        app.addApplicationEventListener("ReadyEnd") { a ->
            println "ReadyEnd (via Closure event handler)"
        }
    }

    def onReadyEnd = { a
        println "ReadyEnd (via Closure property event handler)"
    }

    void onReadyStart(a) {
        println "ReadyStart (via method event handler)"
    }
}
```

As you can see, on this controller you define four event handlers, all of them tied to the start and end of the Ready life-cycle phase. A map-based event listener is registered with the application first. Notice that the keys are the same as the events you intend to handle. The map contains two event handlers; the first is defined as a closure, and the second is an instance of `RunnableWithArgs`. Next a closure is registered as an application event listener; the name of the event is set as the first argument of the `register` method, and the closure is set as the second argument. Next you use a `closure` property as an event handler. It looks similar to an action closure, doesn't it? If that bothers you, then you'll be happy to know that you can choose a method as an event handler too.

Running the application should yield an output like this:

```
ReadyStart (via method event handler)
ReadyStart (via Map closure event handler)
ReadyEnd (via Closure property event handler)
ReadyEnd (via Map runnable event handler)
ReadyEnd (via Closure event handler)
```

Hurray! All the event handlers are working perfectly, and the job wasn't that hard. The power of convention over configuration manifests itself again.

We need to discuss one last important piece of information regarding application event handlers: their relationship with threading concerns.

EVENT HANDLERS AND THREADING

As you may recall from chapter 7, developing multithreaded applications in Swing is hard. Fortunately, you discovered how Griffon greatly simplifies working with multiple threads and the EDT, making a difficult task much simpler. What, then, are the implications of publishing multiple events and having several event handlers ready to process those events?

Well, if you send a lot of events in the EDT and also process them in the EDT, you'll get an unresponsive application. Clearly, there are times when firing an application must be done outside of the EDT, as if the whole call was surrounded with `doOutside{}` or `execOutsideUI{}`. But it's also true that sometimes you want to post an event and return immediately. You can instruct a handler to run inside a particular thread regardless of how the event was published. You can always use any of the threading facilities discussed in chapter 7 to wrap the handler's body; this way, you'll be sure the handler will be executed in a particular thread no matter which thread was used to fire the event.

Pay close attention to this threading fact, because it will most likely trip you up if you publish an event in the EDT and expect the handler to behave synchronously. Remembering that controller actions are executed outside the EDT by default means that publishing an event in the body of an action will happen outside the EDT as well.

If you happen to change the default settings for actions, remember to update the event-firing code accordingly. Oh, that's right, we haven't discussed how events can be fired. That's the topic of the next section.

8.2.3 Firing application events

Recall from earlier in this chapter that you fire a build event by calling `event(eventName, args)` on your script. Well, firing an application event is done pretty much the same way; the only difference is that you have to qualify the call using the `app` variable. As an added benefit, all Griffon applications let you fire application events that don't require an argument without needing to specify an empty list. This is because the remaining contract on `griffon.core.GriffonApplication` defines the following methods:

```
void event(String eventName);
void event(String eventName, List params);
void eventAsync(String eventName);
void eventAsync(String eventName, List params);
void eventOutsideUI(String eventName);
void eventOutsideUI(String eventName, List params);
```

The first pair of methods generates an event that will notify listeners in the same thread as the publisher—in other words, the event is handled synchronously to the publisher. This is the default way to publish application events, because all of them should be handled immediately after they've been posted. But if a listener chooses to handle the event in a different thread, it can use any of the threading facilities we discussed in chapter 7.

The second pair of methods posts an event and return immediately. This means event listeners will be notified in a different thread than the publisher. It doesn't matter if the current thread is the UI thread; these methods guarantee that the event will be handled in a different thread. You can use these methods when your code requires announcing a change but doesn't need to wait for any listeners to finish processing the announcement.

The final pair is a combination of the previous two. These methods post an event, but listeners may or may not be notified in the same thread as the publisher. The condition that controls which thread is used is whether the publisher's thread is the UI thread. In other words, if the event is fired outside the UI thread, then the listeners are notified in the same thread—exactly what happens with the first pair. But if the publisher's thread happens to be the UI thread, the listeners will be notified in a different thread, as with the second pair.

Let's follow up with another example. You'll have a controller fire a Ping event and handle it as well.

SETTING UP PINGCONTROLLER

Start by creating a new application name ping. You'll begin with the event logic and then move forward to the view and model. Open the `PingController.groovy` file that was created with the application. The following listing shows the minimal code required to achieve your goal.

Listing 8.3 `PingController` sending and handling a custom application event

```
class PingController {
    def model
    int count = 0
```

```
    def doPing = { evt = null ->
        model.output = ""
        app.event("Ping")
    }

    def onPing = {
        String text = "Pong! (${++count})"
        executeInsideUIAsync { model.output = text }
    }
}
```

1 Fire Ping event

2 Handle Ping event

`PingController` does nothing more than react to an action, triggered by a view element (which you'll define shortly). The action clears a value in the model, which will be used to display some kind of result judging by its name. Finally it fires a Ping event 1. This event is so simple it doesn't require arguments. Ping events are handled at 2, where you learn that your suspicions about `model.output` were correct. A counter keeps track of how many times the controller has received a Ping event.

Time to move on to the model.

SETTING UP PINGMODEL

You need to add only one property to the model. Open `PingModel` in your editor. It should look like this:

```
import groovy.beans.Bindable

class PingModel {
    @Bindable String output = ""
}
```

Your last task is to fill the details on the view.

SETTING UP PINGVIEW

You know what's coming, don't you? From your editor, open `PingView.groovy`, and then paste in the contents of the following Listing.

Listing 8.4 `PingView.groovy`

```
actions {
  action(id: "pingAction",
    name: "Ping!",
    closure: controller.doPing)
}

application(title: 'Ping',
  size: [200, 100],
  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: 1, rows: 2)
    button(pingAction)
    label(text: bind{ model.output })
}
```

1 Call action handler

2 Wire action to button

3 Bind to output property

Listing 8.4 shows a bare-bones view. Its responsibilities are defining an action that will call out to the controller's `doPing` ❶ action handler. That action is later wired up with a button ❷, which means a Ping event will be fired every time you click the button. Finally, a label ❸ is set up as a witness of the Ping event handler: its `text` property is bound to the model's `output` property.

When you run the application, you'll see a small window with a button and an empty label. Click the button a few times, and you'll see the label's text change. Figure 8.1 shows the state of the application after clicking the Ping! button five times.

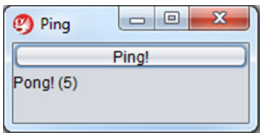


Figure 8.1 Application after clicking the Ping! button five times

Congratulations! Your first custom application event is up and running. But this example is trivial, and its behavior could easily be attained by means of binding to a `model` property. What's the benefit of this approach? Let's add a second MVC group into the mix to find out.

GETTING MULTIPLE CONTROLLERS TO COMMUNICATE

Remember that the `create-app` command created a default MVC group. For this step, you need to create another MVC group named `pong`. Use the `create-mvc` command to make it simpler:

```
$ griffon create-mvc pong
```

This new MVC group will send a Pong event every time it receives a Ping. You'll also fix `PingController` to handle any Pong events that may be sent through the application. Figure 8.2 describes the event flow between MVC groups.

First, open `PongController` in your editor, and add a Ping event handler that triggers a Pong event. This time you'll send a message too:

```
class PongController {
    int count = 0

    def onPing = {
        app.event("Pong", ["Pong! Pong! (${++count})"])
    }
}
```

`PongController` also keeps a count of how many pings it has received. The Pong event sends a message to its handlers; that way you know how many Pings `PongController` has received so far.

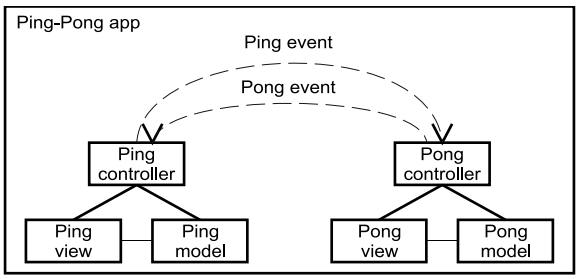


Figure 8.2 When you click the Ping button, the Ping controller fires a Ping event. The event is processed by the Pong controller, which fires a Pong event processed by the Ping controller.

Next, back to `PingController`. You'll fix it so an instance of the `pong` MVC group is created, and append a Pong event handler as well. Append the following snippet at the end of the `PingController` class:

```
void mvcGroupInit(Map args) {
    createMVCGroup("pong")
}

def onPong = { pongText ->
    execInsideUIAsync { model.pongText = pongText }
}
```

> **Creating an MVC group alternative**
>
> We chose to have you create an instance of the `pong` MVC group by calling `createMVCGroup()` directly. Of course, this isn't the only way to do it; you could have added `pong` to the list of groups to be initialized at startup by tweaking `Application.groovy`. (Refer to chapter 6 for more information about MVC groups.)

Now, you need to modify the Ping application's model and view. In the previous code, you can see a reference to `model.pongText`. This is a new `model` property, and you need to add it. Do so by appending the following property definition to `PingModel`:

```
@Bindable String pongText = ""
```

The final step is fixing `PingView` to display the value of `model.pongText`; otherwise you won't be able to see if the Ping and Pong event handlers are properly set up! Open `PingView` again. Locate the layout, button, and label definitions, and overwrite them with the following snippet:

```
gridLayout(cols: 1, rows: 3)
button(pingAction)
label(text: bind{ model.output })
label(text: bind{ model.pongText })
```

That should take care of it. You're ready to launch the application to test it.

WATCHING YOUR CONTROLLERS COMMUNICATE

Type `griffon run-app` at your command prompt. When the application's main frame appears, click the button a few times. You should see something like what's shown in figure 8.3.

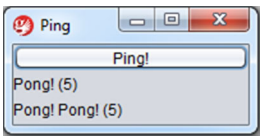


Figure 8.3 Playing ping-pong with events

It works! Now you have the basis of intercommunicating controllers via events. You might have noticed that both controllers define a Ping event handler and there's only one source of Ping events. Extrapolating the conditions of this example, you can have many listeners on the same application event, which is great. But what if you want a bit of privacy? Say you only want Ping and Pong events to be sent between a small set of objects. We're afraid application events won't solve this

problem. What if the event mechanism could be reused somehow? Aha! That's the topic of the next section.

8.3 Your class as an event publisher

Glancing back to the previous discussions of how `griffon.core.GriffonApplication` supports application events and their listeners by means of a set of conventional methods, it looks like you could get away with adding a similar contract to any class. All you would need to do afterward would be to fill in the blanks.

Before you start designing a way to keep track of the multiple options of event listeners—remember, you can register maps, closures, scripts, and any objects—let's think for a second. This task has already been solved by the framework; surely there's a piece of reusable code that you can plug into your own classes. It happens that such support exists in the form of an interface named `griffon.core.EventPublisher`. The contract is as follows:

```
void addEventListener(Object handler )
void addEventListener(String name, Closure handler)
void addEventListener(String name, RunnableWithArgs handler)
void removeEventListener(Object handler)
void removeEventListener(String name, Closure handler)
void removeEventListener(String name, RunnableWithArgs handler)
void publish(String name, List args = [])
void publishAsync(String namename, List args = [])
void publishOutsideUI(String eventName, List args = [])
```

Griffon provides a base implementation of the `EventPublisher` interface as well as a composable component called `EventRouter`. It's usually the case that an event-publishing class implements the `EventPublisher` interface and uses a composed instance of `EventRouter` to delegate all method calls pertaining to events. Great; armed with this knowledge, you can add that contract to your classes, making sure each method makes a delegate call to a private instance of `EventRouter`.

If you think this sounds like copying and pasting a lot of boilerplate code, we couldn't agree more! The task of grafting an `EventRouter` into a particular class sounds eerily familiar to the task of grafting `PropertyChangeSupport` into a class to make it observable. Do you remember how the latter was solved? Using the `@Bindable` annotation and the power of the AST Transformations framework. You can bet there's a similar solution for this case too.

Enter the `@griffon.transform.EventPublisher` annotation and its companion AST transformation. By annotating a class with this annotation, you get all the benefits of event publishing. No need to copy and paste code. How's that for productivity?

We're sure you're itching to try this new feature, so let's dive in! You may be familiar with the game Marco Polo[1] or its variations. In this section, you'll try your hand at an `EventRouter` as you simulate four Marco Polo players.

[1] http://en.wikipedia.org/wiki/Marco_Polo_(game).

8.3.1 A basic Marco-Polo game

The Marco Polo game requires at least three participants. One of the participants is blindfolded. The others can stay put or move around. The blindfolded player must locate one of the other participants by using sound as a guide. At any given time, this player may shout "Marco!" When that happens, all other players must respond by shouting "Polo!" simultaneously. If the blindfolded player locates another player by touching them, then the latter is blindfolded, everyone takes new positions, and the game restarts.

In this section, you'll replicate a portion of the Marco Polo game by having a controller serve as the Marco player and a few beans act as Polo players. All objects will communicate via events.

Create the application by typing the following command at your command prompt:

```
$ griffon create-app marco
```

The next steps in building this Marco Polo example are as follows:

1. Set up the controller (Marco).
2. Create the players (Polo).
3. Set up the model and view.

SETTING UP THE CONTROLLER (MARCO)

The controller contains half the logic you want to implement. It should be able to

- Register players as event listeners
- Fire a Marco event at any time
- React to Polo events sent by players

The next listing demonstrates how each of these responsibilities can be implemented. Note that this code doesn't assert when the searching player locates any of the others.

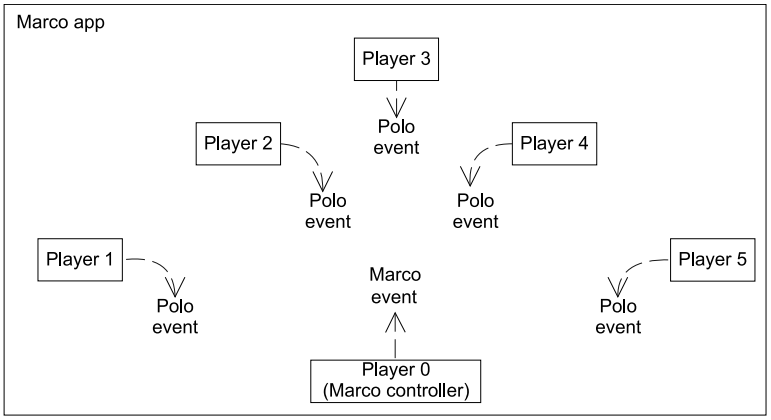


Figure 8.4 A diagram of several players in the Marco Application triggering "Marco" events at any given time

Listing 8.5 MarcoController.groovy

```
import griffon.transform.EventPublisher

@EventPublisher
class MarcoController {
    def model

    void mvcGroupInit(Map args) {
        def createAPlayerAndRegisterIt = { id ->
            def player = new Player(id)
            addEventListener(player)
            player.addEventListener(delegate)
        }
        (1..3).each { createAPlayerAndRegisterIt(it) }
        addEventListener(new Player(4))
        new Player(5)
    }

    def marco = { evt = null ->
        execInsideUISync { model.output = '' }
        publishEvent "Marco"
    }

    def onPolo = { msg ->
        execInsideUIAsync { model.output += msg + "\n" }
    }
}
```

1 Register player with controller

2 Register controller with player

3 Instantiate players

Because any class can be annotated with `@EventPublisher`, you annotate `MarcoController` with it. The controller still has the ability to publish and receive application events, and now it can publish local events. All players are initialized during the controller's initialization phase. A player instance must register itself as a listener on the controller ❶; this way, the player can listen for Marco events. The controller also needs to register itself with the player ❷, because it will listen for Polo events sent by that player. To demonstrate that event listeners aren't automatically registered, you create two additional players ❸. The first registers itself with the controller, but the controller won't listen to events published by it. Finally there's an unconnected player instance; it won't listen to events sent by the controller.

Now let's look at the other part of the equation: the `Player` class.

SETTING UP THE PLAYERS (POLO)

The `Player` class is responsible for

- Registering Polo event listeners
- Posting a Polo event when it receives a Marco

Each `Player` should have an id to distinguish it from other players. You'll print out a message every time a Marco event comes in, so you can verify that each player received the event (or, in the case of Player 5, that they didn't). Figure 8.5 shows the connections between players and events.

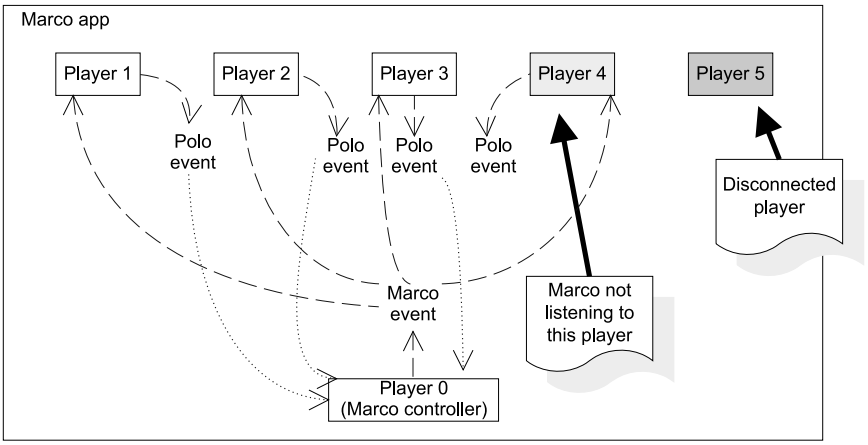


Figure 8.5 Marco Polo event flow

Open your editor once more, and create a file named Player.groovy under the src/main directory. Its contents should be the same as the following listing for now.

Listing 8.6 `Player.groovy` class receiving and sending events

```
import griffon.transform.EventPublisher

@EventPublisher
class Player {
    private final int id

    Player(int id) {
        this.id = id
    }

    def onMarco = {
        println "Player ($id) got a Marco!"
        publishEvent "Polo", ["Polo! ($id)"]
    }
}
```

❶ Handle Marco events

❷ Publish Polo event

Once more you apply the `@EventPublisher` annotation to the `Player` class, making it an automatic event-publishing object like `MarcoController`. You add a Marco event handler ❶ so players can respond to Marco events. Inside the body of the handler, a printout is made and then a Polo event is published ❷ using `publishEvent`, a method injected by the `@EventPublisher` annotation.

Moving on, you need a model property that will be used by the view to display who answered with a Polo event.

ADJUSTING THE MODEL AND VIEW

The only thing left to do is set up the view details. First, though, make sure `MarcoModel`'s source code looks like the next listing.

Listing 8.7 MarcoModel.groovy

```
import groovy.beans.Bindable

class MarcoModel {
    @Bindable String output
}
```

You need a way to trigger a Marco event from the controller and then display any information that may have been set on the model property you just defined. Sounds pretty straightforward, doesn't it? The following listing covers all those points.

Listing 8.8 MarcoView.groovy

```
actions {
    action(id: "marcoAction",
        name: "Marco!",
        closure: controller.marco)
}

application(title: 'Marco! ... Polo!',
  size:[300,160],
  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()
    button(marcoAction, constraints: NORTH)
    scrollPane(constraints: CENTER) {
        textArea(text: bind { model.output })
    }
}
```

All that's left is to launch the application and see what happens.

8.3.2 Running the application

Launch the application by typing `griffon run-app` at your command prompt. When it launches, you see the application's main window, as shown in Figure 8.6.

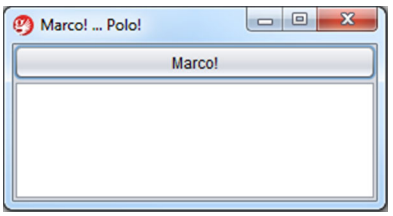


Figure 8.6 Marco application at startup

Click the button. Two things should happen. First, the output at your command prompt should look like the following:

```
Player (1) got a Marco!
Player (2) got a Marco!
Player (3) got a Marco!
Player (4) got a Marco!
```

This means Players 1 through 4 received a Marco event, but Player 5 was left out of the game.

Now look at the application's text area, which should look like figure 8.7.

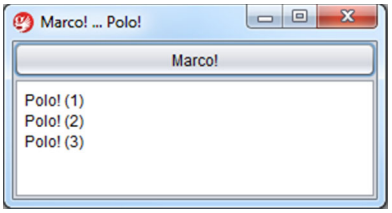


Figure 8.7 Marco application after a Marco event

Only Players 1 through 3 responded with a Polo event that the controller heard. Player 4 did respond (as witnessed in the command prompt's output), but the controller didn't listen to it. Remember that the fourth player listens for Marco events but the `MarcoController` doesn't listen to the player.

That's all there is to enabling local events in any class.

8.4 Summary

Understanding events and how to use them in the Griffon framework is a practical concept that you'll use often. Build events are used by the command-line tools to signal when a build step has started or ended. They also serve to signal other scripts when an action can be taken. We gave the example of creating a directory before sources were compiled, but given that scripts are actually Gant scripts, you can use any available Ant target to accomplish what you need to do.

Then we switched gears to application events. You played Ping-Pong to learn about runtime application events. You also saw how to distinguish them from build events. And you discovered how custom events can be fired and handled.

Finally, you played another game, Marco Polo, to learn how to enhance classes with event-publishing capabilities, effectively turning them into local event publishers.

With a newfound appreciation for events under your belt, it's time to consider testing techniques. The next chapter will look at a few approaches for testing your Griffon applications.

9 Testing your application

This chapter covers

- Testing Griffon applications
- Using FEST for UI testing
- The Spock and easyb plugins
- The CodeNarc, GMetrics, and Cobertura testing and metrics plugins

Let's pause for a moment and review all you've discovered about Griffon so far. First, you've learned that all Griffon applications share the same basic structure. You also know that application artifacts are organized by type and responsibility in specific folders on disk. In addition, the MVC pattern is the cornerstone for Griffon's MVC groups. These, in turn, can be extended by adding other MVC groups within them as members. Finally, you know that you should remember to take threading into account. And you know that all artifacts are glued together via the Groovy language.

You now have models, views, controllers, services, scripts, event handlers, and additional Java/Groovy sources. That's a lot of stuff! And we're not even dealing with the extensions provided by Griffon plugins yet. All this leads to the following questions:

- Where do you start testing?
- What would be a good strategy for tackling tests?
- UI testing: go or no go?
- Is the application solid? Are code paths missing?

A sound answer to the first two questions would be: start small, and build up from there. Another possible answer is: start with the test, and then write the production code. The latter is known as test-driven development (TDD). As expected in this type of scenario, people are divided into two camps: those who say TDD works, and those who say that it's a waste of time. We're not here to try to convince you either way. But should you choose to write tests first, you'll face the issue in the third question.

UI testing requires effort. If you do it manually, it quickly becomes boring; plus manual testing doesn't reliably produce the same results for the same stimuli, because of the human factor involved. Automated UI Testing more often than not requires a lot of setup to be done up front. That's why many developers skip doing these kinds of tests. They can be brittle, because their setup is elaborate; and rapid changes in the underlying code can render them obsolete, which often spells their doom and rapid departure from the codebase.

In addition, when you're using a dynamic language, certain aspects of your code are known at runtime only. But you shouldn't skip testing your software altogether. We think quality should be an integral part of any application, which is why Griffon comes with testing support right from the start. In the immortal words of Phillip Crosby, "Quality is free, but only to those willing to pay heavily for it." Griffon gives you the tools you need, but it's up to you to adapt and apply them to your specific scenarios. Aristotle said, "Quality isn't an act. It's a habit." We believe Griffon's testing facilities will aid you in making that statement a reality.

With all that in mind, let's start with the build-small-then-bigger approach and see where it leads. Later we'll come back to UI testing and see if it's really all that complicated.

First, let's go over some basic testing principles.

9.1 Griffon testing basics

All the Griffon applications you created in the previous chapters have a test file associated with a controller whenever a MVC group is created. If you've used the `create-script` command target to create a custom script, you might have noticed that a test file was created for it, too. And the same happens for a service when using the appropriate command.

In this section, you'll create and run some simple tests. You'll learn how to run individual tests and how to test by type, test by phase, and test by name. You'll apply this information to testing the service portion of a simple dictionary application. You'll first create unit and integration tests and inspect the contents of the created files.

9.1.1 Creating tests

The Griffon commands gently remind you that a test should be written for various artifacts. But the fact that a test for a model class isn't automatically created doesn't mean it's exempt of testing. If it makes sense for your code, by all means make a test for your model class. Griffon has two default command targets for creating tests: `create-unit-test` and `create-integration-test`. Both create a new test file using a default template that's suitable for the type of test that is required.

Assuming you're working on an existing Griffon application (in this case, an empty application named sample), type the following at the command line:

```
$ griffon create-unit-test first
```

This command generates a file named `FirstTests.groovy` under the test/unit/sample directory. Browse to that directory, and open the file in your favorite editor. You should see similar content to the following (white space removed for brevity):

```
package sample
import griffon.test.*

class FirstTests extends GriffonUnitTestCase {
    protected void setUp() { super.setUp() }
    protected void tearDown() { super.tearDown() }
    void testSomething() { fail("Not implemented!") }
}
```

Nothing fancy. Worth noting is the fact that the test class extends from `GriffonUnitTestCase`. This base test class provides a few additions over the standard `GroovyTestCase`. For example, it exposes all methods from `UIThreadManager`. There might be further additions to this class in later releases. Keep an eye on the release notes!

Now let's look at the controller test file that was generated when the application was created. Look in test/integration for a file that bears the name of the application plus the Tests suffix (in this case, SampleTests), and open it in your editor. The following snippet shows the file's contents:

```
package sample
import griffon.core.GriffonApplication
import griffon.test.*

class SampleTests extends GriffonUnitTestCase {
    GriffonApplication app
    protected void setUp() { super.setUp() }
    protected void tearDown() {  super.tearDown() }
    void testSomething() { fail("Not implemented!") }
}
```

Whoa. An integration test class extends from the same base class as a unit test. Well, this means both kinds of tests share the same basic elements. But the other noticeable change is the addition of an `app` property. This property, in theory, will hold the reference to a running Griffon application— the current application under test.

That might make you wonder about the real difference between these two kinds of tests. You see, when unit tests are run, they do so with the smallest set of automatic dependencies possible: zero. The goal of a unit test is to play around with a component in isolation. For this reason, there's no application instance available for unit tests, MVC groups, or services.

> **Mock testing in Groovy**
> While it's true that unit tests should run their components in isolation, there are times when you need to set up a collaborator or a stand-in. This is where mock testing comes into play. There are several Java-based libraries for mocking objects (EasyMock, jMock, Mockito). Groovy has its own version, using its metaprogramming capabilities.

On the other hand, an integration test relies on the actual component relationships being put to the test. This is why an integration test requires a live application.

Now, given that the hierarchy of these tests goes all way back to the basic JUnit TestCase, you can apply JUnit tricks along with some new ones thanks to the power of Groovy. Refer to Groovy's Testing Guide (http://groovy.codehaus.org/Testing+Guide) to learn more about all you can do with the language when it comes to testing.

It's time to run some tests, now that you know how they look.

9.1.2 Running tests

As with many things in Griffon, you can count on a command target to help you at the appropriate time and place. This time it's for running tests. Every test that was created using the `create-*-test` command targets is ready to be run. It's just that the tests do nothing interesting with the code. Nevertheless, go to your command prompt and type

```
$ griffon test-app
```

You should see a few messages about the application code being compiled (if it wasn't up to date already) and then a few more regarding the test code being compiled. Then you'll see a special block of text that specifically mentions the type and number of tests being run. Here's the output of running unit tests on the Sample application:

```
-------------------------------------------------------
Running 1 unit test...
Running test sample.FirstTests...
                  testSomething...FAILED
Tests Completed in 254ms ...
-------------------------------------------------------
Tests passed: 0
Tests failed: 1
-------------------------------------------------------
```

And here's the output for the integration tests:

```
-------------------------------------------------------
Running 1 integration test...
Running test sample.SampleTests...
                    testSomething...FAILED
Tests Completed in 65ms ...
-------------------------------------------------------
Tests passed: 0
Tests failed: 1
-------------------------------------------------------
```

Depending on how the tests go (PASSED or FAILED), you'll get a set of reports in text, XML, and HTML format. Inspect the last lines of the output; you'll see that the reports were placed in the `target/test-reports` directory. The generated HTML reports use the standard templating and conventions that you may already be familiar with. The XML reports also use the same format as standard JUnit, which means you can mine them for data as you would normally do when working with standard Java projects and JUnit.

> **Test phases and types**
>
> From Griffon's point of view, unit and integration tests are phases, not types. The unit phase may contain different types of tests. So far you've seen the standard type. Later in this chapter, you'll learn about the `spock` type. We'll explain phases and types in more detail in the next section.

But there's an additional side effect of running tests in this manner. Maybe you noticed it already: both unit and integration tests are run one after another with a single command invocation. What if you only want to run unit tests, for quicker feedback? Further, what if you only want to run a single test, independent of its phase or type? The `test-app` command is versatile in terms of its options. These and other questions will be answered next.

Running tests by phase or type

Specifying a phase of test to be run is a feature found in the Griffon command's bag of tricks. The same can be inferred about test types. The `test-app` command uses a naming convention on its parameters to recognize your intentions. That convention is `phase:type`.

Want to run all JUnit tests in the `unit` phase? Then type the following:

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

All JUnit test types match their corresponding phase name by default. The default test phases are `unit`, `integration`, and `other`. The `other` phase is used when script tests are available.

This naming convention is also flexible. You can omit either the phase or the type. If the type is omitted, all test types in the same phase are run. (This will make more sense when we introduce the `spock` type.) In the meantime, assuming you have both

JUnit tests and Spock specifications available in the `unit` phase, you can run all of them by invoking

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

But running JUnit tests only and skipping Spock specifications is done by invoking

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

The amazing thing about defining the test type alone is that you can execute all tests of the same type regardless of the phase they belong to. Running all available Spock specifications, including unit and integration ones, is as simple as calling

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

Isn't that a time saver in terms of setup and configuration? We'd like to think so. But we're not done yet. The next section covers running specific tests by name.

RUNNING TESTS BY NAME

Sometimes you need to run a single test because it's the one that's been giving you trouble all morning. There's definitely no need to call all of the test's buddies just to obtain a report on the problem. Again, the `test-app` command is smart enough to recognize several options, depending on what you throw at it.

Need to run a single test file? Then specify it as the sole argument to the command, like this:

```
$ griffon test-app sample.First
```

Note the omission of the Tests suffix in the class name. This is important, because the command will try to match the name to the several types that may be available. Some types use a different file suffix. The command could run more than one test if you have two or more test types that share a name (for example, `sample.FirstTests` and `sample.FirstSpec`). No problem. Add a type specifier, and then you can be certain a single type is called:

```
$ griffon test-app :unit sample.First
```

Phases, types, and test names are additive. Want to run all tests belonging to a specific package, regardless of their phase and/or type?

```
$ griffon test-app sample.*
```

If you want to target a package and all its subpackages, use double asterisks (`**`) instead of a single asterisk.

You can also run all tests of a specific artifact. For example, suppose you have a multitude of tests but only want to run those that affect services. Type the following:

```
$ griffon test-app *Service
```

There's even a more specific option. You can run a single test method if needed:

```
$ griffon test-app First.testSomething
```

All that power in a single, innocent command. Who would have thought it? But we haven't covered all its tricks. A group of build events is triggered whenever a test runs:

events that deal with the overall execution of tests, events that deal with the execution of a particular phase, and events that trigger when a single test starts. The bottom line is that if the testing facilities exposed by the framework aren't enough for your needs, chances are that you can tweak them via custom scripts and build event handlers.

Let's build a new application from scratch and add some tests to it.

9.1.3 Testing in action

You'll build an application and exercise it with some tests. The application is a dictionary query; see figure 9.1. You type a word in a text field, click the Search button, and wait for an answer based on the word's definition.

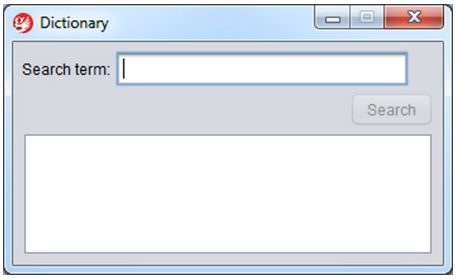


Figure 9.1 Running Dictionary application

We'll keep the code simple, because our intent is to show the testing code, not the production code. Don't forget that you can find the source code for this example at the *Griffon in Action* GitHub site (https://github.com/aalmiray/griffoninaction).

SETTING UP THE APPLICATION AND WRITING A TEST

Start by creating an application named dictionary. You know the magic word already:

```
$ griffon create-app dictionary
```

You now have three artifacts (model, view, and controller) and an integration test. You'll now create a dictionary service. It will serve as your search engine, with the added benefit of a companion dictionary service unit test:

```
$ griffon create-service dictionary
```

You'll fill out the code for the test first; this means you'll get a failing test as a baseline. Then you can fill in the blanks as much as possible to make the test turn green. With a new test state, you can safely refactor the production code: as long as the tests stay green, you're doing it right. Your first attempt verifies some error conditions triggered by insufficient or wrong input, as shown in the following listing.

Listing 9.1 Testing for error outputs in `DictionaryService`

```
package dictionary
import griffon.test.*
import static dictionary.DictionaryService.*

class DictionaryServiceTests extends GriffonUnitTestCase {
    DictionaryService service = new DictionaryService()

    void testServiceCantFindWord() {
            assert service.findDefinition('spock') == FIND_ERROR_MESSAGE
    }

    void testInvalidInput() {
        assert service.findDefinition('') == INPUT_ERROR_MESSAGE
```

Test for undefined word

Test for invalid input

```
        assert service.findDefinition(' ') == INPUT_ERROR_MESSAGE
    }
}
```

Before you run the test, you have to define two constants and one method in your service. Locate the `DictionaryService.groovy` file under `griffon-app/services/dictionary`, and open it in your editor. Make sure its contents resemble the following listing.

Listing 9.2 Adding enough code to `DictionaryService` to turn the test green

```
package dictionary

class DictionaryService {
    static String INPUT_ERROR_MESSAGE = "Please enter a valid word"
    static String FIND_ERROR_MESSAGE = "Word doesn't exist in dictionary"

    String findDefinition(String word) {
        null
    }
}
```

Now you can run the test and see it fail. Execute the `test-app` command, targeting the unit phase and type; in this way, you avoid running the default integration test created when the application structure was initialized. After the code has been compiled and the test has been run, you should see output similar to the following (minus the power assert and exception information) in your console:

```
-------------------------------------------------------
Running 2 unit tests...
Running test dictionary.DictionaryServiceTests...
                    testServiceCantFindWord...FAILED
                    testInvalidInput...FAILED
Tests Completed in 419ms ...
-------------------------------------------------------
Tests passed: 0
Tests failed: 2
-------------------------------------------------------
```

You knew the test was bound to fail from the start. Next you'll fix the code and make the test succeed.

DOES THE TEST PASS?

Go back to the service source, and edit the service method by copying the following snippet:

```
    String findDefinition(String word) {
        if(GriffonNameUtils.isBlank(word)) return INPUT_ERROR_MESSAGE
        FIND_ERROR_MESSAGE
    }
```

Also make sure you add an import statement for `griffon.util.GriffonNameUtils`, a handy class that provides a method that verifies if a `String` is null or empty, among other things. Run the test again; it should succeed this time.

You finish the service implementation by updating the implementation of the `findDefinition()` method. Next you'll add a map as a lookup table for word definitions. The next listing shows the full implementation of the `DictionaryService` class.

Listing 9.3 `DictionaryService` with a lookup table to store definitions

```
package dictionary
import static griffon.util.GriffonNameUtils.isBlank

class DictionaryService {
    static String INPUT_ERROR_MESSAGE = "Please enter a valid word"
    static String FIND_ERROR_MESSAGE = "Word doesn't exist in dictionary"

    static final Map WORDS = [                                  <-- Dictionary
        groovy: "An agile and dynamic language for the Java platform.",
        grails: "A full stack web application development platform.",
        griffon: "Grails inspired desktop application development\
                  platform."
    ]

    String findDefinition(String word) {                        <-- Find/search logic
        if(isBlank(word)) return INPUT_ERROR_MESSAGE
        WORDS[word.toLowerCase()] ?: FIND_ERROR_MESSAGE
    }
}
```

The last step is to update the tests so the reviewed implementation of your service is also tested.

Updating the tests

The following listing displays the entire test case, which now contains a third test method with a helper.

Listing 9.4 Three tests methods exercising `DictionaryService`

```
package dictionary
import griffon.test.*
import static dictionary.DictionaryService.*
import static griffon.util.GriffonNameUtils.isBlank

class DictionaryServiceTests extends GriffonUnitTestCase {
    DictionaryService service = new DictionaryService()

    void testServiceCantFindWord() {
        assert service.findDefinition('spock') == FIND_ERROR_MESSAGE
    }

    void testInvalidInput() {
        assert service.findDefinition('') == INPUT_ERROR_MESSAGE
        assert service.findDefinition(' ') == INPUT_ERROR_MESSAGE
    }

    void testServiceContainsWord() {                            <-- New test method
        ['Groovy', 'Grails', 'Griffon'].each { word ->
            assertValid(service.findDefinition(word))
            assertValid(service.findDefinition(word.toLowerCase()))
```

```
            assertValid(service.findDefinition(word.toUpperCase()))
        }
    }

    static assertValid(String definition) {
        assert !isBlank(definition)
        assert definition != INPUT_ERROR_MESSAGE
        assert definition != FIND_ERROR_MESSAGE
    }
}
```

Helper method used by new test

Run the test once more; it should work correctly. Remember that you can browse the resulting reports, which are written in several formats and located in the `target/test-reports` directory. Figure 9.2 shows, for example, the report generated for your single test case.

That's all you'll do with this application for now. You've seen how unit and integration tests can be created and run. They fit the bill for starting small and then growing the test codebase as needed. Now let's look at the other kind of testing that always give developers trouble: UI testing.

9.2 Not for the faint of heart: UI testing

You may have encountered this scenario: when an application is small, it's fairly easy to run UI tests manually. In other words, you can run the application through a predefined set of scenarios and write down the results. If you notice something doesn't work right, you can hack some code to fix the problem.

But this situation isn't scalable. As soon as the application grows, UI testing becomes a dragging weight, then a constant pain, and finally the thing that must be avoided if you want to make the deadline.

Automated UI testing is nothing new, especially to desktop applications. There are, after all, plenty of options, each one with advantages and pitfalls.

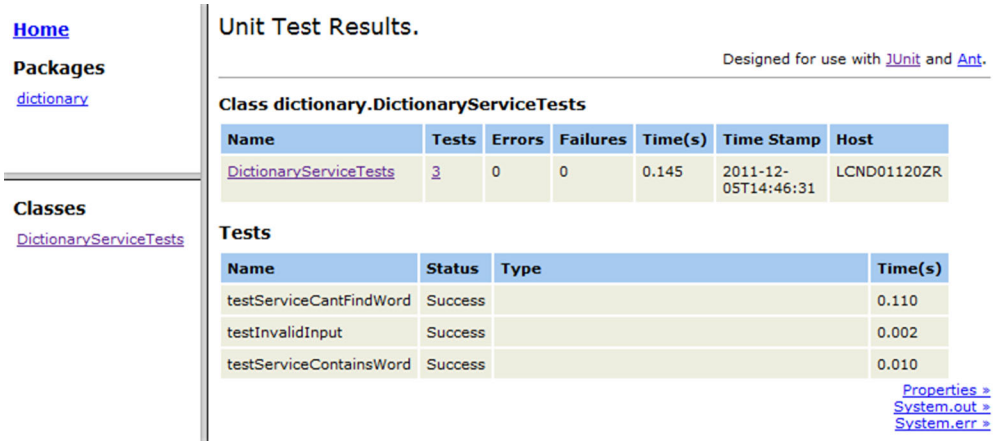


Figure 9.2 HTML report of all the tests you ran on the dictionary application. There's only a single test class with three test methods reported. All succeeded.

If you're a seasoned Swing developer, you've likely heard about the following testing frameworks: Abbot (http://abbot.sourceforge.net/doc/overview.shtml), jfcUnit (http://jfcunit.sourceforge.net), Jemmy (https://jemmy.dev.java.net), and FEST (http://fest.easytesting.org/). These tools work on the assumption that UI components can be found by some programmatic means and that their state can be changed without human intervention. Let's see how they compare to each other.

9.2.1 Setting up a UI component test

Setting up each project varies from tool to tool. Regardless of how the tool must be installed and its dependencies configured, you'll face the following challenge: locating the component on the screen.

Some of the previously mentioned tools rely on a helper class called `java.awt.Robot`. This class is responsible for locating a component on the screen via its coordinates and sending input events as if a human clicked a mouse button on the component or pressed a key. Some rely on a different kind of `Robot`, but that's basically what needs to be done in order to automate this kind of test. Other tools work by referencing the live components.

We'll discuss briefly how each one of the three tools can be used.

ABBOT

In Abbot, for example, you must locate a UI component by conventional means, such as by inspecting a container's hierarchy or accessing a direct reference available in the container instance. Alternatively, you can use Abbot's finder utilities. Next you instantiate a particular Abbot tester that knows how to work with the type of component you want to test. Finally, you invoke the desired behavior on the tester and assert the component's state. Here's a snippet of a `ComponentTestFixture` that shows how it can be done:

```
JTextField textField = getFinder().
    find(new ClassMatcher(JTextField))
JButton button = getFinder().find(new Matcher() {
   public boolean matches(Component c) {
      c instanceof JButton && c.text == "Search"
   }
})
JTextArea textArea = getFinder().find(new ClassMatcher(JTextArea))
JTextComponentTester tester = new JtextComponentTester()
tester.actionEnterText(textField, "Griffon")
tester.actionClick(button)
assert "Griffon is cool!" == textArea.text
```

It works, but it's a bit verbose. We've highlighted two parts of the code. The first shows one of the many options that Abbot exposes to locate a component; this one locates a `JTextField` by type. The second highlighted bit creates an Abbot helper object with the responsibility of sending stimuli to the located UI components. Perhaps if the finders were wrapped with a friendlier abstraction, you would be able to write less code. Sadly, that abstraction isn't provided by default by Abbot.

JFCUNIT

jfcUnit provides a better abstraction for its find mechanism. The following snippet of a `JFCTestCase` shows how the same test behavior can be attained with jfcUnit:

```
Window window = app.windowManager.windows[0]
NamedComponentFinder finder =
     new NamedComponentFinder(JComponent.class, "word")
JTextField textField = (JTextField) finder.find(window, 0)
finder.setName("search")
JButton button = finder.find(window, 0)
finder.setName("result")
JTextArea textArea = finder.find(window, 0)
textField.setText("Griffon")
getHelper().enterClickAndLeave(new MouseEventData(this, button))
assert "Griffon is cool!" == textArea.text
```

Although the finder utilities are much better than Abbot's, the API resembles Java as it was before generics were introduced. You need to cast the found component to the appropriate type. The same is true in Abbot. Now we're down to Jemmy.

JEMMY

Jemmy is popular in the Java Swing community, perhaps due to its close relationship to NetBeans. The following Jemmy snippet illustrates a more concise API for finding components and asserting their state:

```
Window window = app.windowManager.windows[0]
JFrameOperator window = new JFrameOperator("App title")
JTextFieldOperator textField = new JtextFieldOperator(window)
JButtonOperator button = new JButtonOperator(window, "search")
JTextAreaOperator textArea = new JTextAreaOperator(window)
textField.typeText("Griffon")
button.push()
assert "Griffon is cool!" == textArea.text
```

Much better. But we omitted a few details that can get messy in Jemmy when combined with Griffon: the proper application initialization inside a Jemmy test case. Of course, you know that these examples are simple and that these frameworks provide other capabilities, such as record-replay, externalized test configuration using XML, and deep finder features. But the truth is that more often than not, you'll face UI testing code like that you've just seen. It's not pretty, and it's not easy to read. What if there was a way to apply a DSL-like approach to UI testing—perhaps something similar to the SwingBuilder DSL with which you're already familiar?

Enter Fixtures for Easy Testing (FEST). It's aptly named because that's precisely what it does.

FEST

The origins of FEST can be traced back to an Abbot-based extension for TestNG. It was later reworked from the ground up to be a separate project from Abbot, while also adding support for JUnit. What makes FEST shine is its fluent interface design. Here's the same test code you've seen before, this time as a FEST snippet:

```
Window mainWindow = app.windowManager.windows[0]
FrameFixture window = new FrameFixture(mainWindow)
window.textBox("word").enterText("Griffon")
window.button("Search").click()
window.textBox("result").requireText("Griffon is Cool!")
```

That's concise and readable at the same time. Doesn't this look like a winning proposition for UI testing? Let's continue with a real-world example of UI testing using FEST and Griffon.

9.2.2 A hands-on FEST example

Time to get your hands dirty with the FEST API. With the bar green, you can concentrate on building the remainder of the dictionary application as you left it in section 9.1.3.

CONTROLLER

The following listing displays half of the application's logic: the controller. Its job is to process the input typed by the user, call the `DictionaryService` to find a definition for that input, and then send that definition back to the UI.

Listing 9.5 Implementation of `DictionaryController`

```
package dictionary

class DictionaryController {
    def model
    DictionaryService dictionaryService

    def search = {
        model.enabled = false
        String word = model.word

        try {
            String definition = dictionaryService.findDefinition(word)
            execInsideUIAsync { model.result = "${word}: $definition" }
        } finally {
            execInsideUIAsync { model.enabled = true }
        }

    }
}
```

You can see that the controller requires an instance of `DictionaryService`; that was to be expected. It also requires three properties to be available in the model. Those properties are as follows:

- `word`—Contains the input from the user
- `result`—Holds the word's definition or an error message
- `enabled`—Controls the Search button's enabled state

MODEL

The model, in turn, is defined by the following listing.

Listing 9.6 Implementation of `DictionaryModel`

```
package dictionary

import groovy.beans.Bindable
import griffon.transform.PropertyListener
import griffon.util.GriffonNameUtils

class DictionaryModel {
    @ PropertyListener (modelEnabler)
    @Bindable String word
    @Bindable String result
    @Bindable boolean enabled

    def modelEnabler = {
        enabled = !GriffonNameUtils.isBlank(it.newValue)
    }
}
```

1 Listener AST

2 enabled property

The model contains all three previously discussed properties plus a local `PropertyChangeListener` wired up using the `@PropertyListener` AST transformation 1. This listener toggles the value of the `enabled` property 2 depending on the value of the `word` property. The value will be true as long as the `word` property is neither null nor empty.

VIEW

The final artifact to be defined is the view, where all the things you wrote previously come together. This is where you see the model properties being bound to UI components.

Listing 9.7 Implementation of `DictionaryView`

```
package dictionary

actions {
    action(searchAction,
        enabled: bind {model.enabled})
}

application(title: 'Dictionary',
  pack: true, resizable: false, 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')
    label 'Search term:', constraints: 'left'
    textField name: 'word', columns: 20,
         text: bind('word', target: model), constraints: 'wrap'
    button searchAction, name: 'search',
           constraints: 'span 2, right, wrap'
    scrollPane(constraints: 'span 2') {
        textArea name: 'result', editable: false,
                 lineWrap: true, wrapStyleWord: true,
                 columns: 28, rows: 5, text: bind {model.result}
    }
}
```

Define search action

Lay out screen

Note that some components like the text fields, button, and text area have a `name` property. This is because FEST uses the component name to locate components. If a `name` property wasn't defined, you would have to resort to finding the component by type, which could result in the wrong component being found, depending on how the components are laid out. The critical part of making any GUI tests reliable is finding the components in a reliable manner; small changes in the layout shouldn't break the tests. The advantages of declaring this property will become apparent when you create your first FEST test.

There's one additional step before you can run the application and test it manually. You must install two plugins: miglayout and actions. Otherwise, the `migLayout()` node won't be resolved at runtime; nor will the `searchAction` variable be created automatically in the view. Again, you know the magic words already:

```
$ griffon install-plugin miglayout
$ griffon install-plugin actions
```

Run the application to kick its tires. Now, let's continue with the next task: installing and running FEST.

INSTALLING FEST

It should come as no surprise that Griffon sports a FEST plugin. This plugin not only adds the required build-time libraries but also provides a set of scripts that help you create FEST-based tests. Install the FEST plugin by invoking the following command:

```
$ griffon install-plugin fest
```

You'll notice that two additional scripts are now available to you. You'll also see that FEST has a dependency on the Spock plugin. Although the FEST plugin provides integration with Spock, it by no means forces you to use Spock to run FEST. But it's likely that you'll want to do so once you discover the goodness that Spock brings (we'll discuss this later in the chapter).

Create a FEST test for this application by invoking the following command:

```
$ griffon create-fest-test dictionary
```

It's likely that you'll be prompted to overwrite the `DictionaryTests.groovy` file located at `test/integration`. Given that you haven't made changes to this file yet, it's safe to overwrite it. Now open it in your editor. The next listing shows the content of a freshly created FEST test.

Listing 9.8 FEST test for the Dictionary application

```
package sample
import org.fest.swing.fixture.*
import griffon.fest.FestSwingTestCase

class DictionaryTests extends FestSwingTestCase {
    // instance variables:
    // app    - current application
    // window - value returned from initWindow()
    //          defaults to app.windowManager.windows[0]
```

1 **Extend from FestSwingTestCase**

2 **Specify instance variables**

```
    void testSomething() {}

    // protected void onSetUp() {}

    // protected void onTearDown() { }

    // protected FrameFixture initWindow() {
    //     return new FrameFixture(app.windowManager.windows[0])
    // }
}
```

❸ Return first window

The first thing you notice is that the base test case class is FEST-specific ❶. This base class takes care of initializing the application up to the correct phase (in this case, the `main` phase) and sets up a pair of useful properties ❷: the `app` property with which you're already familiar from integration tests, and a `window` property that points to a `FrameFixture` instance. These properties belong to `FestSwingTest`, the superclass of your test class, which is why they've been commented out, as a reminder.

Fixtures are how FEST abstracts a UI component and queries its state. Fixtures are organized in a similar hierarchy to their respective Swing components. There's a `ComponentFixture`, a `ContainerFixture`, and so on. The FEST team has paid close attention to making the design of the FEST API easy to understand. It's IDE friendly because many methods use proper generic signatures. Its fluent interface design also helps to chain methods. You'll discover the full impact of these benefits later in this section.

As you can see in the `initWindow()` method ❸, the first window managed by the application is returned. This becomes the value of the `window` property. Be sure to override this method if you need to test a different window.

The next step in this exercise is to fill out the test code.

YOUR FIRST FEST TEST

Let's start with a sanity-check test. If you recall how the model, view, and controller are set up, the Search button starts in disabled mode and is only enabled when the user types a word in the text field. You'll now write a test to verify that, as shown in the next listing.

Listing 9.9 Test that verifies the Search button is disabled

```
package dictionary
import org.fest.swing.fixture.*
import griffon.fest.FestSwingTestCase

class DictionaryTests extends FestSwingTestCase {
    void testInitialState() {
        window.button('search').requireDisabled()
    }

    protected void onTearDown() {
        app.models.dictionary.with {
            word = ""
            result = ""
        }
    }
}
```

❶ Display disabled Search button

❷ Set model to known state

This test turns out to be a walk in the park. Remember setting a `name` property on the Search button back in listing 9.7? Well, in this test you put it to good use. Notice how the window fixture can locate the button effortlessly by specifying the name of the component you want to find. Pretty much all components can be found in this way.

The `button()` method on the window fixture returns another FEST fixture, one that knows how to query the state of a pushbutton. You can, for example, query its text and its enabled state, which is precisely what you need for this test ❶. FEST's fixtures provide a set of `requireX` methods to query and assert a component's state. The FEST team has gone to great lengths to make error messages comprehensible and unambiguous. But should you need to check a component's state directly, or if a `requireX` method isn't available in the fixture for some reason, you can get the wrapped component and query it directly. Just remember that reading properties from a UI component must be done inside the UI thread. This is a task that the FEST fixtures and their `requireX` methods do well, and they shield you from dealing with inappropriate threading code.

The remaining code of the test ❷ resets the application's state by changing property values in the model. It may seem unimportant at this point, because you only have a single test; but this will become a key aspect for additional tests, because the same application instance is shared across all integration tests. This means tests must be good citizens and clean up after themselves—otherwise you could end up with a red bar caused by a false result.

VERIFYING THE SERVICE ONCE MORE

Now that you've seen FEST in action, you'll write another test that runs the happy path: typing in a word that exists in the dictionary and clicking the Search button results in the definition being shown in the result area. Pay close attention to the last sentence, and look at the following snippet. Does the text description match the code to the letter?

```
void testWordIsFound() {
    window.with {
        textBox('word').enterText('griffon')
        button('search').click()
        textBox('result')
            .requireText('griffon: Grails inspired desktop\
                          application development platform.')
    }
}
```

Outstanding! The code matches the description. This is the real power of the FEST API: it's expressive. Add a bit of Groovy goodness, such as the optional semicolons, and you get a concise DSL for UI testing.

Let's verify the opposite case: that a word that doesn't exist results in an error message being displayed. The test mirrors the last snippet, albeit with a few small changes:

```
void testWordIsNotFound() {
    window.with {
```

```
            textBox('word').enterText('spock')
            button('search').click()
            textBox('result')
                .requireText("spock: Word doesn't exist in dictionary")
        }
}
```

And there you go. These three tests are all that is required to assert that the application is working correctly for now. Assuming all the tests went well, you should end up with a report that looks like figure 9.3.

The FEST API provides fixtures for all components found in the JDK. There are also a few extensions available for SwingX, JIDE Common Layer (JCL), and even Flamingo. Chances are that there might already be a fixture for the component you need to test. If that isn't the case, don't fret; the FEST wiki has plenty of information that will help you dive deeper into the API. The FEST forum is also an active place where you can get your questions answered.

Time to move forward to other types of tests. Yes, it's time to look at Spock and other goodies.

9.3 Testing with Spock and easyb

We've mentioned before that JUnit isn't the only type of test that can be run when invoking the `test-app` command. There's Spock, which we've briefly mentioned previously. There's also easyb, a behavior-driven development (BDD) test framework for Java and Groovy. We'll cover Spock first, because it's perhaps the more exotic option, and then we'll turn to easyb.

9.3.1 Spock reaches a new level

The Spock framework (http://spockframework.org) is the brainchild of Peter Niederwieser, a nice Groovy fellow living in Austria. In Peter's own words, "Spock takes inspiration

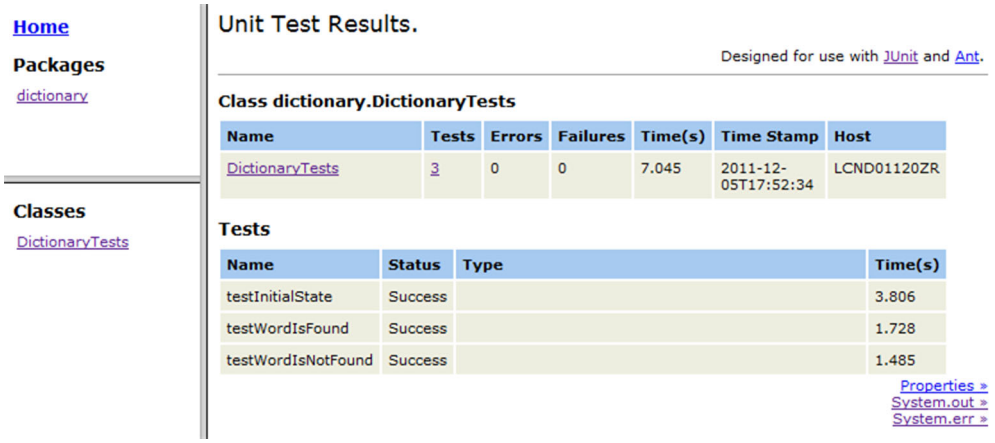


Figure 9.3 HTML report generated for the FEST integration test. All tests succeeded.

from JUnit, jMock, RSpec, Groovy, Scala, Vulcans,[1] and other fascinating life forms." Spock is a Groovy-based tool for testing Java and Groovy applications. Touting itself as a "developer testing framework," it can be used for anything between unit and functional tests. What sets Spock apart from the competition is its unique DSL, which lets you write tests that are far more succinct and expressive than, say, JUnit. Besides that, Spock comes with its own tightly integrated mocking framework and provides extensions for popular application frameworks like Grails and Spring.

Let's take a closer look. Install the Spock plugin, and create a unit specification for `DictionaryService`:

```
$ griffon install-plugin spock
$ griffon create-unit-spec DictionaryService
```

You should now have a file named `DictionaryServiceSpec.groovy` in the test/unit directory. Open it in an editor, and what do you see? The weirdest name for a test method:

```
package dictionary
import spock.lang.*

class DictionaryServiceSpec extends Specification {
    def 'my first unit spec'() {
        expect:
            1 == 1
    }
}
```

It's true: the name of the method is a Groovy string. In case you didn't know it, the JVM supports method names with characters that the Java language doesn't allow, like spaces, but that the Groovy compiler can allow. Therefore, Spock's AST transform moves test method names into annotations and replaces them with safe names in the AST. The Spock runtime later undoes the effect, for example, by applying the reverse replacement in stack traces. Another feature that might have caught your eye is the usage of a block label and the fact that the expectation (an assertion) doesn't require the `assert` keyword.

Spock uses statement labels to divide test methods into *blocks*, each of which serves a special role. For example, the `when` block exercises the code under test, and the `then` block describes the expected outcome in terms of assertions and mock expectations. Another frequently used block is `expect`, a combination of `when` and `then` that's used for testing simple function calls. Every statement in a `then` or `expect` block that produces a value (whose type isn't `void`) is automatically treated as an assertion, without having to use the `assert` keyword.

What follows is a crash course on Spock features. You'll update the `DictionaryService` specification to mirror the previous test you wrote. First you'll test the error states that occur when invalid input or no input is given to an instance of the `DictionaryService`. The following listing shows how the specification looks.

[1] Vulcans?! Seriously: http://en.wikipedia.org/wiki/Vulcans.

Listing 9.10 `DictionaryServiceSpec` with two test methods

```
package dictionary
import spock.lang.*

class DictionaryServiceSpec extends Specification {
  def srv = new DictionaryService()

  def "No input results in an error"() {
    expect:
      srv.findDefinition('') == DictionaryService.INPUT_ERROR_MESSAGE
  }

  def "Wrong input results in an error"() {
    expect:
      srv.findDefinition('spock') == DictionaryService.FIND_ERROR_MESSAGE
  }
}
```

Test no input

Test not found input

Nothing new here from the original template. You add two test methods for each of the error conditions you may encounter with your service. Perhaps the biggest win so far is a more descriptive test name and implicit assertions. Next you'll add a third method that checks the existence of words in the dictionary. Recall from listing 9.4 that you had to roll your own assertions and that you looped through a list of words. The following listing shows one way to do it with Spock.

Listing 9.11 `DictionaryServiceSpec`: correct input results

```
@Unroll("Entering '#word' results in '#definition'")
def "Correct input results in a definition being found"() {
  expect:
    definition == srv.findDefinition(word)

  where:
   word      | definition
   'Groovy'  | 'An agile and dynamic language for the Java platform.'
   'Grails'  | 'A full stack web application development platform.'
   'Griffon' | 'Grails inspired desktop application development platform.'
}
```

Pay close attention to the `definition` line. Notice the usage of two variables that have yet to be defined. Then look at the contents of the `where:` block. It looks like a table, and the undefined variables appear to be column headers. Now observe the annotation that's attached to the test method on the first line: you'll see that the undefined variables are used again. This is a Spock feature called *data tables*, and it goes like this:

- Spock runs the code block marked with `expect:` as many times as rows are found in the data table defined in the `where:` block.
- The column headers represent variable placeholders. For each iteration, they take the appropriate row value according to their column index.
- The `@Unroll` annotation makes sure a correct number of test methods are reported, even using the variable placeholder values to adjust the method names.

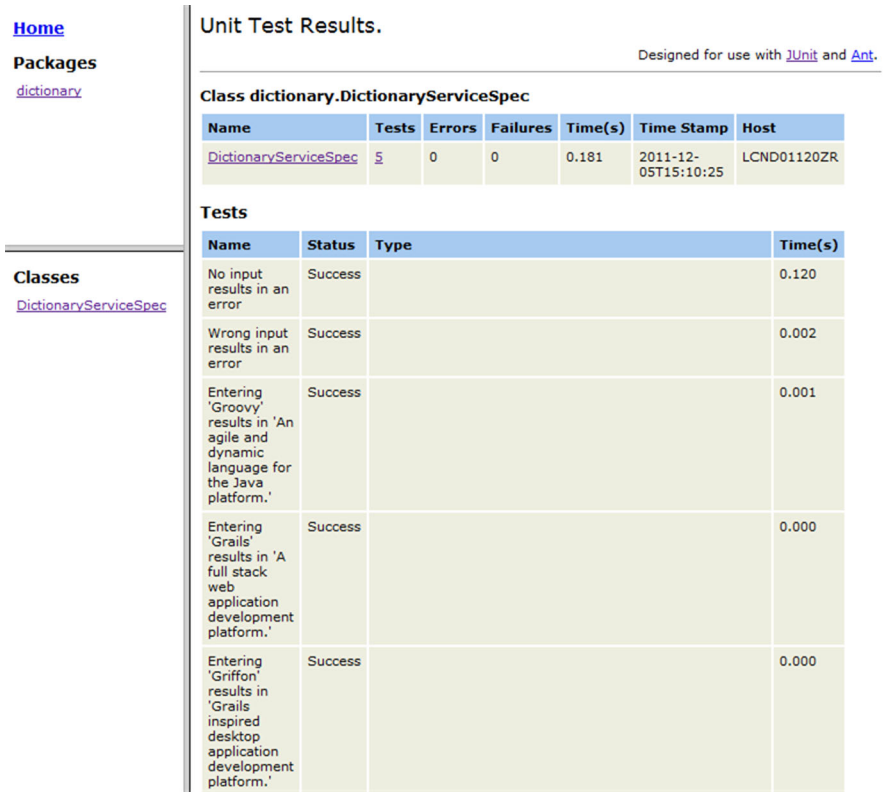
It's time to run this specification to see what happens. You'll take advantage of the `phase:type` test targeting to run this spec and this spec only:

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

After the spec is compiled, you should see output similar to the following:

```
-------------------------------------------------------
Running 3 spock tests...
Running test dictionary.DictionaryServiceSpec...PASSED
Tests Completed in 217ms ...
-------------------------------------------------------
Tests passed: 5
Tests failed: 0
-------------------------------------------------------
```

The console report initially states that three Spock tests will be run. Those three tests match the test methods you just wrote. But the number of tests passed is slighter larger— five. This means the `@Unroll` annotation generated two additional methods, given that there are three rows in the data table. The generated HTML confirms this too, as you can see in figure 9.4.



Home

Packages

dictionary

Classes

DictionaryServiceSpec

Unit Test Results.

Designed for use with JUnit and Ant.

Class dictionary.DictionaryServiceSpec

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
DictionaryServiceSpec	5	0	0	0.181	2011-12-05T15:10:25	LCND01120ZR

Tests

Name	Status	Type	Time(s)
No input results in an error	Success		0.120
Wrong input results in an error	Success		0.002
Entering 'Groovy' results in 'An agile and dynamic language for the Java platform.'	Success		0.001
Entering 'Grails' results in 'A full stack web application development platform.'	Success		0.000
Entering 'Griffon' results in 'Grails inspired desktop application development platform.'	Success		0.000

Figure 9.4 HTML report of `DictionaryServiceSpec`. Pay close attention to the name of the third method. Notice that the variable placeholders have been replaced with values from the data table.

There are of course more things you can do with Spock, but we'll leave it for now. Well, not exactly. Curious about using Spock and FEST together? Keep reading!

9.3.2 FEST-enabled Spock specifications

The FEST plugin comes with Spock support out of the box. There's an additional script named `create-fest-spec` just waiting to be put to work. Create a FEST specification for your Dictionary application:

```
$ griffon create-fest-spec Dictionary
```

This specification will be placed under the test/integration/dictionary directory, given that it requires a running application. Open it, and enter the following code.

Listing 9.12 Full implementation of the `DictionarySpec` specification

```
package dictionary
import griffon.fest.*
import org.fest.swing.fixture.*

class DictionarySpec extends FestSpec {
  def "Initial state: 'Search' button is disabled"() {
    expect:
      window.button('search').requireDisabled()
  }

  def "Typing in a known word results in the
  ➥ definition being displayed"() {
    when:
      window.with {
        textBox('word').enterText('griffon')
        button('search').click()
      }
    then:
      window.textBox('result')
        .requireText('griffon: Grails inspired desktop application
        ➥ development platform.')
  }

  def "Typing in an unknown word results
  ➥ in an error message"() {
    when:
      window.with {
         textBox('word').enterText('spock')
         button('search').click()
      }
    then:
      window.textBox('result')
        .requireText("spock: Word doesn't exist in dictionary")
  }

  void onCleanup() {
    app.models.dictionary.with {
      word = ""
```

Test name

Expected result

Test name

Stimulus/ simulated user input

Expected result

Test name

Stimulus/ simulated user input

Expected result

```
            result = ""
        }
    }
}
```

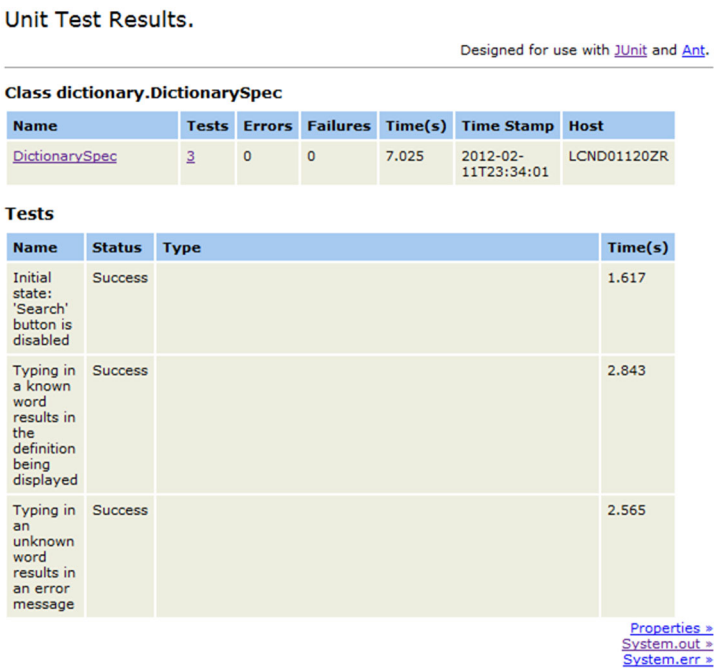
Although we don't show it, you could use a data table in the first test method. Because this specification and the previous integration test case are so simple, there seems to be little advantage in using Spock and FEST together. But as soon as the tests grow, you'll notice the difference. Being able to type fewer keystrokes while remaining expressive is a real productivity booster. Figure 9.5 shows the generated reports, which closely resemble previous reports. Spock, FEST, and JUnit integrate seamlessly in Griffon.

That was quite the fun ride! Let's leave Spock and take a glance at easyb.

9.3.3 easyb eases up BDD

easyb (http://easyb.org) is a BDD[2] framework created and lead by Andrew Glover. If Andy's name sounds familiar to you, it may be because he is a coauthor of the *Groovy in Action* book—specifically, of the testing chapter, no less.

BDD is seen by many as the successor to TDD. It allows a team of disparate people (developers, QA, and stakeholders) to come together and agree on what the application must do and how it should do it. Perhaps this doesn't sound different from a typical waterfall meeting. The catch, however, is that the agreements are recorded in a


Unit Test Results.

Designed for use with JUnit and Ant.

Class dictionary.DictionarySpec

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
DictionarySpec	3	0	0	7.025	2012-02-11T23:34:01	LCND01120ZR

Tests

Name	Status	Type	Time(s)
Initial state: 'Search' button is disabled	Success		1.617
Typing in a known word results in the definition being displayed	Success		2.843
Typing in an unknown word results in an error message	Success		2.565

Properties »
System.out »
System.err »

Figure 9.5 Test report of a FEST+Spock specification after it runs successfully

[2] http://en.wikipedia.org/wiki/Behavior_driven_development.

language that's nontechnical: that of the stakeholders. Yes, plain natural language. Then another twist comes in: developers can take that language and produce matching code that exercises the application behavior as expected. The trick is in the tool that's used to record the agreements and execute the code at the same time. Does this sound like a good deal? Let's try it. Install the easyb plugin, and create a basic story:

```
$ griffon install-plugin easyb
$ griffon create-unit-story dictionaryService
```

The file created is named `DictionaryServiceStory.groovy` and is placed in test/unit/dictionary. This file contains a user story (a rather simple one) to give you a hint of what easyb can do. The next listing shows the file's contents.

Listing 9.13 Basic easyb story as created by default

```
package dictionary

scenario "Hello Groovy", {
   given "A prefix string 'Hello '", {
      prefix = "Hello "
   }

   and "A name is chosen, such as 'Groovy'", {
      name = "Groovy"
   }

   when "Both the prefix and name are concatenated into a greeting", {
      greeting = prefix + name
   }

   then "The greeting should be equal to 'Hello Groovy'", {
      greeting.shouldBe "Hello Groovy"
   }
}
```

Easyb methods ❶

This is a Groovy script with a few keywords ❶ like `scenario`, `given`, `and`, `when` and `then`. These keywords are methods that the easyb framework understands. Each method takes two parameters. The first is a `String` and is typically used to register the behavior in the terms of the stakeholder. The second is what developers write to fulfill the stakeholder's expectations, surrounded by a Groovy closure.

Running the specification results in a report similar to that written by a JUnit test. But you get an additional report that presents the story in layman's terms. Run the story by invoking the following command:

```
$ griffon test-app unit:easyb
```

After the command has finished, look in target/test-reports/plain. You'll see two text files. Open the one that has *stories* in its name. This is what it should contain:

```
1 scenario executed successfully. (0 behaviors were ignored)
  Story: dictionary service
    scenario Hello Groovy
      given A prefix string 'Hello '
      given A name is chosen, such as 'Groovy'
```

```
      when Both the prefix and name are concatenated into a greeting
      then The greeting should be equal to 'Hello Groovy'
```

That's language that stakeholders can understand perfectly. With easyb, you can go from one mode (stakeholder) to the next (developer) and back without much effort.

You might be wondering about the `create-integration-story` script. Yes, it works under the same rules as a JUnit integration test or a Spock integration specification. Yes, this also means you can combine easyb and FEST. Having several options is great, isn't it? We won't reproduce an easyb+FEST story; its content is similar to what you've seen already, due to the simple nature of the application. But you can modify the current story to have it exercise the `DictionaryService`. The following listing defines a single scenario for the happy path: the service is asked for a definition you know for certain can be found.

Listing 9.14 easyb story that tests whether a word can be found

```
package dictionary
import static dictionary.DictionaryService.*

scenario "DictionaryService can find the word 'Griffon'", {
   given "an instance of DictionaryService is available", {
       service = new DictionaryService()
   }

   when "the word 'Griffon' is used as parameter", {
       result = service.findDefinition('Griffon')
   }

   then "the definition should be found", {
      result.shouldBe "Grails inspired desktop application development\
                       platform."
   }
}
```

Running this story results in the following report found in target/test-reports/plain/easyb-stories-unit.txt:

```
1 scenario executed successfully.
  Story: dictionary service
    scenario DictionaryService can find the word 'Griffon'
      given an instance of DictionaryService is available
      when the word 'Griffon' is used as parameter
      then the definition should be found
```

easyb makes a distinction between stories and scenarios. You can create scenario hierarchies and even have scenarios as preconditions of other scenarios. The possibilities multiply with each scenario you write.

Now you know there are several ways to test a Griffon application, but tests aren't the only tools you can use to measure the health of an application. You should apply metrics to it too, and that's precisely the topic we'll discuss in the next section.

9.4 Metrics and code inspection

Code-analysis tools have existed for the Java language since its early days. A myriad of options exist, both in the open source and commercial software spaces. In this section, we'll mention the open source Java-based tools, but our focus is on Groovy-based code analysis tools. We'll look at how they can be used to code issues.

Let's start with two plugins that deal with Java code.

9.4.1 Java-centric tools: JDepend and FindBugs

A lot of ink has been devoted to JDepend and FindBugs. A quick search on the internet will give you plenty of results ranging from the tools' official websites to blog posts, forums, presentation slides, and, last but not least, configuration and code examples.

JDepend (www.clarkware.com/software/JDepend.html) is a tool mainly used to measure the coupling within classes. Unfortunately, it gives accurate readings for Java source code only. But given that a Griffon application supports both Groovy and Java sources, it might be a good idea to run this tool against a codebase of moderate to large size. Of course, there's a JDepend plugin for Griffon that you can install.

FindBugs (http://findbugs.sourceforge.net) is a popular choice with Java developers. This tool comes loaded with tons of rules and metrics that can help make your code rock solid by identifying both common and obscure pitfalls of the Java language. FindBugs can be run against Groovy source too, but there may be some false positives given that the code generated by the Groovy compiler hasn't been migrated completely (this claim is true for the 1.7.x series at least).

Here's an example. The compiler generates calls that create new instances of primitive wrappers as was the custom in Java 1.4; that is, the compiler generates bytecode that calls `new Integer(1)` and `new Long(2)`, for example. Referencing primitive wrappers since Java 1.5 has changed slightly, to `Integer.valueOf(1)` and `Long.valueOf(2)`. The former calls are inefficient and should be avoided whenever possible, in favor of the latter form. Still, you'll get accurate readings for your Java source code. As you might expect, there's a FindBugs plugin for Griffon too.

If these tools are Java-source centric, what can be done for Groovy source? It turns out a couple of tools are available that can produce similar reports, but that work specifically with Groovy source. Those tools are CodeNarc and GMetrics plugins. We'll also cover the Cobertura plugin.

9.4.2 Reporting Groovy code violations with CodeNarc

The elevator pitch for CodeNarc (http://codenarc.sourceforge.net) is "FindBugs for Groovy code." Ambitious but true. CodeNarc's creator, Chris Mair, has done what some thought was a very difficult task: build a static code analysis tool for a dynamic language. What makes CodeNarc special on its own is that it relies on Groovy's AST transformation framework to walk the Groovy code and discover code violations. Yes,

the same AST transformation framework used by `@Bindable`. Give it a try and see what you get for the Dictionary application:

```
$ griffon install-plugin codenarc
```

That should do the trick. This plugin adds a new script that calls CodeNarc's code processor with a standard configuration. You can change and tweak that configuration at will. Be sure to review the plugin's documentation page (http://griffon.codehaus.org/Codenarc+Plugin) to learn about those tunable configuration flags. Running CodeNarc as follows against the current application yields the report shown in figure 9.6:

```
$ griffon codenarc
```

Hmmm. The report looks clean. Is it true that the code is well crafted? Yes, at least for CodeNarc's standard set of violations. You'll now change the code a bit and run CodeNarc again, to see if you get a different picture of the codebase. Sprinkle a few declared but unused variables in the code, paired with a couple of unnecessary imports. Make sure you catch an exception in the controller code (you already have a `try/finally` block that can be used for this negative refactoring). Ready? Re-run CodeNarc, and inspect the generated report. You might get one that looks like figure 9.7.

Excellent! Wait— we're getting excited because the code now fails a set of violations? Ahem. Well, you just verified that the code was in a pristine and healthy condition. The generated CodeNarc report contains links to explanations of each rule validation. It also links to the offending source lines and code, making your job of fixing the problems much easier.

Next in our list of tools is the GMetrics plugin.

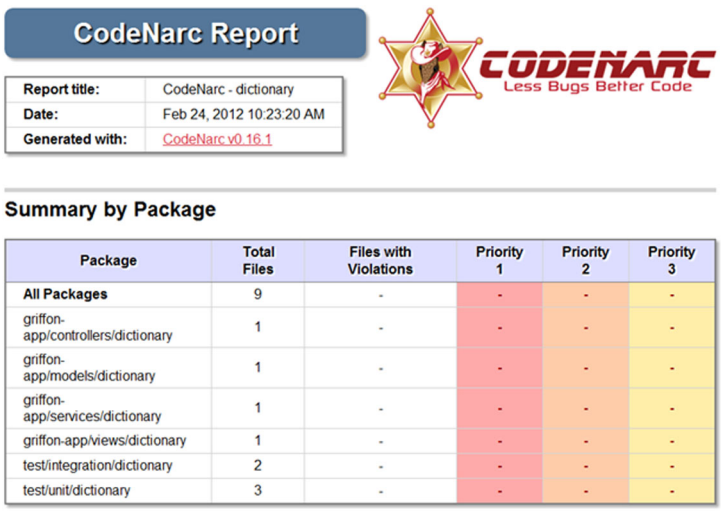
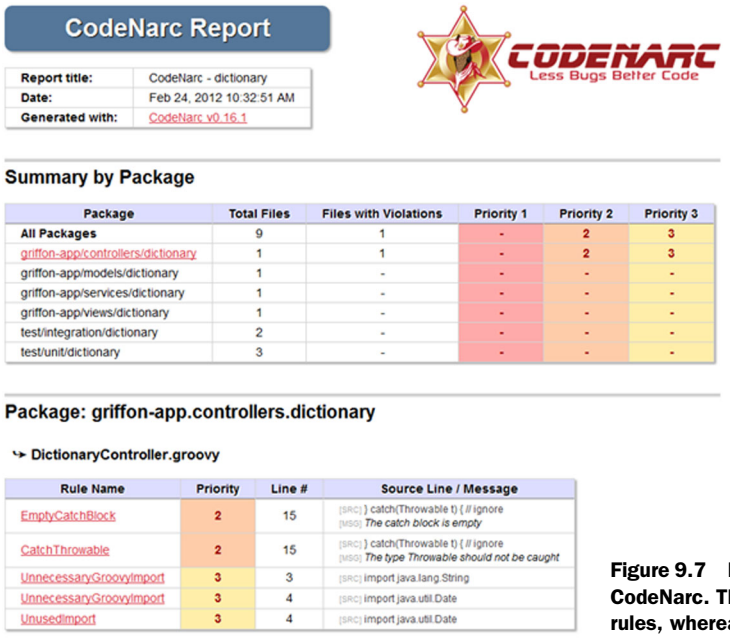


Figure 9.6 CodeNarc report for the Dictionary application. Everything appears to be in order.



CodeNarc Report

Report title:	CodeNarc - dictionary
Date:	Feb 24, 2012 10:32:51 AM
Generated with:	CodeNarc v0.16.1

CODENARC
Less Bugs Better Code

Summary by Package

Package	Total Files	Files with Violations	Priority 1	Priority 2	Priority 3
All Packages	9	1	-	2	3
griffon-app/controllers/dictionary	1	1	-	2	3
griffon-app/models/dictionary	1	-	-	-	-
griffon-app/services/dictionary	1	-	-	-	-
griffon-app/views/dictionary	1	-	-	-	-
test/integration/dictionary	2	-	-	-	-
test/unit/dictionary	3	-	-	-	-

Package: griffon-app.controllers.dictionary

DictionaryController.groovy

Rule Name	Priority	Line #	Source Line / Message
EmptyCatchBlock	2	15	[SRC] } catch(Throwable t) { // ignore [MSG] *The catch block is empty*
CatchThrowable	2	15	[SRC] } catch(Throwable t) { // ignore [MSG] *The type Throwable should not be caught*
UnnecessaryGroovyImport	3	3	[SRC] import java.lang.String
UnnecessaryGroovyImport	3	4	[SRC] import java.util.Date
UnusedImport	3	4	[SRC] import java.util.Date

Figure 9.7 Files in violation as found by CodeNarc. The controller fails three rules, whereas the view fails just one.

9.4.3 Measuring Groovy code complexity with GMetrics

The test bar may be green. The code may be healthy given a certain set of rules. But is it too complex? Is there a tight coupling between two classes that should be broken into a simpler form? These are the questions GMetrics (http://gmetrics.sourceforge.net) can answer for your Groovy code.

GMetrics is another of Chris Mair's projects. As a matter of fact, GMetrics' configuration closely mirrors CodeNarc's. You'll now install and run GMetrics:

```
$ griffon install-plugin gmetrics
$ griffon gmetrics
```

Those commands should install the plugin and run GMetrics with the default configuration.

GMetrics measures three things in your code:

- The number of lines per class
- The number of lines per method
- Cyclomatic complexity[3]

Figure 9.8 shows the report generated against the example codebase.

Well, well. Looks like the number of lines for classes and methods are well within a reasonable range. The cyclomatic complexity numbers, on the other hand, are a surprise. Scroll down further, and you'll see a different picture: the cyclomatic complexity

[3] http://en.wikipedia.org/wiki/Cyclomatic_complexity.

GMetrics Report

Report timestamp: Dec 5, 2011 10:32:42 AM

Metric Results

Package/Class/Method	Complexity (total)	Complexity (average)	Class Lines (total)	Class Lines (average)	Method Lines (total)	Method Lines (average)
[p] **All packages**	23	1.1	190	21.1	99	5.2
[p] griffon-app	6	1.5	61	15.3	8	2.7
[p] griffon-app/controllers	1	1.0	15	15.0	1	1.0
[p] griffon-app/controllers/dictionary	1	1.0	15	15.0	1	1.0
[c] dictionary.DictionaryController	1	1.0	15	15	1	1.0
[m] search	1	1	N/A	N/A	1	1
[p] griffon-app/models	1	1.0	10	10.0	3	3.0
[p] griffon-app/models/dictionary	1	1.0	10	10.0	3	3.0
[c] dictionary.DictionaryModel	1	1.0	10	10	3	3.0

Figure 9.8 GMetrics report on the Dictionary application. The class and method line numbers look OK, but the cyclomatic complexity is high.

of the test code is really high (16), whereas the cyclomatic complexity of the application is really low (6). The most complex class of the production code is `DictionaryService`. That was to be expected because it's the only one that has branching code. A high cyclomatic complexity number isn't too worrying at the moment. But keep an eye on those numbers; they'll help you spot potential problems in your test code.

There's one last tool we'd like to cover. It's always a good idea to know how much of the production code is exercised by your tests, especially when TDD isn't the driving methodology. When you follow TDD to the letter, you'll end up more often than not with 100% code coverage.

9.4.4 Code coverage with Cobertura

When it comes to code-coverage tools for the Java platform, Cobertura (http://cobertura.sourceforge.net) is the one many people think of. That might well be because of its open source nature. Other popular choices for instrumenting code and obtaining a code-coverage measure are EMMA (http://emma.sourceforge.net) and Atlassian's Clover (www.atlassian.com/software/clover). But when this chapter was written, the Cobertura-based plugin was the only one available. Be sure to visit the Griffon Plugins page (http://artifacts.griffon-framework.org/plugins) to see if a new coverage tool has been added as a plugin. You can install the Cobertura plugin with the following command:

```
$ griffon install-plugin code-coverage
```

Once the `code-coverage` plugin has been installed, you only need to run a set of tests with an additional command flag:

```
$ griffon test-app -coverage
```

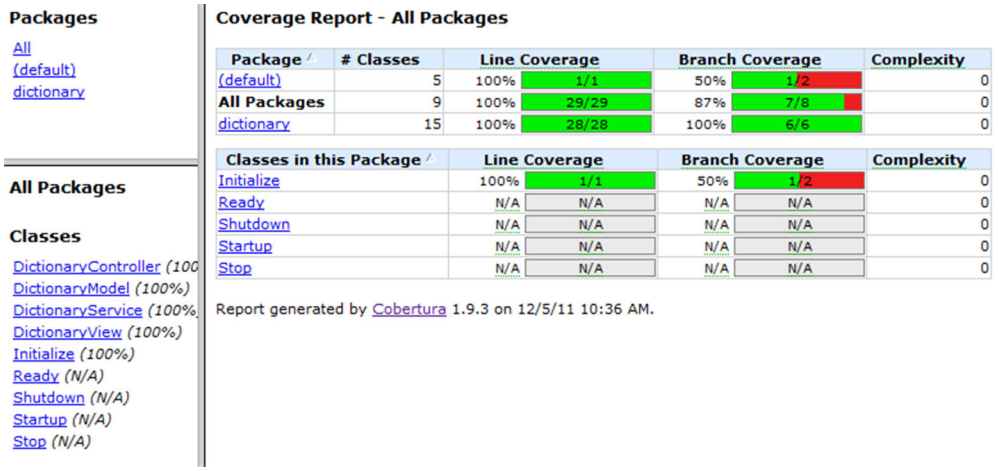


Figure 9.9 Code-coverage report with Cobertura

This command exercises all tests found in the Dictionary application with all production classes instrumented for code coverage. Figure 9.9 shows the coverage report obtained after running this command.

It appears that you have good code coverage: it's reported at 100%! But the report also shows that one of the classes/scripts isn't fully covered. The script in question is the Initialize script. Glancing at its content, you discover the truth. At the end of the file is this line:

```
SwingBuilder.lookAndFeel((isMacOSX ? 'system' : 'nimbus'), 'gtk', ['metal',
  [boldFonts: false]])
```

It includes a branching statement. Given that the condition is platform-specific, there's no way you can verify for certain that all lines of code and all branches have been covered. But you're close. You can leave it as it stands.

In case you're wondering about code instrumentation, it's basically a code transformation that inserts new bytecode that registers the execution of code paths. The Cobertura plugin is preconfigured to overwrite the original bytecode with instrumented code, and it also cleans up after itself. Pay close attention to the compiled code before you package and ship it. You don't want to inadvertently ship a version that contains instrumented code, do you?

It's been a wonderful journey through the guts of your application. Thanks to Griffon's testing facilities and its many test-related plugins, you can be certain that your application is in good shape.

9.5 Summary

Testing is one of those tasks that often get relegated to the end of the development cycle. Reasons vary, but typically it's because writing testing code isn't as fun as writing

production code. The Griffon framework recognizes this and tries its best to make writing and executing test code as fun as doing the same with production code.

Griffon provides a `test-app` command that comes loaded with a powerful and flexible set of options that allows you to run tests in pretty much any way you want. Test executions are separated into phases and types. Phases can cover many types, and types can span several phases. This is great when you want to run all Spock specifications regardless of their phase, or if you want to run all unit tests regardless of whether they're JUnit tests, Spock specifications, or easyb stories.

If writing testing code seems like an arduous task, writing UI testing code is often viewed as certain doom. FEST brings a refreshing experience to writing tests from the UI's perspective, thanks to its fluent interface design, generics-aware API, and powerful abstractions.

You've also learned that JUnit is no longer the only game in town. Spock provides a rich testing DSL by stretching the capabilities of the Groovy language. easyb veers to BDD, allowing a disparate team to communicate effectively and get results in their own terms.

Finally, you used CodeNarc, GMetrics, and Cobertura to probe and measure the health of an application.

Now that you know how to build a Griffon application and make sure it's working correctly, we can examine the last step in the application development cycle: shipping the application to your customers.

10 Ship it!

This chapter covers

- Packaging your application
- Packaging with the Installer plugin

After spending a good amount of time having fun building a Griffon application, suddenly you realize you must somehow deliver the application to your users. And building an application takes time, even if you use the Griffon framework—after all, the application doesn't write itself, does it? There are many options for packaging an application—which ones will be best suited for your needs? Memories of painful packaging experiences may start to flow…

The good news is that packaging an application is a task common to all applications, and Griffon provides packaging support out of the box. With Griffon, you get a few choices that should get you out of trouble quickly. But if you require a packaging option that packs more power, the solution is just one plugin install away.

In this chapter, we'll explore packaging options and packaging targets. We'll also walk you through using the Installer plugin.

10.1 Understanding the common packaging options

In chapter 2, we discussed using the Griffon configuration file, `BuildConfig.groovy`, to determine which options to use when building an application. Turns out some of those options control how some files are generated, particularly the applet and webstart support files. It should come as no surprise that those configuration options are reused when packaging the application for distribution.

Before we get into how you configure jar, zip, applet, or webstart options, let's look at the options shared by all Griffon packaging targets:

- The packaging target runs in the production environment by default. It's configured to sign and pack all jars. You can change these settings by editing `BuildConfig.groovy`.
- All packaging targets must build the application jar file, whose name is taken from the following configuration flag: `griffon.jars.jarName`. The value of this configuration flag can be determined by convention too. Have a look at the `application.properties` file locate at the root of the application. You'll notice there's one entry that specifies the application's name while there's a second one that spells out what's the latest version. In the case of the GroovyEdit application these entries look like the following ones:

```
app.name=GroovyEdit
app.version=0.1
```

Now, look at figure 10.1, which shows the layout of the configuration directory of the GroovyEdit application you first created in chapter 1.

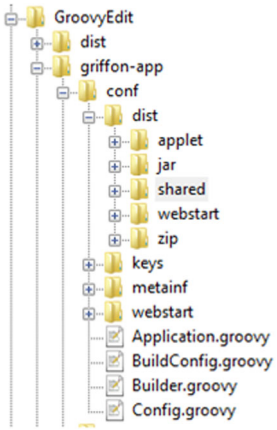


Figure 10.1 Standard directory structure of an application's configuration files

You may remember the keys and webstart directories, but dist and its subdirectories are new. Of particular importance is the shared directory (shown highlighted in figure 10.1). Each of the directories in griffon-app/conf/dist matches a packaging target that will be discussed shortly, except for shared. The responsibility of the shared directory is to hold files that are common across all packaging targets (including those exposed by the Installer plugin). This folder is a perfect place to put a README or LICENSE file, for example.

Now, as you've probably guessed, the other directories provide target-specific files. Executing the zip target (`griffon package zip`) results in all application files associated with that target being created. They're all the files from `griffon-app/conf/dist/zip` and from `griffon-app/conf/dist/shared`.

Based on the directory names in `griffon-app/conf/dist`, you can probably guess what the default packaging targets supported by Griffon are, and you'd be right. The packaging targets are jar, zip, applet, and webstart. We'll look at each of them next.

10.2 Using Griffon's standard packaging targets

The applet and webstart targets make perfect sense, given that an application can be run in both of those modes. It follows that you should be able to package applications for those modes. The jar and zip options take care of the standalone mode. You can package the application in a single jar or zip that contains a conventional structure, including platform-specific launcher scripts. The launcher scripts are used to start the application. So, as you'd expect, when building a zip distribution, there's a GroovyEdit .bat batch file for Windows OS environments and a `GroovyEdit` shell script for Linux and Mac OS X environments.

Packaging an application for a desired target is as easy as running the following command:

```
$ griffon package zip
```

You can even generate all default targets just by omitting a target. The following command generates all four default targets:

```
$ griffon package
```

Let's start by looking at the simplest packaging target, then move on to the more complex ones.

10.2.1 The jar target

Packaging your application in a single jar is perhaps the quickest way to let your users enjoy your brand new Griffon application. Many platforms are configured by default to launch the application when a user double-clicks the jar file. Figure 10.2 shows the files created for the GroovyEdit application after the following command has been executed:

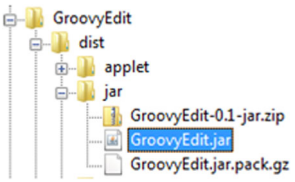


Figure 10.2 Directory outline of a jar distribution

```
$ griffon package jar
```

As an alternative, a user can invoke the following command to run the application:

```
$ java -jar GroovyEdit.jar
```

This particular packaging mode will expand the contents of all jar files required by the application and package them in a single jar file. There may be times when a few files may result in duplicate entries; for example, two of the application's dependencies may provide a file named META-INF/LICENSE.txt or build.properties or some other file entry that may be fairly common. The default configuration keeps the first file that was added to the main jar and ignores any subsequent duplicate files. But this might not work for some cases. For instance, it's common for classes that perform dynamic lookup of services to use a filename by convention, which contains the fully qualified class names of the services to be instantiated. If the duplicate file is skipped, there's a high chance that the application won't work correctly, because at least a

portion of the services won't be initialized! This happens because the service definitions were discarded when duplicate entries were skipped. Clearly you need an alternative to just keeping the first file that appeared in the classpath. Fortunately, there's one solution to this problem: configuring a merging strategy per matching entry.

The Griffon build includes a mechanism that lets you specify a merging strategy for a particular file or path, using a pair of values: a regular expression that specifies the path, and a class name that identifies the strategy to use. Table 10.1 describes the currently available merging strategies and their behavior.

Table 10.1 Merging strategies that can be applied to file paths when building an application in a single jar

Strategy	Description
`Skip`	Avoids any processing. Essentially keeps the previous file untouched and discards the duplicate. This is the default behavior.
`Replace`	Discards the previous file and keeps the newest.
`Append`	Appends the contents of the new file at the end of the previous file.
`Merge`	Common lines found in the new file are discarded. New lines in new file are appended at the end of the previous file.
`MergeManifest`	Manifest keys are merged by overriding the values of common keys and adding missing keys found in the new file.
`MergeProperties`	Like MergeManifest but works with properties files.
`MergeGriffonArtifacts`	This special merging strategy uses a specific file that defines the types and names of the artifacts that should be loaded at runtime.

Now that you know about these merging strategies, how do you put them to good use? These settings only apply when building the application jar so surely you'll need to update `BuildConfig.groovy` somehow. This is exactly what you're going to do.

Imagine for a moment that you know that two distinct XML files with the same names and paths are included somewhere in the classpath; each one is provided by a different jar file. It follows that they'll collide when building a single jar for the application. For some reason you've decided that you want to retain the last file encountered in the classpath (which is the opposite of the default convention). This means you must configure a merging strategy for it. Open `BuildConfig.groovy`, and locate the `griffon.jars` block. Update the text with the following snippet:

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

The `merge` block is actually a Map, where the keys are regular expressions and the values are class instances—that's why the `Replace` strategy uses the fully qualified class name, but you can use an `import` statement to shorten it up a bit. This particular setting looks for files named my.xml placed at the root of the application's jar file. If a match is found, the newest file will be kept and the older one discarded. If no matches are found, nothing special happens for that path.

The Griffon build system provides a default set of merging mappings, because some paths are fairly common and their merging strategy is pretty consistent from application to application. Table 10.2 lists those mappings.

Table 10.2 Commonly used paths and their default merging strategies

Path	Strategy
/META-INF/griffon-artifacts.properties	`MergeGriffonArtifacts`
/META-INF/MANIFEST.MF	`MergeManifest`
/META-INF/services/.*	`Merge`
.*.properties	`MergeProperties`

There are several things worth noting about merges:

- Your settings will override any defaults provided by the system.
- Paths should be defined from the most specific to the least.
- `griffon.jars.jarName` takes care of the application's jar filename.

There are no additional settings you can configure for the jar packaging target.

10.2.2 The zip target

The zip target generates a distribution directory commonly found in Linux-like systems, but it isn't restricted to such platforms. It includes platform-specific launcher scripts for Windows, and the Linux launcher can be used in Mac OS X too.

Figure 10.3 shows the directory structure of a zip distribution for the GroovyEdit application you developed back in chapter 1.

In short, the bin directory contains the application launchers; the lib directory contains all application jar files. The zip packaging target will also create a zip file with all of these files and directories, as you can see in figure 10.3 (the file is GroovyEdit-0.1.zip).

Next, we'll cover both the applet and webstart targets, because they share a lot of options.

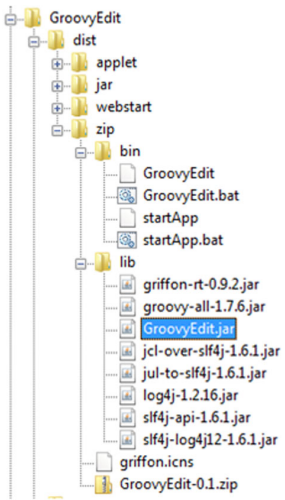


Figure 10.3 Directory outline of a zip distribution

10.2.3 The applet and webstart targets

If you edit the file `BuildConfig.groovy`, you'll find a configuration section pertaining to the webstart and applet targets, as shown here:

```
griffon {
  . . .
  webstart {
     codebase = "${new File(griffon.jars.destDir).toURI().toASCIIString()}"
     jnlp = 'application.jnlp'
  }
  applet {
     jnlp = 'applet.jnlp'
     html = 'applet.html'
  }
}
```

As you can see, you configure the location of the JNLP file as well as the `codebase` property of the webstart target. For the applet target, you configure the location of the JNLP and HTML files used to launch the applet.

To override the `codebase` property at the command line, execute the following command:

```
$ griffon package webstart -codebase=http://path/to/your/codebase
```

As you can see, there isn't much magic going on when you use Griffon's standard packaging options, just a clever directory naming convention and some configuration flags available for the central configuration file. It's worth noticing that plugins (discussed in chapter 11) can also provide packaging artifacts; just follow the same directory naming conventions.

Before we move on to additional packaging targets found outside of the default set, we should look at a common option available to all packaging targets, including those we'll discuss later in this chapter. This feature is related to the manifest file that every jar file must contain, including the application's jar file.

10.2.4 Customizing the manifest

You might not know this, but jar files are pretty much the same thing as zip compressed files. The difference is that jar files require a few additional files that provide some metadata. This is the role of the manifest.mf file. If you're curious, you might have taken a peek at a jar file and found the following entry: `META-INF/MANIFEST.MF`.

A manifest file is a collection of key/value pairs in plain text. In it, you'll usually find information about the tool that was used to create the jar file, as well as the version of the JVM and maybe the license and the file's creator.

Griffon is aware of these common settings and will gladly generate a manifest file that contains more information about the application you just packaged. Here, for example, is the default manifest created for the GroovyEdit application:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.1
```

```
Created-By: 19.1-b02-334 (Apple Inc.)
Main-Class: griffon.swing.SwingApplication
Built-By: aalmiray
Build-Date: 21-06-2011 19:02:28
Griffon-Version: 0.9.3
Implementation-Title: GroovyEdit
Implementation-Version: 0.1
Implementation-Vendor: GroovyEdit
```

As you might have guessed, all this information was gathered from the application itself. And although this is fine and dandy, what happens if you require additional values in the manifest? What if you want to change the value of the `Built-By` key? Well, you're in luck, because these tasks can be performed by updating the build configuration file. That's right, you're back to editing `BuildConfig.groovy`.

Remember the `griffon.jars` block we described a few sections ago? We'll touch on that block once more by adding another configuration option. The following piece of code demonstrates how you can set a new key as well as override the value of a predefined key:

```
griffon {
  jars {
    manifest = [
       'SomeKey': 'Some Value',
       'Built-By': 'The Awesome Team'
    ]
  }
}
```

The manifest should have different values now. It should look something like this:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.1
Created-By: 19.1-b02-334 (Apple Inc.)
Main-Class: griffon.swing.SwingApplication
Built-By: The Awesome Team
Build-Date: 21-06-2011 19:10:31
Griffon-Version: 0.9.3
Implementation-Title: GroovyEdit
Implementation-Version: 0.1
Implementation-Vendor: GroovyEdit
SomeKey: Some Value
```

Notice the appearance of a new key (`SomeKey`) and that `Built-By` has a different value.

The final aspect we'll cover for both applet and webstart packaging targets is the templates used to generate the JNLP and HTML files required to launch the application in the appropriate mode.

10.2.5 Customizing the templates

Earlier we described the contents of `griffon-app/conf/dist`. You may have noticed that there's another directory that appears to be a duplicate entry: `griffon-app/conf/webstart`. This directory is not, in fact, a duplicate but a common location for

both applet and webstart packaging targets. It contains the basic templates and icons that will be used when packaging the application in the desired deployment mode.

You can alter any of these files at will; just be careful with the variables surrounded by `@` characters, because they will be used by the build system to store the values gathered by inspecting the application's conventions. Here are a few of them, in case you're curious:

```
<title>@griffon.application.title@</title>
<vendor>@griffon.application.vendor@</vendor>
<homepage href="@griffon.application.homepage@"/>
```

You can instruct the build system to apply a different value even if you don't change the templates themselves. Take another look at `BuildConfig.groovy`. Right at the end of the file you'll see a block of code that looks similar to the following:

```
deploy {
    application {
        title = "${appName} ${appVersion}"
        vendor = System.properties['user.name']
        homepage = "http://localhost/${appName}"
        description {
            complete = "${appName} ${appVersion}"
            oneline  = "${appName} ${appVersion}"
            minimal  = "${appName} ${appVersion}"
            tooltip  = "${appName} ${appVersion}"
        }
        icon {
            'default' {
                name   = 'griffon-icon-64x64.png'
                width  = '64'
                height = '64'
            }
            splash {
                name   = 'griffon.png'
                width  = '391'
                height = '123'
            }
            selected {
                name   = 'griffon-icon-64x64.png'
                width  = '64'
                height = '64'
            }
            disabled {
                name   = 'griffon-icon-64x64.png'
                width  = '64'
                height = '64'
            }
            rollover {
                name   = 'griffon-icon-64x64.png'
                width  = '64'
                height = '64'
            }
            shortcut {
```

```
                name   = 'griffon-icon-64x64.png'
                width  = '64'
                height = '64'
            }
        }
    }
}
```

Feel free to make some changes and package the application in either applet or webstart mode. You should see a difference in the generated files reflecting your modifications. The values for `${appName}` and `${appVersion}` are directly taken from the application.properties file.

Now that you know how to use Griffon's standard packaging commands to package an application, let's look at the Installer plugin. The Installer plugin gives you a few more options and the ability to create installers and launchers for multiple platforms.

10.3 Using the Installer plugin

Every Java developer knows the Java mantra by heart: compile once, run anywhere. But *package* once install anywhere doesn't work, especially when you want your users to have a great user experience when installing your application. Installing the application must appear native to the operating system your user is using.

You can choose from several packaging options for each platform, and some of them have been rounded up in a handy plugin called Installer. In this section, you'll install the Installer plugin, build a distribution, and take a quick look at several installer targets: izpack, rpm, deb, mac, jsmooth, and windows.

Installing the plugin is done the same way you install other plugins. The Installer plugin can be installed by invoking the following command:

```
$ griffon install-plugin installer
```

Upon installing it, you'll notice that this plugin provides plenty of new scripts—a pair of scripts for each packaging target. At the time of writing, the targets are as listed in table 10.3.

Table 10.3 Installer plugin targets

Target	Description
izpack	A universal installer using the IzPack project (http://izpack.org) and batch/shell scripts for launching
rpm	Creates a package in RPM format for Linux distributions
deb	Creates Debian-based packages
mac	Creates a Mac OS X application bundle plus a DMG
jsmooth	A Windows launcher based on JSmooth (http://jsmooth.sourceforge.net) and zip files
windows	An alias for the jsmooth target

How complicated is it to package an application using the Installer plugin? Executing the packaging target isn't complicated at all. The Installer plugin does a great job and saves you a lot of work and headaches. Let's take a closer look at using it.

10.3.1 Building a distribution

Each packaging target has a prepare script and a create script. The first step in building a distribution is to run the prepare script. This script copies any templates that may be required by a particular distribution, placing them in `$projectTargetDir/installer/<target>`. You can make as many changes to those templates as you choose. Once you're comfortable with those changes, it's time to proceed to the next step.

TIP If you aren't sure where the installer files are located, look at the output of the `package` command. It has the exact path to the location of the files.

The second step is running the create script. This script is the real workhorse for each packaging target. When it completes, the appropriate distribution will have been built and placed in the standard distributions directory (dist).

Figure 10.4 shows the result of executing some of the packaging targets in the GroovyEdit application.

Each packaging script fires convenient build events that you can hook into, making the automation of hooking customized templates on your application a trivial task. It also means you can invoke `create-*` right after `prepare-*`. These build events have a single argument—the type of the package being built. Here's a list of the events fired:

- `PreparePackage<type>Start`, `PreparePackage<type>End`—Triggered by the prepare-* scripts
- `CreatePackage<type>Start`, `CreatePackage<type>End`—Triggered by the create-* scripts

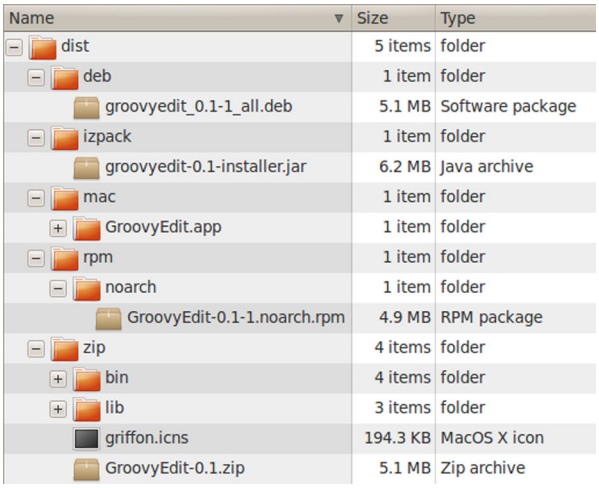


Figure 10.4 The outcome of running deb, izpack, mac, rpm, and zip packaging targets on GroovyEdit

As an alternative to invoking two commands, `prepare` and `create`, to build a distribution, you can append the packaging target to the `griffon package` command. This means you can package an application with an IzPack-based installer by typing the following at your command prompt:

```
$ griffon package izpack
```

Remember, too, that these additional packaging targets will automatically include all shared and specific distribution files available at `griffon-app/conf/dist`.

Let's review the configuration options for each packaging target.

10.3.2 The izpack target

An IzPack installer provides a wizard-like experience when installing an application. You can configure how many steps are used, how each step page looks, and even hook into specific platform settings. As a result, an IzPack-based installer is perhaps the most versatile option at your disposal, given that it produces a Java-based installer that can be run on any platform. There are plenty of configuration options for IzPack, and we encourage you to visit the official IzPack documentation site to learn more about them (http://izpack.org/documentation).

Figure 10.5 An IzPack installer for the GroovyEdit application. The default installer template runs eight steps.

Figure 10.5 shows the first page of the IzPack installer created for the GroovyEdit application using the default values provided by the templates. Remember that you can either double-click the generated installer jar or execute the following command at your console prompt:

```
$ java -jar <izpack_installer>.jar
```

The heart of this installer is located at `$projectTargetDir/installer/izpack/resources/installer.xml`. Every referenced resource is relative to that file's location. You'll want to customize this file in order to add or remove steps.

Griffon has several pieces of information (application name, version, author, and so on) that could be valuable when configuring the installer. Rather than having to duplicate this information in the install source files, Griffon has created build variables to represent this information. The variables are independent from IzPack, and Griffon will resolve them when the installer is created:

- `@app.name@`—The name of the application as it's found in the `application.properties` metadata file
- `@app.version@`—The current application version, also from the application's metadata

- `@app.author@`—The name of the application's author; defaults to "Griffon"
- `@app.author.email@`—The author's contact details; defaults to user@griffon.codehaus.org
- `@app.url@`—The application's website; defaults to http://griffon.codehaus.org

Now let's investigate platform-specific installers.

10.3.3 The rpm target

RPM stands for Red Hat Package Manager. Originally developed by Red Hat in the mid 90's, this packaging option is commonly used in several Linux flavors. RPM uses a special file, called the *spec*, to drive the creation of a package. You'll find plenty of documentation for the RPM spec file format by doing an internet search. The http://rpm.org site is a good place to start.

The following default variable placeholders are used in the spec file, which you'll find in `$projectTargetDir/installer/rpm/SPECS`:

- `@app.name@`—The name of the application as found in the `application.properties` metadata file
- `@app.version@`—The current application version, also from the application's metadata
- `@app.license@`—The license of your application; defaults to "unknown"
- `@app.summary@`—A brief description of the application; defaults to the application's name
- `@app.description@`—A detailed description of the application's features; defaults to "unknown"
- `@app.url@`—The application's website; defaults to "unknown"

The following is the output of querying the generated RPM package for GroovyEdit:

```
rpm -qpil dist/rpm/noarch/GroovyEdit-0.1-1.noarch.rpm
Name        : GroovyEdit

Version     : 0.1
Release     : 1
Install Date: (not installed)
Group       : Applications/GroovyEdit
Size        : 5817059
Signature   : (none)
Packager    : Andres Almiray
URL         : unknown
Summary     : GroovyEdit
Description : unknown
Relocations : (not relocatable)
Vendor      : (none)
Build Date  : Mon 24 May 2010 03:30:21 AM PDT
Build Host  : aalmiray
Source RPM  : GroovyEdit-0.1-1.src.rpm
License     : unknown
// file list omitted for brevity
```

RPM isn't the only option for Linux-based installers. The next section describes another popular one.

10.3.4 The deb target

Debian-based installers have been around as long as RPMs. There are several Debian-based distributions out there; perhaps the best known is Ubuntu, due to its ease of use and attention to detail regarding the overall user experience.

Building a Debian package is much like building with an RPM or IzPack installer. You just need to tweak the default settings found in `$projectTargetDir/installer/deb/resources/deb_settings.properties`.

The variable placeholders are pretty much the same ones as before, plus a few Debian-specific ones. These are all of them, for the sake of completeness:

- `@app.name@`—The name of the application as found in the `application.properties` metadata file
- `@app.version@`—The current application version, also from the application's metadata
- `@app.author@`—The name of the application's author; defaults to "Griffon"
- `@app.author.email@`—The author's contact details; defaults to user@griffon.codehaus.org
- `@app.synopsis@`—A brief description of the application; defaults to the application's name
- `@app.description@`—A detailed description of the application's features; defaults to "unknown"
- `@app.depends@`—The dependencies on other packages; defaults to "sun-java5-jre | sun-java6-jre"

The following output is obtained by querying the generated .deb file for GroovyEdit:

```
new debian package, version 2.0.
 size 5346426 bytes: control archive= 737 bytes.
     236 bytes,    10 lines      control
    1025 bytes,    13 lines      md5sums

 Package: groovyedit
 Version: 0.1-1
 Section: contrib/misc
 Priority: extra
 Architecture: all
 Depends: sun-java5-jre | sun-java6-jre
 Installed-Size: 5839
 Maintainer: Griffon <user@griffon.codehaus.org>
 Description: groovyedit-0.1
  Unknown
```

On to the next platform: Mac OS X.

10.3.5 The mac target

The Apple-branded UNIX platform, Mac OS X, has become a popular choice for developing applications. Love it or hate it, the fact is that Mac OS X's first directive is user experience. They make working with applications so easy and intuitive that you don't need a manual most of the time. Installing an application is also a simple click-and-drag operation.

Applications in Mac OS X are installed via application bundles, which are nothing more than a conventional directory structure—sound familiar? Application bundles are also often distributed using a DMG file. The mac install target can generate both types of archives, but DMG files can only be generated if you execute it when running on Mac OS X.

This time there are no variable placeholders that you can tweak, but you can customize the application's icon, by placing an .icns file that matches the application name under `griffon-app/conf/dist/mac`. This means that if you were to distribute a Mac-based installer for GroovyEdit, and you wished to include a custom icon for it, you'd have to name it `griffon-app/conf/dist/mac/GroovyEdit.icns`. What happens if no custom icon file is defined? The application bundle will use the default griffon.icns file.

10.3.6 The jsmooth target

JSmooth is an executable wrapper around Java. This packaging target will generate an .exe file that can be used to run the application on Windows. There are several options you can configure; perhaps the most interesting one allows the wrapper to embed a particular version of a JRE, which means your application can run in self-contained mode if a suitable JRE isn't installed on the target computer.

JSmooth uses a template file similar to the other packaging targets. This template includes variable placeholders, but there's no need to change them because the create-jsmooth script takes care of updating them with the appropriate values. As a matter of fact, it's recommended that you don't manually substitute any of the variable placeholders you'll find in the template.

Refer to the JSmooth manual to learn more about the options available in the template (http://jsmooth.sourceforge.net/docs/jsmooth-doc.html).

10.3.7 The windows target

The windows packaging target is just an alias for the jsmooth target; there's nothing new to add here. Perhaps in the future a different Windows-based option that doesn't rely on JSmooth might be added to the Installer plugin.

10.3.8 Tweaking a distribution

Let's recap for a moment. The Griffon build system provides four default packaging targets (jar, zip, applet, and webstart), which should cover your basic needs for deploying an application. There's also the Installer plugin, which provides additional

packaging targets (izpack, rpm, deb, dmg, and jsmooth). These additional packaging targets are fully integrated with the `package` command.

The conventions laid out by these packaging targets are good enough for most cases. But if you need to tweak the configuration, what do you do? You can take advantage of the following aspects of the build system:

- Every build operation is encapsulated by a Gant target.
- Start and end events for each invoked target are automatically triggered.
- Event handlers can be built that react to those triggers.

Armed with this knowledge, you only need to know the name of the event you want to handle and the location of the configuration files you need to tweak.

Say, for example, you want to change the configuration for the izpack packaging target. As we said before, this target requires an installer descriptor file, normally named installer.xml. This file is usually generated automatically by the plugin and is placed at `${projectWorkDir}/installer/izpack/resources`. You must provide different contents for this file, which can be done easily by overwriting the file at the right time, say at the end of the prepare phase but before the create script is called. That's the first requirement covered: knowing what to change and when. Now you need to find a way to make the change happen.

The second step is writing an event handler. In chapter 8, we discussed build events and scripts—that's precisely what you'll use here. The build system will honor all your build event handlers, as long as you place them in a file named `_Events.groovy` and place it inside the scripts directory. You can create such a file and fill it with the contents of the following listing.

Listing 10.1 Build event handler to override izpack packaging target settings

```
eventPreparePackageIzpackEnd = {
  ant.copy(todir: "${projectWorkDir}/installer/izpack/resources", overwrite:
     true) {
    fileset(dir: "${basedir}/src/installer/izpack/resources", includes:
"**")
  }

  ant.replace(dir: "${projectWorkDir}/installer/izpack/resources") {
    replacefilter(token: "@app.name@", value: griffonAppName)
    replacefilter(token: "@app.version@", value: griffonAppVersion)
  }
}
```

First you define the event handler according to the convention. You might remember it from section 10.3.1. Next, you overwrite the files created by the prepare phase with your own. This script assumes that the new installer files are located relative to the application's root directory; you can pick a different location if required. Finally, you replace any tokens found in the freshly copied files with their appropriate values; that way you can keep the files parameterized. You can also define your own tokens and values.

This is a trivial example—you're only copying static files and replacing some tokens. A more elaborate tweak could involve generating files on the fly, for example. There's no limit to what you can do at the build level. Remember, you have the full power of the Groovy language combined with the Gant target mechanism and all of the Griffon libraries.

10.4 Summary

Packaging an application is one of those common tasks that an application framework should help you get done consistently and quickly. Griffon supports several packaging targets out of the box, depending on the target environment on which you intend to distribute the application.

The jar and zip targets are usually the preferred methods of distributing an application for standalone consumption, whereas applet and webstart are used for their respective environments.

When the standard packaging targets prove to be inadequate for your application's needs, you can rely on the Installer plugin, which provides more targets that can be hooked into the `package` command. Currently supported targets include izpack, rpm, deb, mac, jsmooth, and windows. Additional targets may be added in future releases of the Installer plugin.

At this point, you've seen how to create a Griffon application from end to end. We've covered packaging—what could be next? You've used them, and now it's time to learn how to create one: plugins.

11 Working with plugins

This chapter covers

- Working with plugins
- Creating your own plugins

We've covered a lot of ground discussing what Griffon can do for you, but there's a limit to what it can do alone. After all, it can't predict what a future app will need—sometimes Griffon requires a helping hand. This is where plugins fit in.

You've seen plugins being put to work before. Recall from chapter 7 that SwingXBuilder provides additional threading facilities—this builder can easily be added to your application by installing a plugin. In chapter 9 you learned about FEST and easyb, two testing frameworks that can be configured to work with any Griffon application. Using these tools is as simple as installing the appropriate Griffon plugin. And in the last chapter, you learned how platform-specific installers can be created, also via a plugin.

In this chapter, we'll look at how plugins work and how to make one. First, let's find out how plugins can be located, installed, and uninstalled.

11.1 Working with plugins

As with all the recurring tasks you can perform with Griffon, there are a few command targets exposed by the `griffon` command that are available to you when working with plugins.

In this section, we'll briefly cover each plugin-related target.

11.1.1 Getting a list of available plugins

You may want to know which plugins are already available before you decide to create one. Chances are there's already a plugin that does what you need.

Invoking the `list-plugins` command target at your command prompt displays a table of all plugins in the central plugin repository (located at http://artifacts.griffon-framework.org). Be mindful that this command will attempt to establish a network connection to a remote server, so in some cases a corporate firewall may prevent you from reaching the server. If this happens, make sure you configure an HTTP proxy before launching the `list-plugin` command. Griffon includes a set of commands that deal with HTTP proxies, most specifically `add-proxy` and `set-proxy`. Figure 11.1 shows the first plugins in that list.

Another way to view available plugins is to point your browser to the following address: http://artifacts.griffon-framework.org/plugins. That page provides additional information about each plugin, such as compatible platforms and UI toolkits. The next command target provides more information about a specific plugin.

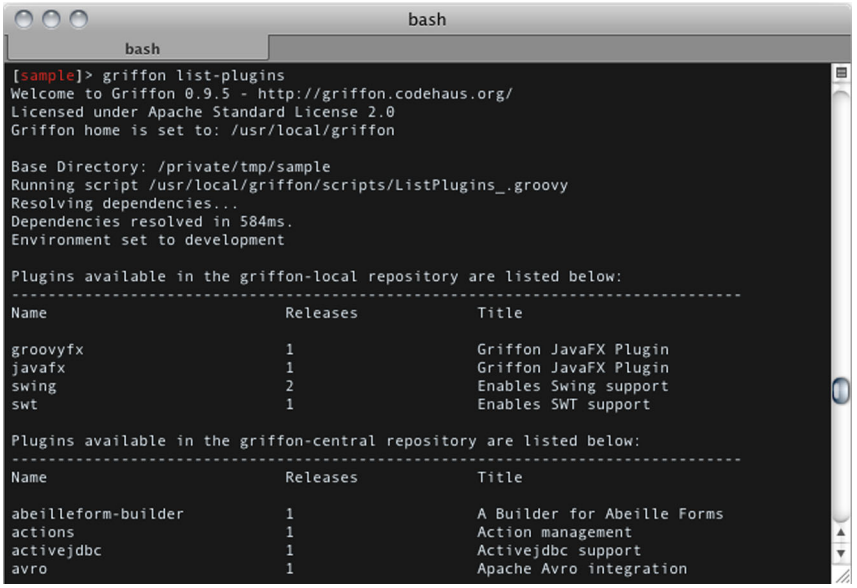


Figure 11.1 A list of available plugins, with their names, versions, and short descriptions

11.1.2 Getting plugin-specific information

Now that you know which plugins are available, you may want to know more about a particular plugin without installing it. This can be accomplished by calling the `plugin-info` command target with the plugin's name as parameter. Figure 11.2 shows what you get when querying the Spring plugin.

Invoking the command yields the author's name and email address, which may be useful should you get stuck using the plugin. You also get a link to the plugin's documentation page and additional information on UI toolkit and platform compatibility. The last line will list all available releases in the repository, should you choose to install a version other than the latest one.

Speaking about installing, that's our next command target.

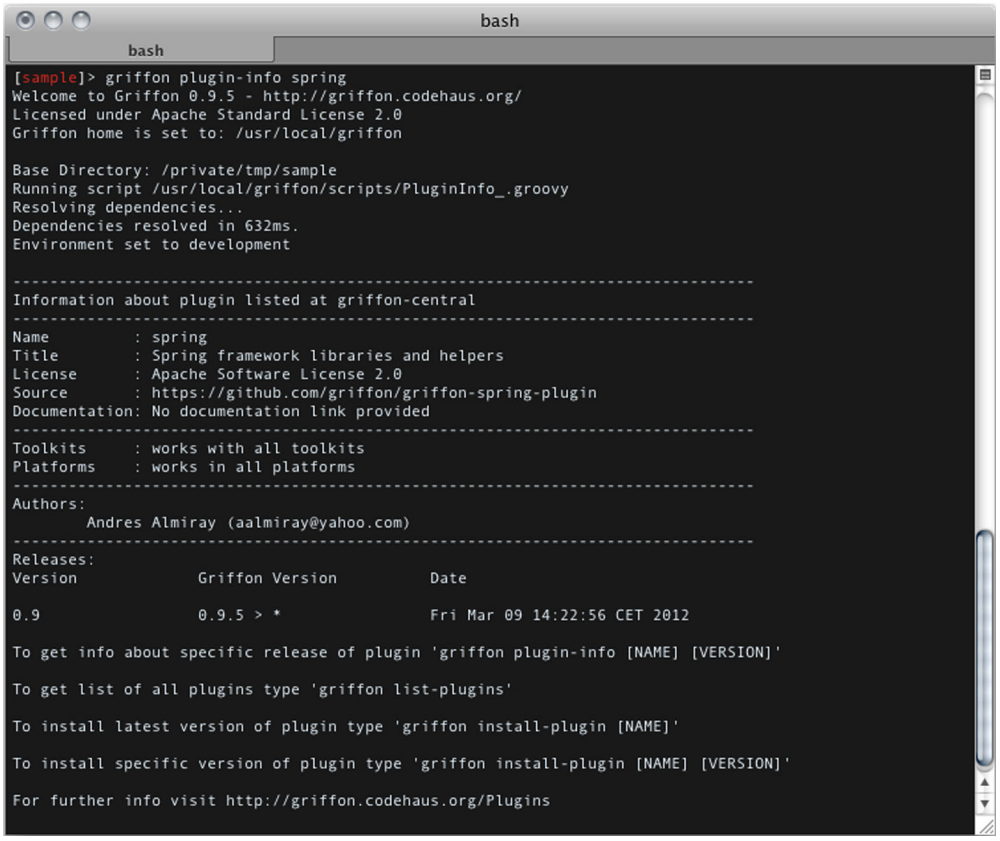


Figure 11.2 Information on the Spring plugin. Notice that this plugin works with every UI toolkit and on every platform.

11.1.3 Installing a plugin

Installing a plugin is quite easy. You've done it a few times already. The most typical method is the following:

```
$ griffon install-plugin gsql
```

This will install the latest version of the GSQL plugin (http://artifacts.griffon-framework.org/plugin/gsql).

Alternatively, you can install a specific version if the latest one doesn't suit your needs. You just need to add a version number as a parameter:

```
$ griffon install-plugin gsql 0.8
```

Remember that you can list all available releases of a plugin by invoking `plugin-info`. What about unreleased plugins? Let's say a friend of yours is developing a new plugin, and it hasn't been released yet and you'd like to try it out. Or maybe you're doing plugin development and want to use your unreleased plugin in a project. Is it possible to install the plugin?

Yes! You can install a plugin if you have access to the zip file that contains the plugin. You need to specify the full path to the zip file as a parameter, instead of the plugin name, like this:

```
$ griffon install-plugin /scratch/dev/custom/griffon-custom-0.1.zip
```

No matter how you install the plugin, you'll see output similar to what's shown in figure 11.3.

Installing a plugin also installs all of its dependencies. In this case, the Clojure plugin depends on the LangBridge plugin. Both plugins provide new scripts that become available through the `griffon` command. You saw this in chapter 8 when we looked at build-time events. Notice also that the Clojure plugin creates a new directory when installed (the directory name is src/clojure). You'll see how that works when you create your first plugin, just a few sections ahead.

Installing a plugin isn't just a matter of downloading a zip file and expanding its contents in a specific directory—the application's metadata must be updated too. For example, installing the Clojure plugin in an application named sample would result in these entries in the sample application metadata file (application.properties):

```
#Griffon Metadata file
#Sun Mar 11 18:20:06 CET 2012
app.griffon.version=0.9.5
app.name=sample
app.toolkit=swing
app.version=0.1
archetype.default=0.9.5
plugins.clojure=0.9
plugins.lang-bridge=0.5
plugins.swing=0.9.5
```

Next, if you have the option to install a plugin, it makes sense to have an option to remove it as well.

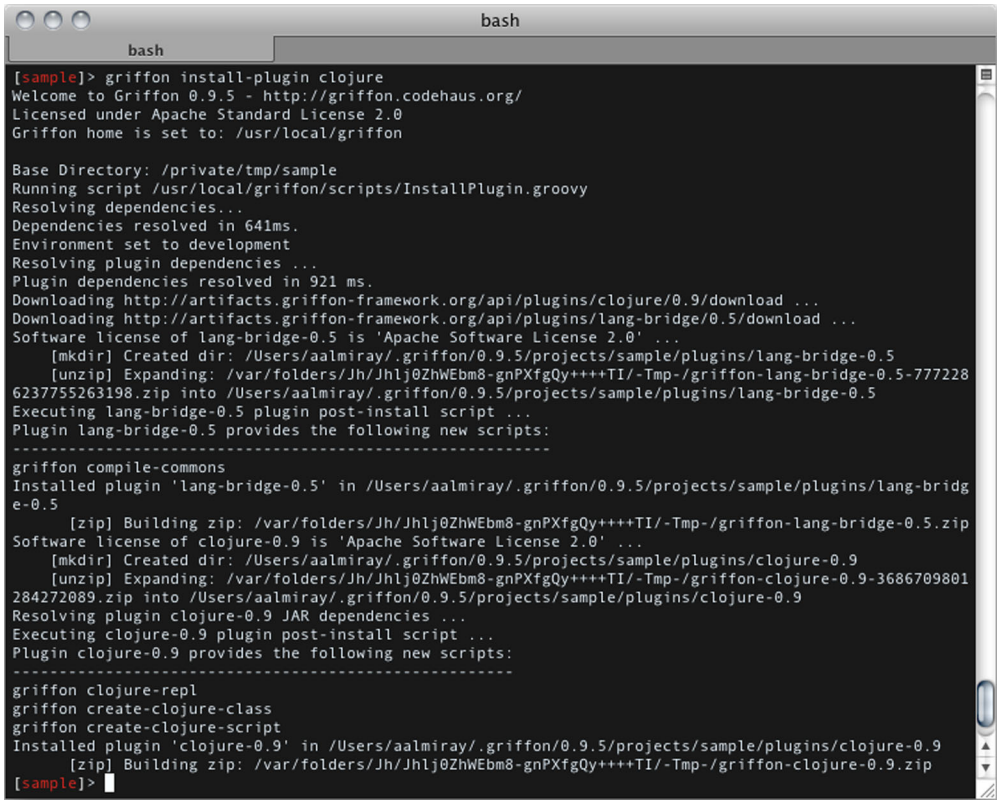


Figure 11.3 Installing the Clojure plugin. Notice that this plugin provides new scripts.

11.1.4 Uninstalling a plugin

To remove a plugin, call the `uninstall-plugin` command target. To uninstall the Clojure plugin, type

```
$ griffon uninstall-plugin clojure
```

This target won't just remove the plugin files from their install directory; it will also update the application's metadata properties and trigger uninstall events.

Now that you know how to work with plugins, we'll look at the two types of plugins you can create.

11.2 Understanding plugin types

We've said that plugins extend what Griffon as a framework can do. They can also extend what an application can do. There are two types of plugins: build time and runtime plugins. Build-time plugins are perhaps the easiest to create, but runtime plugins give you more bang for your buck. We'll cover both in this section, starting with build-time plugins.

11.2.1 Build-time plugins

Build-time plugins are usually seen as framework extensions. They extend what Griffon can do, but they aren't available while the application is running. This is a departure from the Grails plugin system (on which Griffon's is based) because every plugin in Grails is available both at build time and runtime.

These are the main responsibilities of a build-time plugin:

- *Make additional libraries available at runtime*—This is what the SwingXBuilder plugin does.
- *Provide additional scripts and build-time libraries*—The FEST and easyb plugins do this.
- *Deliver an addon*—Addons are, as you'll see later in the chapter, runtime plugins.

Of course, you can mix these responsibilities. For example, the Clojure plugin provides new scripts and delivers an addon that allows an application to load Clojure code at any time while running.

CREATING A PLUGIN

You create plugins much the same way as you create applications—by invoking a specialized command target:

```
$ griffon create-plugin foo
```

This will create a plugin named foo, whose structure is shown in figure 11.4.

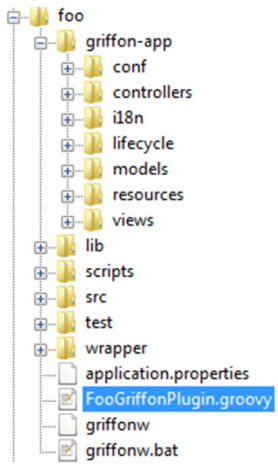


Figure 11.4 The directory structure of a plugin. Notice the resemblance to an application's structure. The plugin descriptor is highlighted.

Go ahead and open the plugin descriptor (`FooGriffonPlugin.groovy`) in your favorite editor. The following listing contains a summarized version of its contents (comments and reminder text have been omitted).

Listing 11.1 Initial contents of a plugin descriptor

```
class FooGriffonPlugin {
    String version = '0.1'
    String griffonVersion = '0.9.5 > *'
    Map dependsOn = [:]
    List pluginIncludes = []
    List toolkits = []
    List platforms = []
    String license = '<UNKNOWN>'
    String documentation = ''
    String source = ''
    List authors = [
        [
            name: 'Your Name',
            email: 'your@email.com'
        ]
    ]
    String title = 'Plugin summary/headline'
```

1 Plugin versions and dependencies

2 Plugin information

```
    String description = '''
Brief description of Foo.

Usage
----
Lorem ipsum

Configuration
-------------
Lorem ipsum
'''
}
```

❸ Plugin documentation

Every plugin must define a `version` number ❶. Although the suggested value is a number, you can change the value to be alphanumeric, like `0.2-BETA`, or make it a full normal string, like `bombastic`. Griffon expects the numbering scheme to follow the major.minor.patch pattern but doesn't enforce it. As a matter of fact, a popular convention during development is to append `-SNAPSHOT` to your plugin version, which clearly states that the plugin is currently under development and is not yet ready to be released.

The next property, `griffonVersion`, informs the Griffon build with which Griffon versions the plugin can work. It can be set as a simple number or as a range, like this:

```
lowerBound > upperBound
```

You can read the previous range as "works with any version from `lowerBound` up to `upperBound`, inclusive." You can use a wildcard (`*`) for either `lowerBound` or `upperBound`, but not both at the same time. For example, the following range

```
* > 1.0
```

means "any version up to 1.0," whereas the next one

```
1.0 > *
```

means "from 1.0 upwards, including the latest version."

The last property on the first block, `dependsOn`, defines which plugins are marked as dependencies. Dependencies are declared in a map by name and version number. The version number may again be a simple number, an alphanumeric string, or even a version range. The following snippet states a dependency on the transitions and rest plugins:

```
Map dependsOn = [transitions: '0.1.3', rest: 0.2]
```

Starting from version 0.3, Griffon supports several UI toolkits, not just Java Swing, as well as platform-specific libraries. We won't get into details right now, but you should know that a plugin can state which platforms it can run on (using the `platforms` property), and which toolkits it's compatible with (using the `toolkits` property).

The plugin information properties ❷ are the plugin's first line of documentation. The information you place there will be displayed by both the `list-plugins` and `plugin-info` command targets, so be sure to write what you think will be useful to you

and to others who might use your plugin. Also, be mindful about the license that governs the code you'll write. It's always a good idea to let others know immediately under which rules your code can be used with theirs. The value you specify in the license property will be displayed every time the plugin is installed.

The last property you see in the descriptor, `documentation` ❸ should be a multiline string containing a few paragraphs about how to use the plugin. The format accepted by this string is Markdown text (http://daringfireball.net/projects/markdown).

At this point you can make changes to the descriptor as you see fit. You can also add as many scripts to the plugin as desired (remember to use `griffon create-script`). You can even add an `_Events.groovy` script file (as described in chapter 8). This last script will allow your plugin to participate as a build event handler.

Speaking of events, we'll discuss install and uninstall events next.

PLUGIN EVENTS

Recall from figure 11.3 that the Clojure plugin performs some actions upon installation. Every plugin has the option to execute custom code when installed, and there's a similar option for when they're uninstalled. As you may suspect, this behavior is attained via a special pair of scripts that follow a specific naming convention.

Look inside your plugin's scripts directory. You should see at least three files whose names starts with an underscore (_) character. That character marks those scripts as special—you can't invoke them directly using the `griffon` command.

The first file, `_Install.groovy`, can be used to execute code when the plugin is installed. This is a regular Gant script file; you're free to place any code that may help your plugin in this script, the most typical scenario being the creation of a directory.

The next file, `_Uninstall.groovy`, will be called when the plugin is uninstalled. This is the time to clean up any plugin-specific artifacts that should not be present if the plugin is gone. Also be sure to remove any configuration settings related to the plugin that might cause trouble if left behind.

Finally there's `_Upgrade.groovy`. This script is called when `griffon upgrade` is invoked on your application. This is the perfect moment to synchronize any differences between plugin versions installed on an application.

That's all for now regarding build-time plugins. Let's continue with runtime plugins and then you'll be ready to make a plugin of your own.

11.2.2 Runtime plugins

Runtime plugins (or *addons*, as we prefer to call them) can dramatically enhance what an application can do. They're responsible for delivering a wide range of runtime elements. Addons are usually packaged within a plugin, which means build-time plugins are the main delivery option for an addon.

An addon can be created by invoking the `create-addon` command target:

```
$ griffon create-addon foo
```

This command will generate an addon descriptor, similar to a plugin descriptor, and will append extra code to the plugin's special scripts. You'll typically run this command

inside an existing plugin project with the same name. If you don't, the `create-addon` command will first create the plugin project and then create the addon descriptor. It's important to note that the addon descriptor and the plugin names must match; that is, if the addon name is `foo`, then the name of the plugin must also be `foo`. That code is needed to configure the addon so it can be used by an application; most of the time it can be left untouched because the Griffon conventions kick in. The convention is that the addon's name should be the same as the plugin's. This is enforced if you call `create-addon` outside of a plugin project; a new plugin matching the addon's name will be automatically created. Figure 11.5 shows the layout of a typical plugin/addon combination project.

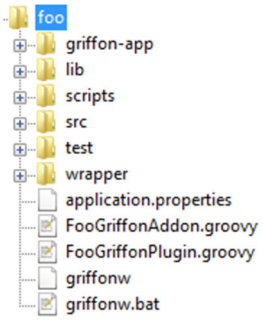


Figure 11.5 An addon named foo has been added to a plugin, also named Foo. The addon descriptor is highlighted.

Open the addon descriptor in an editor, and you'll see plenty of comments and reminder text for each of the possible contributions an addon can deliver. Table 11.1 summarizes what an addon can bring to an application.

Table 11.1 A list of all possible runtime elements that an addon can deliver to an application. Not all of them need to be specified.

Element	Description
`factories`	Nodes that will become available to MVC members
`methods`	Additional methods to be injected to MVC members
`props`	Additional properties to be injected into MVC members
`mvcGroups`	New MVC groups
`events`	Runtime event handlers

`attributeDelegates`
`preInstantiateDelegates`
`postInstantiateDelegates`
`postNodeCompletionDelegates`
	Additional strategies used to tweak node building

Addons have their own life cycle, and they're also tightly integrated with the runtime event system. The Griffon runtime will fire a series of events before and after each addon has been loaded and configured, and before and after all addons have been processed. You can listen to these events by placing appropriate application event handlers at `griffon-app/conf/Events.groovy` (see chapter 8 for a quick reminder on how to do that). Table 11.2 summarizes the events pushed by the application while loading addons.

Table 11.2 Events fired by an application while addons are loaded at boot time

Event	Arguments	Description
`LoadAddonsStart`	`app`	Fired before any addons are loaded.
`LoadAddonStart`	`name, addon, app`	Fired just before an addon's contributions are processed.
`LoadAddonEnd`	`name, addon, app`	Fired after an addon has been fully processed.
`LoadAddonsEnd`	`app, addons`	Fired after all addons have been loaded and processed. The `addons` argument is a map of addon instances keyed by name.

If you're curious, you may be wondering about the first lines of an addon descriptor. There are four methods whose signatures match the following snippet:

```
void addonInit(GriffonApplication app) { ... }
void addonPostInit(GriffonApplication app) { ... }
void addonBuilderInit(GriffonApplication app) { ... }
void addonBuilderPostInit(GriffonApplication app) { ... }
```

These methods are invoked by an application at specific points during startup. The first method is the local equivalent to the `LoadAddonStart` event, and the second method is equivalent to `LoadAddonEnd`. The remaining methods are called before and after builder contributions (factories, methods, properties, and builder delegates) are processed. You aren't forced to implement these methods; in fact, they're usually left untouched. You only need to be concerned with them should your addon need to run specialized code while it's being initialized.

Enough theory—it's time to build a plugin plus an addon to exercise what you just learned.

11.3 Creating the Tracer plugin and addon

Everybody knows that as an application grows, it gets harder and harder to visualize data flows, particularly when the user interacts with the application. During development, developers often rely on two techniques to keep track of the data flow: either launch the application in debug mode, attach it to a debugger, place some breakpoints at the appropriate places and see the data live, or litter the code with `println` statements. But there's a third alternative: dynamically intercept method calls. Seasoned Java developers may recognize this technique as applying an around advice or a before advice, as suggested by aspect-oriented programming (AOP). AOP became popular in the early 2000s and successfully penetrated the enterprise in tandem with the Spring framework (www.springframework.org). If AOP is an alien concept to you, don't worry. Groovy greatly simplifies applying AOP-like techniques thanks to its

extensive metaprogramming capabilities; this means you don't need to learn an AOP framework nor an AOP API in order to enhance your application.

It's settled, then. You'll use an AOP-like approach to intercept controller actions and model properties, similar to what figure 11.6 shows. By intercepting a controller action, you'll know when it has been activated; intercepting model properties will let you know when the data has changed and how.

You'll package this new behavior with a plugin/addon combination. This means you can install it at any time, and then uninstall it when it's no longer needed, thus leaving your code untouched and relieving you from launching a debugger.

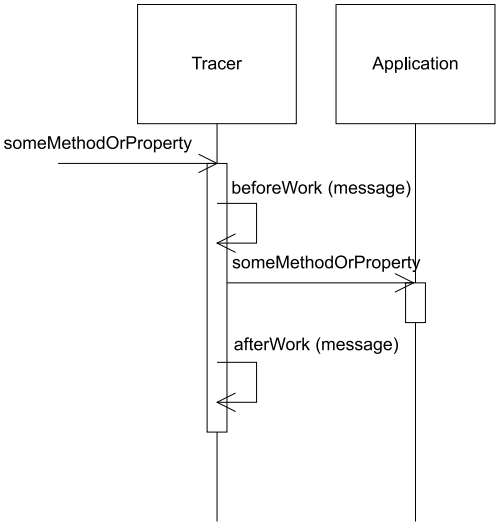


Figure 11.6 Intercepting a call to an application

11.3.1 Bootstrapping the plugin/addon

The first step is to bootstrap the plugin and addon descriptors. You know what's coming, don't you? That's right! You'll use the Griffon command-line tools to create both descriptors, as explained in section 11.2.

For this project, you'll name the plugin Tracer. Execute the following commands at the console prompt:

```
$ griffon create-plugin tracer
$ cd tracer
$ griffon create-addon tracer
```

You should have now a familiar set of files, similar to those shown in figure 11.7.

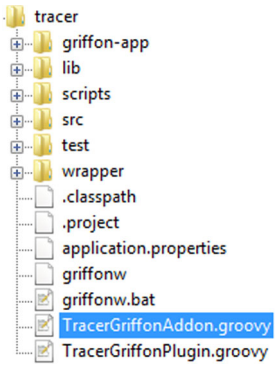


Figure 11.7 Contents of the Tracer plugin. You can see the plugin and addon descriptor files created by the Griffon commands. The addon descriptor is selected.

You'll keep the plugin/addon simple, which means you won't include any external libraries or create additional files. All the behavior will be concentrated in the addon descriptor. Go ahead and modify the plugin descriptor as you see fit.

Next you need to edit the addon descriptor by adding a skeleton of the behavior you want to provide, as shown in the following listing.

Listing 11.2 Initial addon code in `TracerGriffonAddon.groovy`

```
import java.beans.*
class TracerGriffonAddon {
```

```
    def events = [
        NewInstance: { klass, type, instance ->
        }
    ]

    void message(msg) {
        println msg
    }
}
```

To be implemented

You can see that this addon relies on an event handler to intercept and inject new behavior into a recently created instance. The new code will be injected into every single instance created by the application, regardless of its artifact type—some common artifact types are controller and service, for example. You can further tune this design choice if you want; for example, you could restrict the type to controller and model only.

`message()` is a generic message-printing method. Alternatives to this implementation would be to use a proper logging mechanism, or to send the message to a file or even to a database. You have the last word on this design choice too—we're just presenting the basics.

Now it's time to start adding some behavior to the addon.

11.3.2 Intercepting property updates

For now, you'll rely on the fact that observable beans publish change events whenever one of their properties changes value. Recall from chapter 3 that Java uses `PropertyChangeEvent` and `PropertyChangeListener` to enable change events and their handlers. Adding a `PropertyChangeListener` to the intercepted instance should be enough for now.

Edit the addon descriptor once more, so that its contents match the following listing. The revised parts are shown in bold.

Listing 11.3 Intercepting property updates on an observable instance

```
import java.beans.*
class TracerGriffonAddon {
    def events = [
        NewInstance: { klass, type, instance ->
            addPropertyChangeListener(instance)
        }
    ]

    void addPropertyChangeListener(target) {
        MetaClass mc = target.metaClass
        if(mc.respondsTo(target, 'addPropertyChangeListener',
           ➥[PropertyChangeListener] as Class[])) {
            target.addPropertyChangeListener({ e ->
               message "${e.propertyName}: '${e.oldValue}' ->
 '${e.newValue}'"
            } as PropertyChangeListener)
        }
    }
```

1 Intercepts observable bean

2 Verifies bean is observable

3 Adds listener

```
    void message(msg) {
        println msg
    }
}
```

At ❶ you add a call to a helper method that will be responsible for inspecting the bean and applying the new behavior. You can see at ❷ that the helper method doesn't blindly assume that the instance is observable; it checks via the instance's metaclass to see if it responds to the `addPropertyChangeListener` method. In this way you avoid an exception from being thrown at runtime. Finally at ❸ you define a closure and cast it to `PropertyChangeListener` using Groovy's `as` keyword. This is much better than defining an inline inner class,[1] don't you think?

11.3.3 Using the plugin

You have enough behavior to try out the plugin. You need to package it before you use it, though. This is again a simple task thanks to the griffon command line. Type the following at your command prompt:

```
$ griffon package-plugin
```

You should see a few lines pass by and finally get a griffon-tracer-0.1.zip file in the current directory. The name of the plugin is computed by convention. The version number is taken from the plugin descriptor, so make sure to update that file whenever you want to build a newer version.

You can install the plugin now that it has been packaged. Remember from section 11.1.2 that you can specify the path to a plugin zip if the plugin isn't available from a repository, which is precisely the case here.

Try installing the plugin in an existing Griffon application—using the `install-plugin` command and giving it the path of the zip file that was generated by the `package-plugin` command— and see what happens when you run the application. For illustration purposes, you'll use a simple yet effective calculator as see in figure 11.8. It takes two inputs and calculates their sum once you click the button. Inputs, the output, and the button's enabled state are bound to observable properties on the model.

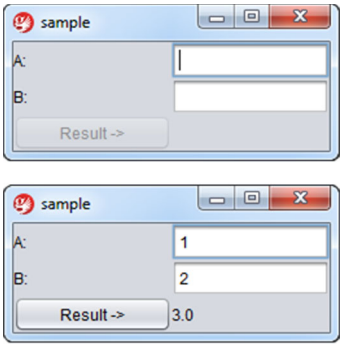


Figure 11.8 The calculator application. The first screen shows the application as it looks when it's launched. The second screen shows the application after inputs have been entered and the Result button has been clicked.

The application code is shown in the following listing. We won't discuss it thoroughly as our main concern is the Tracer plugin. First comes the view, where you can see all bindings being set up.

[1] Groovy has supported inner class definitions since version 1.7, but casting a closure is more concise.

Listing 11.4 The calculator's view

```
application(title: 'sample',
  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 'A:'
   textField(columns: 10,
             text: bind(target: model, 'inputA') )
   label 'B:'
   textField(columns: 10,
             text: bind(target: model, 'inputB') )
   button('Result ->', actionPerformed: controller.click,
          enabled: bind{ model.enabled })
   label text: bind{ model.output }
}
```

The next listing shows the model. You can appreciate all property definitions. There's also a local event handler that updates the model's `enable` property if both inputs have a value.

Listing 11.5 The calculator's model

```
import groovy.beans.Bindable
import griffon.transform.PropertyListener

@PropertyListener(enabler)
class SampleModel {
    @Bindable String inputA
    @Bindable String inputB
    @Bindable String output
    @Bindable boolean enabled = false

    private enabler = { evt ->
        if(evt.propertyName in ['enabled', 'output']) return
        enabled = inputA && inputB
    }
}
```

Last comes the controller, shown in the following listing. Remember that it uses proper threading to calculate the output outside of the EDT, and then it goes back inside the EDT to update the view by setting the model's `output` property.

Listing 11.6 The calculator's controller

```
class SampleController {
    def model

    def click = {
        execSync { model.enabled = false }
        String a = model.inputA
        String b = model.inputB
```

```
            try {
                Number o = Double.valueOf(a) + Double.valueOf(b)
                execSync { model.output = o }
            } catch(NumberFormatException nfe) {
                execSync { model.output = Double.NaN }
            } finally {
                execAsync { model.enabled = true }
            }
        }
    }
```

If you made the plugin correctly, you should get a console output every time a model property is updated. Launching the application yields

```
inputA: 'null' -> ''
inputB: 'null' -> ''
```

Hold on a second! What just happened? If you remember what was discussed in chapter 3, bindings take effect as soon as they're parsed in a View script; this means both inputs change value from `null` to empty strings, as those are the values of their respective source text fields. Updating both inputs with values (see figure 11.7) and clicking the button yields the following output in the console:

```
inputA: '' -> '1'
enabled: 'false' -> 'true'
inputB: '' -> '2'
enabled: 'true' -> 'false'
output: 'null' -> '3.0'
enabled: 'false' -> 'true'
```

Perfect! Now you know which values are held by model properties at a certain point during the application's execution, without needing to modify the application's code. You don't have a time reference on each output message, though perhaps adding one would be in order; the logging alternative looks more appealing now.

Let's get back to the tracer plugin in order to intercept whenever a controller action is called.

11.3.4 Intercepting action calls

In the next listing you'll go back to the plugin sources and edit the addon descriptor. You'll add another intercept handler to the `NewInstance` event handler, much like you did before.

Listing 11.7 Updated addon descriptor with additional behavior

```
import java.beans.*
class TracerGriffonAddon {
    def events = [
        NewInstance: { klass, type, instance ->
            addPropertyChangeListener(instance)
            injectActionInterceptor(klass, instance)
        }
    ]
```

1 Intercept bean

```
    void injectActionInterceptor(klass, target) {
        Introspector.getBeanInfo(klass).propertyDescriptors.each { pd ->
            def propertyName = pd.name
            def oldValue = target."$propertyName"
            if(!oldValue?.getClass() ||
               ➥!Closure.isAssignableFrom(oldValue.getClass())) return
            def newValue = { evt = null ->
                message "Entering $propertyName ..."
                oldValue(evt)                                    <-- Invoke old action behavior (2)
            }
            target."$propertyName" = newValue
        }
    }
}
```

The new handler requires both the instance and its class (1) because you're going to use the standard Java Beans inspection mechanism to figure out which properties can be intercepted. This is better than blindly assuming a specific property format or property definition. The handler then inspects the instance's class and iterates over all `PropertyDescriptor` instances that the class exposes. The handler performs some metaprogramming magic for each property whose value is a closure—it skips those that don't match the required criteria. You also save a reference to the current property value, which will allow you to call the default behavior later on.

Next you define the new value for a target action, which turns out to be another closure. Note how it relies on the previous saved reference (2) to the old behavior; this is how you can chain things together. Using Groovy's closures makes these steps a snap—you'd have to jump through a few hoops if you were using regular Java instead.

The last step is to assign the new behavior to the instance's property. You can rely on Groovy's dynamic dispatch capabilities to resolve the actual property name.

But what about the intercepting code? It calls the `message()` method and then forwards the call to the old behavior. This is known as an *around advice* in AOP terminology, because you're decorating the call before it's executed and then invoking the original behavior. Note that you can remove the original behavior in this manner, either intentionally or inadvertently.

This concludes what you can do with this plugin. It's time to test the changes. Don't forget to repackage the plugin by issuing the following command:

```
$ griffon package-plugin
```

If at any time you feel you need to start over, at least in terms of packaging, you can issue the `clean` command. This will remove any compiled sources and the latest version of the plugin zip file.

11.3.5 Running the plugin again

Install the plugin in the same fashion as you did before, by pointing the `install-plugin` command target to the zip file's path in your filesystem.

> **Get the latest version**
> Before you install the plugin, double-check that you have the latest version. Many headaches can be averted by a few seconds of carefully verifying that your tools, artifacts, and sources are in the proper state.

Running the calculator application and clicking the button, following the same steps as you did before, yields the following output on the console. The updated output is shown in bold:

```
inputA: 'null' -> ''
inputB: 'null' -> ''
inputA: '' -> '1'
enabled: 'false' -> 'true'
inputB: '' -> '2'
enabled: 'true' -> 'false'
Entering click ...
output: 'null' -> '3.0'
enabled: 'false' -> 'true'
```

There you have it! Of course, we only looked at a small fraction of what addons can do. The most common uses for addons are to provide new node factories and metaclass enhancements, besides event handlers, as you just saw. Be sure to browse the extensive list of plugins and addons that can be found at Griffon's plugin repository (http://artifacts.griffon-framework.org/plugins). You'll find plenty of addons that provide node factories, as you'll see in chapter 12.

Feel free to play around with the Tracer plugin, or better yet, create a plugin of your own. Once you're comfortable with it, you may want to release it to the public so that others can use it in their own applications. This is the topic of the next section.

11.4 Releasing the Tracer plugin

Releasing a plugin is another task that's automated by the Griffon command line. Isn't that great? Back in section 11.2.1, we described the properties that belong to a plugin's descriptor. The Griffon command line relies on them to generate appropriate metadata in the form of a JSON formatted file.

Perhaps you've noticed that file already while packaging your plugins. Here's the generated metadata for the Tracer plugin (the file's name is located in `target/package/plugin.json`):

```
{
    "type": "plugin",
    "name": "tracer",
    "title": "Plugin summary/headline",
    "license": "<UNKNOWN>",
    "version": "0.1",
    "source": "",
    "documentation": "",
    "griffonVersion":"0.9.5 > *",
```

```
    "description": "Brief description of Tracer.\n\nUsage\n----\nLorem
     ipsum\n\nConfiguration\n-------------\nLorem ipsum",
    "authors": [
        {"name":"Your Name","email":"your@email.com"}
    ],
    "dependencies": [],
    "toolkits": [],
    "platforms": []
}
```

It looks a bit impersonal, because we didn't update the plugin descriptor with specific information (author, title, and license elements). Nevertheless, it gives you an idea of what information will be stored in the plugin repository that can be used to identify a plugin. This same information is used by the install mechanism when downloading and installing a plugin in an application.

There are at least two approaches to releasing a plugin into the public:

- Manual
- Standard

The manual way gives you more choices in source storage because you're fully responsible for adding plugins to your source control system. You're free to choose the hosting solution for the plugin's zip file and source code. The drawback is that the plugin won't be visible with the `list-plugins` and `plugin-info` command targets, at least not until custom plugin repositories become available in Griffon.[2] You basically package the plugin using the method you now know, and then publish the zip file to a URL where other people can download it.

The standard way will add all the plugin sources to the Griffon plugin repository. You must be an authorized developer in order to do this, so make sure you send a request to the developer's list (dev@griffon.codehaus.org) before attempting to release a plugin using this approach.

After you're authorized, invoke the following command at your console prompt to release the Tracer plugin using the standard approach:

```
$ griffon release-plugin
```

This command target will take care of several tasks:

- Compile the plugin's sources
- Generating the plugin zip and descriptor
- Uploading the plugin zip to the central repository
- Updating the central plugin metadata

You can appreciate why you need to be an authorized developer to use this release approach.

No matter which approach you follow, it's always a good idea to send a message to the user list (user@griffon.codehaus.org) to let other developers know about the

[2] We expect this feature to be ready by the time this book hits the shelves.

availability of your plugin. Feedback is a powerful tool; rely on your peers to get that feedback sooner rather than later. And don't forget: release early, release often.

11.5 Summary

Perhaps the greatest strength found in the Griffon framework is its ability to be extended by plugins. This is an inherited feature from Grails. Both frameworks share a lot of traits in their plugin systems; for example, listing, installing, and uninstalling a plugin is virtually the same.

But when it comes to developing a plugin, you'll start to notice a few differences. Griffon distinguishes between build time and runtime plugins, which are also known as addons. This difference allows Griffon to be precise about the type of artifacts, libraries, and behavior that's exposed at build time versus at runtime.

A build-time plugin can add new libraries and provide build-time events and scripts. This is basically how you extend the framework's capabilities. Good examples of this kind of plugin are the testing-related FEST and easyb, because they never affect the running application.

An addon can add not only new libraries but also other runtime aspects, such as application event handlers, metaclass enhancements, and node factories. Addons expose another set of events that can be used to coordinate their initialization; and addons can even communicate with each other, whether they have a strict dependency on one another or not.

Regardless of what kind of plugins you choose to build, you'll eventually want to publish them so other developers can use them. There are two approaches to achieving this goal, manual and standard, both with their pros and cons. No matter which approach you choose, just remember that the Griffon command line is there to help you get to your goal.

Now it's time to make your application look great. In the next chapter, we'll examine different techniques and plugins that can be used to enhance the look of your application.

12 Enhanced looks

This chapter covers

- Adding custom nodes to views
- Using third-party view extensions

Views draw the user's attention, either by how they look or how they behave. It doesn't matter if you're a newcomer to Java Swing or a seasoned expert—there will be times when the standard widget set exposed by the JDK isn't enough to create a compelling user interface or to provide an incredible experience.

Fortunately, Swing components are extensible. There are multiple third-party components out there—some commercial, some open source. You can also change how standard and custom components look by applying new Look & Feel classes. If you don't know what "Look & Feel" means in Swing, think of themes and skins.

In this chapter, you'll learn how views can be extended by registering additional nodes that will handle custom components. You'll also learn how the view build process can be modified at specific points. Finally, we'll enumerate and explain some of the official SwingBuilder extensions harbored by the Griffon project.

12.1 Adding new nodes

You now know that SwingBuilder is a thin abstraction layer over Java Swing. This layer only covers the standard set of Swing components. Or does it? There are ways to register custom components directly on views (remember `widget()`, `container()`, and `bean()` from chapter 4?). There's also the option to register factories in an addon (as seen in the previous chapter), which requires you to build and package an addon.

But there are other ways to add new nodes to a view. We'll discuss them in the following order:

- *Adding new node factories*—This is the quickest method, and we'll cover it first.
- *Using implicit addons*—This method goes a step further as it increases code reuse.
- *Making your own builder*—This technique is the most versatile.

12.1.1 Registering node factories

As you know, SwingBuilder relies on node factories to manipulate and instantiate nodes. You may register as many node factories on a view script as needed. The advantages of registering a node factory are as follows:

- The node name is reusable everywhere in the script after the factory has been registered. This isn't the case when using the special nodes `container()`, `widget()`, and `bean()`.
- Factories determine how an instance is created. This is a useful feature when the component you want to instantiate doesn't provide a no-args constructor.
- Factories have additional methods that let you tweak parent-child relationships, how properties are set on a target instance, and so forth.

Let's assume for a moment that you'd like to add a `textField` component that has the ability to display a prompt. A quick search on the web should yield xswingx (http://code.google.com/p/xswingx/) as a result. Browsing the code, you'll discover `JXTextField`, a component that extends `JTextField`. You can add a node that knows how to handle this component as follows:

```
import org.jdesktop.xswingx.JXTextField
registerBeanFactory("promptField", JXTextField)
promptField(prompt: "Type something here", columns: 20)
```

You've taken a shortcut here. The method `registerBeanFactory()` takes two parameters: a string that will become the node name, and a class that's assumed to be a `javax.swing.Component` subclass. You can use this shortcut because `JXTextField` follows the JavaBeans conventions because it extends from `JTextField`, which also follows the conventions.

There's a more general version of the previous method that can be used to register any implementation of the `groovy.util.Factory` interface. This is its signature:

```
registerFactory(String name, groovy.util.Factory factory)
```

About xswingx

xswingx is a project that provides a set of components that, according to its creator, were missing from the popular SwingX project. Although you can certainly embed these particular components with the technique we just demonstrated, there's an easier way to do it. We'll revisit xswingx when we discuss official extension later in this chapter.

As a matter of fact, SwingBuilder will resolve all `registerBeanFactory()` calls to a `registerFactory()`; it wraps the supplied component subclass into an appropriate `ComponentFactory`.

All this talk about factories and we haven't showed you what a factory looks like! The following listing shows the contract of `groovy.util.Factory`.

Listing 12.1 The `groovy.util.Factory` interface

```
public interface Factory {
  boolean isLeaf()
  Object newInstance(FactoryBuilderSupport builder,
  ➥       Object name, Object value, Map attributes)
  ➥       throws InstantiationException, IllegalAccessException
  boolean onHandleNodeAttributes(FactoryBuilderSupport builder,
  ➥       Object node, Map attributes )
  void setParent(FactoryBuilderSupport builder,
  ➥       Object parent, Object child)
  void setChild(FactoryBuilderSupport builder,
  ➥       Object parent, Object child)
  void onNodeCompleted(FactoryBuilderSupport builder,
  ➥       Object parent, Object node)

  boolean isHandlesNodeChildren()
  boolean onNodeChildren(FactoryBuilderSupport builder,
  ➥       Object node, Closure childContent)

  void onFactoryRegistration(FactoryBuilderSupport builder,
  ➥       String registeredName, String registeredGroupName)
}
```

(1) FactoryBuilderSupport instance

Note that almost all methods take a `FactoryBuilderSupport` instance (1). As you may remember, `SwingBuilder` is a subclass of `FactoryBuilderSupport`, and Griffon's `CompositeBuilder` works with subclasses of `FactoryBuilderSupport`. Coincidence? Of course not! `CompositeBuilder` knows how to handle factories of any type.

Let's drill down into each of the factory's methods from the factory implementation perspective:

- `isLeaf()`—This method controls whether the node handled by this factory supports nesting of other nodes. Returning `true` will mark the node as a leaf node. You'll get a `RuntimeException` if you attempt to nest a child node in a leaf node.

- `newInstance()`—Perhaps the most important method that must be implemented, `newInstance()` is responsible for creating instances. You have access to the current context via the `builder` parameter, the node's name via the `name` parameter, the node's value (if any) via the `value` parameter, and any property that was set using Map syntax via the `attributes` parameter. When implementing this method, we recommend that you remove attribute keys for each value in the `attributes` map that was used to create the target instance; otherwise they'll be set again by the method we'll discuss next.
- `onHandleNodeAttributes()`—This method is responsible for setting any remaining properties that may not have been handled when `newInstance()` was called, and sets them on the target instance. Note that it returns a Boolean value. If the return value is `false`, the builder will ignore any remaining properties that the factory did not process explicitly. If the return value is `true`, the builder will attempt to set any properties that are still left on the Map. Overriding this method can be useful in various circumstances, such as when a type conversion must be executed before setting a property's value or when there are multiple setters for a particular property name.
- `setParent()` and `setChild()`—These methods handle the relationships between nodes. The former is called when the current node is the parent node, and the latter is called when the current node is the one being nested inside another.
- `onNodeCompleted()`—This method is your last chance to modify the node before the builder continues with the next sibling node found in the script. This is the perfect time to perform cleanup or assign lazy relationships, because all child nodes have been processed at this point. The builder context comes in handy when you need to keep track of private node data.

If all this seems a bit daunting, don't worry. You'll find plenty of code examples in the various SwingBuilder factories (http://mng.bz/jYi1). We're sure you'll encounter plenty of goodies and code snippets in that link that can help you.

The remaining methods are rarely used outside of very specialized factories:

- `isHandlesNodeChildren()` and `onNodeChildren()`—These are used to let the factory take control of the node's nested closure instead of the builder. This is how the `withWorker()` node (remember it from chapter 7?) processes nested closures that are mapped to the various life cycle methods on a SwingWorker instance.
- `onFactoryRegistration()`—You can use this method to perform sanity checks right after the factory has been added to a builder.

Figure 12.1 shows the flow followed by a factory's methods when the builder requests a node to be built.

If you're thinking of creating your own factories for handling custom components, we suggest you create a subclass of `groovy.util.AbstractFactory` as a starting point. It implements all factory methods except `newInstance()` with default behavior. To create a

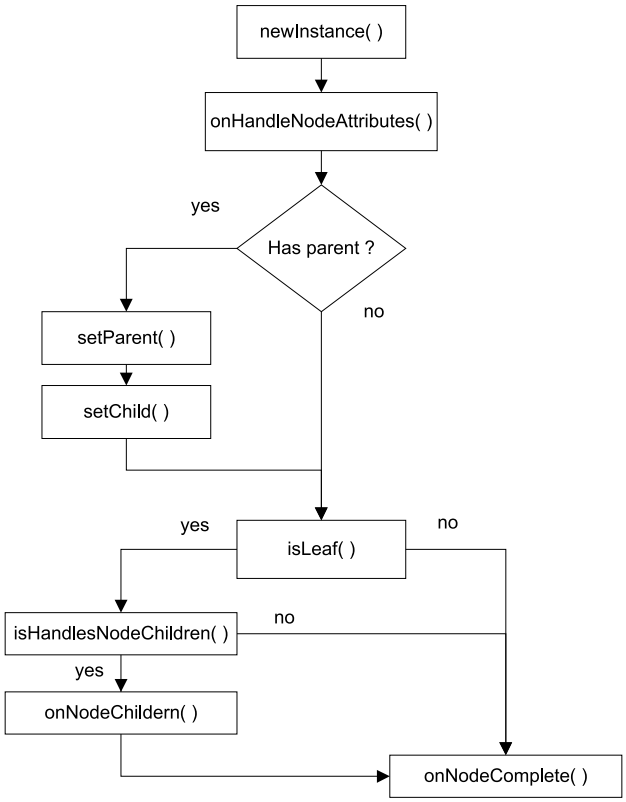


Figure 12.1 Factory methods used during node building. The `setChild()` method is called on the parent factory if it exists.

factory for a custom component that requires a constructor argument based on an orientation parameter, for example, you'd do the following:

```
import com.acme.MyCustomComponent
class CustomComponentFactory extends AbstractFactory {
  Object newInstance(FactoryBuilderSupport builder, Object name,
  ➥Object value, Map attributes)
    throws InstantiationException, IllegalAccessException {
    String orientation = attributes.remove('orientation') ?: 'left'
    return new MyCustomComponent(orientation)
  }
}
```

And that would be it!

This technique is useful for reusing the node many times in the same script. But what if you need to reuse the node across scripts? Registering the node on each script is a clear violation of the DRY (Don't Repeat Yourself) principle—there must be another way.

As a matter of fact, there are two other ways to solve this problem. The first involves an implicit addon, and the second is to make your own builder. Let's look at the implicit addon option first.

12.1.2 Using an implicit addon

The implicit addon technique for adding nodes requires you to touch a configuration file. Recall from chapter 4 that `griffon-app/conf/Builder.groovy` contains the configuration for all nodes available to the `CompositeBuilder`. This means all you have to do is somehow register a set of factories.

Let's look again at the default contents of `Builder.groovy` after a new application has been created:

```
root {
    'groovy.swing.SwingBuilder' {
        controller = ['Threading']
        view = '*'
    }
}
```

Here you can see the `root` prefix. That prefix name is a special one; it instructs `CompositeBuilder` to put all nodes into a default namespace. Node names will be available without any modifications.

What if there was another special prefix that instructed the builder to process an implicit addon? It turns out that there is! That special prefix is `features`, and it can hold all the properties an addon descriptor can define, except events and mvcGroups. This means you're left with factories, methods, props, attributeDelegates, preInstantiateDelegates, postIntantiateDelegates, and postNodeCompletionDelegates. Phew! That's still quite a lot!

To register a node for `MyCustomComponent` using an implicit addon, you only need to append the code shown in the following listing to `Builder.groovy`.

Listing 12.2 Defining a custom node using an implicit addon in `Builder.groovy`

```
features {
   factories {
      root = [
         myCustomComp: new MyCustomComponentFactory()
      ]
   }
}
```

A node named `myCustomComp` will be available to all view scripts now. You specify `root` as a prefix; this avoids node name collisions. The `root` prefix is special because it's handled as if there were no prefix specified; in other words, it's empty, and the node name can be used as is. If you set any other value as a prefix, it should be used when referring to the node.

The following snippet shows an example of setting a different node prefix:

```
features {
   factories {
      my = [
         button: new MyCustomButton()
      ]
   }
}
```

Now you can refer to this particular component as `mybutton()` in any view, as the following example depicts:

```
panel {
    mybutton(text: 'Click me!')
}
```

One neat trick that you can apply when registering a node factory is taking advantage of conventions. If the component you need to register behaves like a regular Java Swing component—if it has a no-args constructor and extends from `JComponent` or any of its superclasses—you can skip creating a factory and declare the component class, as follows:

```
features {
   factories {
      my = [
         clicker: javax.swing.JButton
      ]
   }
}
```

That wasn't so bad, was it? You can apply this trick to factories defined in a regular addon as well.

But as neat and concise as implicit addons are, they have one disadvantage: you can't share the node registration with other applications. In order to do that, you'll have to create a plugin/addon combination. We covered plugins and addons in the last chapter. What's new is that you must define a factories block, similar to the one demonstrated for implicit addons back in listing 12.2, but in this case you don't need to define a node prefix at all. Here's how it's done for the MigLayout plugin/addon:

```
import groovy.swing.factory.LayoutFactory
import net.miginfocom.swing.MigLayout
class MiglayoutGriffonAddon {
   def factories = [
      migLayout: new LayoutFactory(MigLayout)
   ]
}
```

That wasn't so bad either. Creating your own builder, on the other hand, requires more effort, but just a little more. Fair warning: we're going to dive deeper and deeper into the intricacies of builders and metaprogramming. But don't feel discouraged by this; creating a builder is a simple task, despite all the magic that occurs behind the curtains.

12.1.3 Creating a builder

This is perhaps the most thorough option you have at your disposal. Creating a builder means you have full control over which node factories get registered and how. You also have the option to register as many delegates as needed. This sounds similar to addons, doesn't it? The main advantage of creating a builder is that you can distribute your builder to be used outside of a Griffon application. Why would you want to

do that? Think of IDE integration for one, or the ability to use your builder with a simple Groovy script. As a matter of fact, this is how SwingXBuilder (and some of the official extensions you'll see later in the chapter) came to be in the first place.

Builders should extend from `FactoryBuilderSupport`. This ensures that the builder has all it needs to process factories and delegates, leaving you with the responsibility of adding as many factories as you like.

Say you'd like to create a builder for a `MyCustomComponent` and other cool components you may have lying around. The following listing shows what such a builder could look like.

Listing 12.3 Basic builder implementation

```
class CoolBuilder extends FactoryBuilderSupport {
    CoolBuilder(boolean init = true) {
      super(init)
    }

    void registerCoolWidgets() {
        registerFactory('myCustomComp', new MyCustomComponentFactory())
        registerFactory('pushButton', new PushButtonFactory())
    }
}
```

Take note of the builder's constructor. Every builder created in this way will automatically search for methods that follow a naming convention; the assumption is that these methods will register factories on the builder. We can appreciate the convention being applied to the `registerCoolWidgets()` method. This naming convention is as follows:

```
void register<groupName>() { ... }
```

This means that every method that starts with the word `register`, doesn't return a value, and takes no arguments will be automatically called after the builder instance is created. If that method happens to make a call to `registerFactory()`, those node factories will be added to the builder. You can use this approach to call anything you'd like during node registration; for example, you can make calls to register any delegates needed by the builder.

There's an additional benefit to automatic registration: all node factories and delegates declared in a single `register<groupName>` method will share the same `groupName`. What are groups you ask? As seen in chapter 4, all builder nodes are put together in groups, which facilitates the task of selecting which nodes can be activated or applied to other MVC members. A perfect example is the `Threading` group declaration found in `Builder.groovy`—you may recall that this configuration line makes it possible for controllers to gain access to SwingBuilder's threading facilities.

Once you're happy with the state of your builder, it's time to add it to an application. You can use a plugin to deliver the builder, as if it were an addon. Make sure you write appropriate install and uninstall instructions in the plugin's install and uninstall

scripts. What's more, you can use the same code that `create-addon` puts in those scripts; just substitute the addon reference for your builder's. If you get lost somehow, you can inspect existing builder plugins for additional clues.

That's it for adding nodes. You now know more about the magical things that happen during the node-building process, but there's more. We've mentioned builder delegates a few times in this and the previous chapter; perhaps it's time to inspect them further and discover exactly how they come into play.

12.2 Builder delegates under the hood

As you surely know by now, `FactoryBuilderSupport` is the base class for all builders that rely on factories to build nodes. Factories encapsulate a great part of a node's build cycle, as you saw in the previous section. But it's the builder that orchestrates when and how those factories are called.

It all starts by matching what appears to be a method name to a node name backed by a factory. Node names are *pretended* methods, because there's no way that `FactoryBuilderSupport` can know in advance which node names will be used. `FactoryBuilderSupport` relies on a Groovy metaprogramming technique called *method missing*, which works as follows. Whenever you call a method on a Groovy class, the Meta Object Protocol (MOP) follows a series of paths searching for a suitable way to invoke the method. If no match is found, it will attempt calling a special method named `methodMissing()`. If such a method is provided by the class (there's no default implementation for this method as handling this type of call is an optional operation), then it's up to the implementer to decide how the call should be handled.

This is basically what `FactoryBuilderSupport` does. Whenever a method is invoked and it isn't found by regular means, `methodMissing()` kicks in. The builder will attempt to resolve the node name to a factory, and if a match is found it will then kick start the node build cycle. But if no match is found, you'll get a `MissingMethodException` as a result of the method call.

Once a factory is resolved, the build cycle continues; the builder will call out specific methods defined internally. Some of those methods expose extension points via the closure delegates. Figure 12.2 shows the sequence of method calls exchanged between builder and factory in order to build a node.

Without further ado, these are the types of delegates you may add to a builder:

- PreInstantiate delegates
- PostInstantiate delegates
- Attribute delegates
- PostNodeCompletion delegates

We'll discuss each delegate type in the order the builder encounters them during the build cycle.

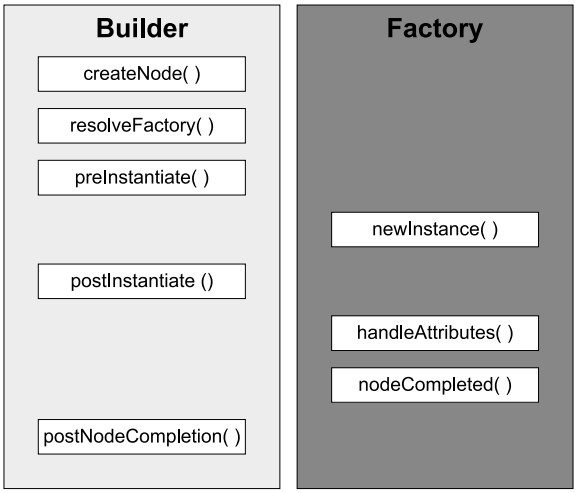


Figure 12.2 Builder and factory engaged in the node build cycle. Delegates are called on each phase that matches their name.

12.2.1 Acting before the node is created

PreInstantiate delegates are the first to be invoked, right before the node is created. This is their signature:

```
{FactoryBuilderSupport builder, Map attributes, value -> ... }
```

The most typical use for this type of delegate is to perform sanity checks on the attributes that are generic to all nodes.

For example, you might want to check that an attribute named `orientation` always has exactly one of the following string values: `"left"` or `"right"`. If for some reason that isn't the case, your delegate may attempt a conversion or set a default value. After this, every node that requires a valid value for its `orientation` property will have it. Here's what this delegate would look like:

```
builder.addPreInstantiateDelegate { builder, attributes, value ->
   String orientation = attributes.orientation?.toString()?.toLowerCase()
   if(orientation in ['left', 'right'])
       attributes.orientation = orientation
   else
       attributes.orientation = 'left'
}
```

The advantage of adding a delegate of this type is that you can effectively update attributes without needing to change the code of a single factory.

12.2.2 Tweaking the node before properties are set

PostInstantiate delegates are the counterparts of the previous type. They go into effect once the node has been built but before the builder has had a chance to set any remaining attributes on it. This is their signature:

```
{FactoryBuilderSupport builder, Map attributes, node -> ... }
```

Note the slight change in the signature compared with the previous type. The third argument is the node that has just been built.

What are delegates of this type good for? Well, you could perform additional specific sanity checks on attributes, now that you know which node was built. You could also inspect the node hierarchy in order to tweak parent-child relationships or to add a temporal variable to the parent context—this variable could later be processed by the parent's factory or another delegate.

Access to the current builder context is key to getting the job done. Here's how you can do it:

```
builder.addPostInstantiateDelegate { builder, attributes, node ->
   builder.context?.special = attributes.remove('special') ?: false
}
```

This delegate provides a hint for the parent factory so that it knows the current node (its child) must be processed in a special way. But what if the factory doesn't have code that can handle this flag? The next two delegates can help remedy this situation.

12.2.3 Handling node properties your way

Perhaps you remember seeing an `id` property being used on view scripts from time to time. This property doesn't belong to any component class; rather, it's handled by the builder using an attribute delegate. Attribute delegates are guaranteed to be called after a node instance has been created but before any properties are set by the factory or the builder. This means you can apply synthetic properties to a node, like the `id` property.

Attribute delegates have the following signature:

```
{FactoryBuilderSupport builder, node, Map attributes -> ... }
```

Note the subtle change in the order of arguments when compared to the postInstantiate delegates.

The following snippet shows how SwingBuilder handles the `id` attribute set on any node, in case you were wondering:

```
{FactoryBuilderSupport builder, node, Map attributes ->
    String id = attributes.remove('id')
    if(id) builder.setVariable(id, node)
}
```

These are likely the type of delegates that you'll encounter the most. SwingBuilder has a few of them (besides the `id`-handling one), and other builders have their own, as you'll soon find out when we discuss CSS for Swing components later in this chapter.

There's one last delegate type that needs to be discussed.

12.2.4 Cleaning up after the node is built

PostNodeCompletion delegates are rarely seen, as most factories prefer to handle cleanup code in their own `onNodeCompleted()` method. But in the event that you

need to perform cleanup operations that may affect several nodes, these delegates are the perfect solution. They don't get a chance to process a node's attributes, as they all should have been consumed by now, so their signature is as follows:

```
{FactoryBuilderSupport builder, parent, node -> ... }
```

This means it's a good idea to store temporal data on the builder's current context, using any of the other three delegates we just discussed. For example, this is how you can recall the special flag we set on the postInstantiate delegate example:

```
{FactoryBuilderSupport builder, parent, node ->
    if(builder.context.special) {
        parent?.postInit(node)
    }
}
```

This snippet assumes that the parent node has a `postInit()` method that should be called only when the current node has been marked as special. You can choose to remove a child element, change the node's appearance, or build a new object on the fly. Quite frankly, the possibilities are endless once you get the hang of the build cycle.

You might want to explore the vast space of third-party Swing components now that you know more about the inner workings of builders and their delegates, but before you embark on that wonderful journey, let's take a closer look at some Griffon-related projects and plugins that can make your life easier. It also makes sense to run a survey of the existing Griffon plugins (http://artifacts.griffon-framework.org/plugins) before you start to build your own. It would be sad to invest your time creating a builder or plugin for a set of components that have been covered by an official extension already, wouldn't it?

12.3 Quick tour of builder extensions in Griffon

SwingBuilder isn't the only builder that can be configured in a Griffon application. You know this because we just discussed how custom builders can be created. We've also mentioned SwingXBuilder a few times now. As a matter of fact, there are plenty of builder extension options. We'll briefly cover the main builders you're most likely to encounter in a daily basis. Let's start with SwingXBuilder because you already know a bit about it.

12.3.1 SwingXBuilder

In chapter 7, we explained that SwingXBuilder provides additional threading facilities in the form of the `withWorker()` node, an abstraction over SwingLabs' `SwingWorker` and JDK 6's `SwingWorker`. There's more to SwingXBuilder than that. The SwingX project was born as an incubator for ideas and components that might eventually find their way into the JDK. `SwingWorker` is one of the few that made the transition.

You'll find a number of interesting components in the SwingX component suite. Many were designed as replacements for existing Swing components; for example, `JXButton` replaces `JButton`. SwingX offers more than just components, though. It also

provides a Painters API that lets you override how a component is painted on the screen without needing to create a subclass of said component.

The easiest way to install SwingXBuilder (http://griffon.codehaus.org/SwingXBuilder) on an application is by installing its companion plugin. That's right, let the framework do the hard work for you; the plugin will configure the builder on the application's metadata and config files. It will also copy required libs and dependencies when needed. Invoke the following command at your command prompt to install the plugin and builder:

```
$ griffon install-plugin swingx-builder
```

Once you have it installed, you're ready to work.

One last piece of information before you begin—remember we said that some SwingX components were designed as replacements? Well, SwingXBuilder took that into consideration, which means that some of its node names may clash with SwingBuilder's. That's why this builder is configured with a different prefix: `jx`. Whenever you see a node name that starts with that prefix, it's a node that has been contributed by SwingXBuilder.

Figure 12.3 shows a simple application that contains two main elements: a header and a TaskPane container. The header component uses the Painters API. This painter, in particular, is a compound of three other painters:

- The first painter provides the base color, a very dark shade of gray—almost black.
- The second painter provides the pinstripes. You can change and tweak the stripes' angle, color, spacing, and more.
- The last painter provides a glossy look. Everything looks better with a bit of shine.

The TaskPane container emulates the behavior seen on earlier versions of the Windows file explorer. It's a `TaskPaneContainer`, and as its name implies it serves as a place to embed TaskPanes. There are two TaskPanes in this container. Clicking Task Group 2 will expand its contents (with a smooth animation); clicking it again will collapse it. You can embed any Swing components in a TaskPane.

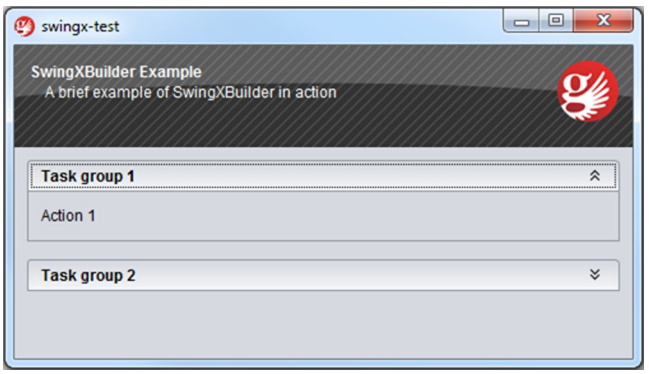


Figure 12.3 A simple application composed of SwingX components. The header's visuals were updated by a set of painters.

What's that? What about the code, you say? It couldn't be any easier, as shown in the next listing.

Listing 12.4 Some SwingXBuilder components in action

```
import java.awt.Color
import org.jdesktop.swingx.painter.GlossPainter

gloss = glossPainter(
paint: new Color(1f, 1f, 1f, 0.2f), position:
➥GlossPainter.GlossPosition.TOP)
stripes = pinstripePainter(paint: new Color(1f, 1f, 1f, 0.17f),
➥spacing: 5.0)
matte = mattePainter(fillPaint: new Color(51, 51, 51))
compound = compoundPainter(painters: [matte, stripes, gloss])

application(title: 'swingx-test', 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()
    jxheader(constraints: NORTH, title: "SwingXBuilder Example",
        description: "A brief example of SwingXBuilder in action",
        titleForeground: Color.WHITE,
        descriptionForeground: Color.WHITE,
        icon: imageIcon("/griffon-icon-48x48.png"),
        preferredSize: [480,80],
        backgroundPainter: compound)
    )

    jxtaskPaneContainer(constraints: CENTER) {
        jxtaskPane(title: "Task group 1") {
            jxlabel("Action 1")
        }
        jxtaskPane(title: "Task group 2", expanded: false) {
            label("Action 2")
        }
    }
}
```

1 Define painters

2 Define compound painter

3 Define JXHeader

4 Define JXTaskPaneContainer

First, the painters are defined 1. SwingXBuilder exposes a node for each of the basic painters you'll find on the Painters API. Then the compound painter is created 2; it only requires a list of painters to be used. At 3 the header component is built. Note that all it takes is setting a few properties on that component. Setting the compound painter is done as with every other property—there's no magic to it. Finally the TaskPanes are added to their container 4. Note that the first TaskPane refers to a `jxlabel` node, whereas the second refers to a `label` node. This means that the first pane should have a `JXLabel` instance and the second should have a `JLabel` instance. The use of the `jx` prefix makes switching from one type of node to the other a simple task.

What's more, the `CompositeBuilder` makes mixing nodes from two builders a trivial task. The alternative would be quite verbose in terms of keeping the appropriate builder references and node contexts in check. If Griffon did not provide a `CompositeBuilder` then Views would look like this:

```
swingx.jxpanel {
    swing.borderLayout()
    swingx.jxheader(constraints: NORTH, title: "SwingXBuilder",
        description: "Life without CompositeBuilder",
        titleForeground: Color.WHITE,
        descriptionForeground: Color.WHITE,
        icon: imageIcon("/griffon-icon-48x48.png"),
        preferredSize: [480,80],
        backgroundPainter: compound)
    )
    swing.panel {
        swing.gridLayout(cols: 1, rows: 2)
        swing.button('Regular Swing button')
        swingx.jxbutton('Enhanced SwingX button')
    }
}
```

This snippet assumes there are two pre-existing builder instances; `swing` points to a SwingBuilder instance, whereas `swingx` points to a SwingXBuilder instance. Notice how you must qualify nodes with their respective builders in order to get the expected result.

Make sure you check out SwingX's other components. In particular, we're sure you'll find these interesting:

- `JXErrorPane`—Useful for displaying errors caused by an exception. Highly configurable.
- `JXTitlePanel`—A `JPanel` replacement that comes with its own title area.
- `JXTable`—Row highlighting and sorting are but a few of its features. Some of its features have been merged into JDK's JTable.
- `JXHyperLink`—A button that behaves like a hyperlink as seen on web pages.

The next builder we'll cover is also quite popular.

12.3.2 JideBuilder

The announcement of the JIDE Common Layer (JCL) project (http://java.net/projects/jide-oss/) becoming an open source project back in 2007 surprised many. Up to that point, JIDE was a closed source component suite developed by JIDE Software. Many components have been open sourced since the announcement. You'll find some easily recognizable components, like split buttons (a component that's both a button and a menu), lists that can display check boxes (without the need of a custom renderer), and combo boxes that support autocompletion, just to name a few. Despite the number of components in the JCL, it only represents about 35 percent of the components provided by JIDE Software; the rest are available through the purchase

of a commercial library. JideBuilder, like the previous builders we've discussed, provides a set of components ready to be used in Griffon Views. JideBuilder covers all components available in JIDE CL; the commercial components aren't supported by the builder. But should you need to support any of the commercial components, the task of adding them to the builder isn't that difficult: you just need to register a factory for each component on the builder itself, and of course have a valid JIDE license available in your settings.

To install the builder, install its companion plugin. This is a recurring theme, isn't it? We did mention that plugins come in handy, and, as you'll soon see, all officially supported builders can be installed via plugins. The following command should do the trick:

```
$ griffon install-plugin jide-builder
```

Take a look at the builder's documentation site (http://griffon.codehaus.org/JideBuilder) to find out more about the components that are now at your disposal. We're sure you'll find SplitButton useful, especially for an application that requires enterprise-like behavior. Here are some other components that fall into that category:

- `AutoCompletionComboBox`—A combo box that comes with an autocompletion feature. Options will be selected as you type on the combo box's input field.
- `TriStateCheckBox`—Useful when enabled and disabled settings are not enough.
- `SearchableBar`—Adds searching capabilities to tables and trees.
- `DateSpinner`—A spinner component that knows how to work with dates.

Figure 12.4 presents an application that showcases three JIDE components: `checkBoxList`, `jideButton`, and `jideSplitButton`.

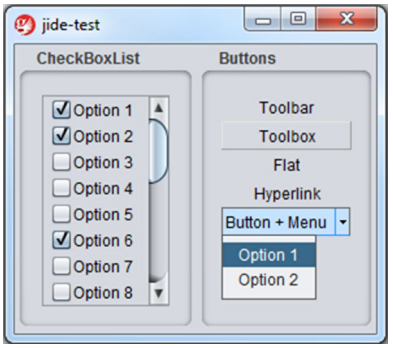


Figure 12.4 An application built with JIDE components. The `checkBoxList` is at the left. Four `JideButtons`, each with a different style applied, are at the right. The fifth and last button is a `JideSplitButton`, a combination of button and menu.

The list on the left in figure 12.4 doesn't require additional properties to be set in order to display a checkbox per entry. Just set the data you need, and that's all.

The first four buttons on the right belong to the same type, but they're rendered with different styles. Hovering over Toolbar gives the button raised edges, like the ones Toolbox has. Hovering over Flat doesn't change its appearance; it remains flat. Hovering over the fourth button, Hyperlink, gives the text an underline.

Button + Menu is the special button we've been talking about: a combination of button and menu. If you keep the button pressed, a popup menu appears, as shown in figure 12.4. The code is straightforward, as the following listing shows.

Listing 12.5 JIDE's `CheckBoxList` and buttons

```
import com.jidesoft.swing.ButtonStyle

data = (1..20).collect([]){"Option $it"} as Object[]
application(title: 'jide-test', 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(border: titledBorder(title: "CheckBoxList"), constraints: WEST) {
      scrollPane(constraints: context.CENTER) {
          checkBoxList(listData: data)
      }
   }
   panel(border: titledBorder(title: "Buttons"), constraints: CENTER) {
      panel{
         gridLayout(cols: 1, rows: 5)

         ["Toolbar", "Toolbox", "Flat", "Hyperlink"].each
➥{ style ->
            jideButton(style, buttonStyle:
➥ButtonStyle."${style.toUpperCase()}_STYLE")
         }
         jideSplitButton("Button + Menu",
            customize: { m ->
               m.removeAll()
               (1..2).each{ m.add "Option $it" }
            })
      }
   }
}
```

1 Sample date for list

2 CheckBoxList

3 Four JideButtons

4 jideSplitButton

Just pass an array of objects 1 or a model as data to create `CheckBoxLists` 2.

`JideButtons` have a style property that controls how they're rendered to the screen. You can see a fairly common Groovy trick 3 used to read a constant field from a Java class that follows a pattern. In this case, each element on the styles list serves as the button's text and as the basis to read a constant field on the `ButtonStyle` class.

Creating the menu of a `JideSplitButton` 4 is a matter of defining a closure for its `customize` property. Notice that all menu items are removed first, and then some are added. Duplicate menu items will start piling up every time you display the menu if it's not done this way. This is due to `JideSplitButton`'s behavior of keeping a reference to the menu it created the first time.

The next builder is sure to catch your eye.

12.3.3 CSSBuilder

Hold on a moment! Is that CSS as in Cascading Style Sheets? As in a technology that's typically associated with web content but not desktop? The answer is, happily, yes! Styling desktop components by means of CSS, or a CSS-like solution, is one of those goals that desktop developers often look for, besides better threading and binding.

CSSBuilder is a wrapper on a handful of classes that belong to the Swing-clarity project (http://code.google.com/p/swing-clarity/), whose creators are Ben Galbraith and Dion Almaer, from ajaxian.org fame. Those guys used to work on the desktop side before riding the Ajax wave revolution.

The CSS support provided by Swing-clarity is able to parse CSS2 selectors and colors. On top of Swing-clarity, CSSBuilder adds support for 71 or more custom properties that are specific to Java Swing. Figure 12.5 depicts a trivial application where all the visual components have received a facelift via CSS.

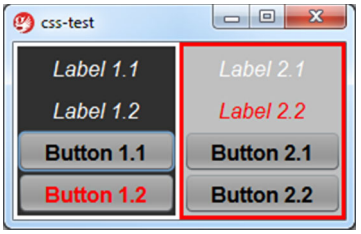


Figure 12.5 A CSS-styled Griffon application. Each section has a border applied, labels use italics, and buttons have bolded text. Button 1.2 and Label 2.2 share red as their foreground color.

In figure 12.5, the left side's background is darker than the right. Labels have an italicized style, whereas buttons have a bold weight. All buttons share the same background color regardless of where they're placed. One button and one label have a red foreground color. The text on all components is centered. Are you ready to see the CSS for this? The following listing contains all that we just described, in CSS format.

Listing 12.6 Swing CSS stylesheet

```
* {
   color: white;
   font-size: 16pt;
   swing-halign: center;
}
#group1 {
   background-color: #303030;
   border-color: white;
   border-width: 3;
}
#group2 {
   background-color: #C0C0C0;
   border-color: red;
   border-width: 3;
}
jbutton {
   font-weight: bold;
   background-color: #777777;
   color: black;
}
jlabel { font-style: italic; }
.active { color: red; }
```

1 General purpose styles
2 Custom CSS property
3 Left group style
4 Right group style
5 Button style
6 Label style
7 Targeted style

As you can see, CSSBuilder lets you apply a generic style 1 to all components. The custom `swing-halign` property 2 is but one of the many Swing-specific properties you can use; this one, in particular, will center the text of a component. Next, you can see CSS properties for the left 3 and right 4 groups, with background and border settings. Notice that groups use a selector that starts with a # character; its nature will

be revealed soon in the view code. Next, you can see button ❺ and label ❻ properties. They're defined using another type of selector that matches the class name of the target component. Last, you can see another selector ❼ that defines a foreground property with red as the value.

The view code is simple, as the following listing attests.

Listing 12.7 A CSS styled view

```
application(id: "mainFrame", title: 'css-test',
  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: 1)
  panel(name: "group1") {
      gridLayout(cols: 1, rows: 4)
      label("Label 1.1")
      label("Label 1.2")
      button("Button 1.1")
      button("Button 1.2", cssClass: "active")
  }
  panel(name: "group2") {
      gridLayout(cols: 1, rows: 4)
      label("Label 2.1")
      label("Label 2.2", cssClass: "active")
      button("Button 2.1")
      button("Button 2.2")
  }
```

❶ Resolves to named selector

❷ Resolves to class selector

❶ Resolves to named selector

❷ Resolves to class selector

There isn't a lot of information here that suggests that this view can be styled with CSS, other than the obvious `cssClass` property ❷. Hold on a second: that property is applied to a `JLabel` and a `JButton`, but those components know nothing about CSS. Their node factories also know nothing about CSS. How, then, is the application able to apply the correct style?

The answer lies in attribute delegates. CSSBuilder registers a custom attribute delegate that intercepts the `cssClass` attribute and applies the style. Remember when we said that groups use a special selector? Well now you know how it can be defined ❶. As a rule, any node that has a `name` property defined (this is a standard Swing property, by the way) will be accessible via the `#` selector (like `#group1` and `#group2`), whereas any node that defines a `cssClass` property will be accessible via the dot (`.`) selector (like `.active`).

There are additional features to be found in CSSBuilder, such as jQuery-like component finders using the `$()` method. Make sure you review the builder's documentation (http://griffon.codehaus.org/CSSBuilder) to learn more about all the properties and methods.

You must perform two additional tasks before trying this example for yourself:

1. Save the CSS stylesheet in a file with a conventional name and location. Save the contents of listing 12.6 as griffon-app/resources/style.css. Don't worry, you can

to use a different name and location if needed, but placing the file there with that particular name will save you the trouble of additional configuration.

2 Tell the CSS system which elements need to be styled. Look carefully at listing 12.7 and you'll see that the application node has an `id` declared with the value `mainFrame`. You'll use this `id` to instruct the CSS system that the frame and its contents need styling.

   Open `griffon-app/lifecycle/Startup.groovy` in your favorite editor. We've chosen this life cycle script because all views have been constructed by the time it's called. This means the `mainFrame` will be ready to be styled.

   Type the following snippet into the script, save it, and run the application:

```
import griffon.builder.css.CSSDecorator
CSSDecorator.decorate("style", app.builders.'css-test'.mainFrame)
```

We've covered component suites and component styling so far. Let's jump into graphics and drawings for a change.

12.3.4 GfxBuilder

The JDK comes with a number of drawing primitives and utility classes that are collectively known as Java 2D (http://java.sun.com/docs/books/tutorial/2d). Because every Swing component is drawn using Java 2D, it makes sense to review what you can do in Java 2D—if you want to go the long route, that is. Java 2D suffers from the same problems you'll encounter in plain Swing code. That's why GfxBuilder was born in the first place, just as SwingBuilder was for Swing.

Based on this information, you can expect GfxBuilder to provide a node for each of the Java 2D drawing primitives (like Rectangle, Ellipse, and Arc). It also provides nodes for setting antialiasing (removing those ugly jaggies, http://en.wikipedia.org/wiki/Jaggies), rendering hints, area operations, and much more. The following list summarizes the types of nodes that are available to you when using GfxBuilder:

- *Canvas*—The surface area where you place your drawings.
- *Standard shape nodes*—Rect, circle, arc, ellipse, path.
- *Additional shape nodes*—Asterisk, arrow, cross, donut, lauburu, and more; these shapes come from the jSilhouette (https://github.com/aalmiray/jsilhouette-geom) shape collection.
- *Standard and custom strokes*—Think of strokes as shape borders.
- *Area operations*—Add, subtract, intersect, xor.
- *Utility nodes*—Such as color, group, clip, image.

What really makes using GfxBuilder a better experience than just plain Java 2D (other than the use of Groovy features) is that it comes with a scene graphs baked right in. Scene graphs allow graphics to be defined in *retained* mode, whereas Java 2D works in *direct* mode. Direct mode means that graphics primitives will be drawn to the screen as soon as the code that defines them is processed. Retained mode means that a scene graph is created and a node is assigned to each drawing instruction. The scene graph

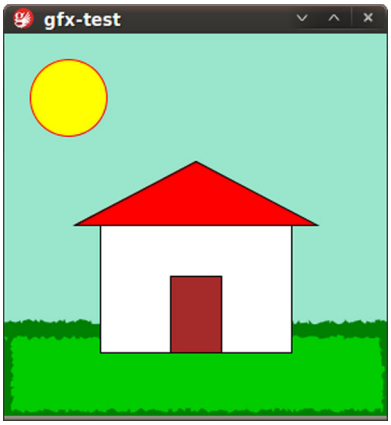


Figure 12.6 A simple drawing made with geometric objects. There are three rectangles, one triangle, and one circle, with different colors and strokes applied.

controls when the graphics should be rendered to the screen and when they should be updated. This relieves you of the burden of keeping track of areas that need to be redrawn or updated—the scene graph does it for you.

Figure 12.6 is a computerized rendition of what many of us drew as children at elementary school: a red-roofed house sitting on a patch of green grass, with the sun shining over it and a clear blue sky.

We know—it's a bit childish. It will surely never grace the halls of a respected art gallery, but one can dream, right? Still, this picture is a composition of geometric shapes, colors, and strokes. How complicated can it be? We'll let the code speak for itself in the next listing.

Listing 12.8 Drawing a happy house with Groovy

```
application(title: 'gfx-test', 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]){
   canvas(preferredSize: [300, 300]) {
      group {
         antialias true
         background(color: color(r: 0.6, g: 0.9, b:0.8))
         rect(y: 230, w: 300, h: 70, f: color(g: 0.8), bc: color(g:0.5)){
            wobbleStroke()
         }
         rect(x: 75, y: 150, w: 150, h: 100, f: 'white')
         triangle(x: 55, y: 150, w: 190, h: 50, f: 'red')
         rect(x: 130, y: 190, w: 40, h: 60, f: 'brown')
         circle(cx: 50, cy: 50, r: 30, f: 'yellow', bc: 'red')
      }
   }
}
```

Blue sky

That's pretty straightforward, isn't it? A background color turns out to be the blue sky. There's a patch of green grass, complete with some grass leaves (the wobbly stroke).

The house is composed of a white wall, a red roof, and a brown door. Last, the sun shines over the whole scene.

Perhaps it's lost in the code's simplicity, but notice that Swing nodes (application and canvas) and graphics nodes (group, rect, circle, and so on) merge in a seamless way. There's no artificial bridge between them. We know we've said this a few times already, but this is precisely the kind of power that Griffon's `CompositeBuilder` puts at your fingertips.

Although there are other features to be found in GfxBuilder (which could fill a chapter of their own), we must keep things simple. Suffice it to say that every gfx node is also an observable bean, and almost every property triggers a `PropertyChangeEvent`, which means you'll be able to use binding with them. Another powerful feature is the ability to define your own nodes via node composition, and not just by subclassing a node class.

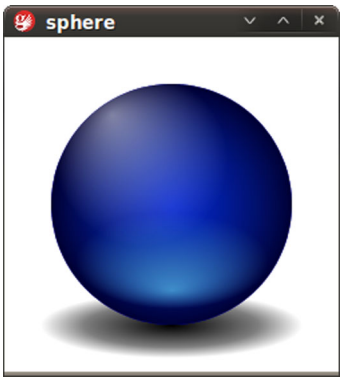


Figure 12.7 A sphere drawn with nothing more than two circles and four gradients

Figure 12.7 is a remake of an example shown in Chet Haase and Romain Guy's *Filthy Rich Clients* book (http://filthyrichclients.org; highly recommended if you want to learn the secrets for good-looking and well-behaving applications). It's a sphere created from circles and gradients alone; in other words, 2D primitives giving the illusion of a 3D object.[1]

This time the code is partitioned in two: the view and a custom node that knows how to draw spheres. Let's look at the view first (see the next listing), because it's the simplest of the two.

Listing 12.9 The `SphereView` view script

```
application(title:'sphere', 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]) {
  canvas(preferredSize: [250, 250]) {
     group {
        antialias true
        background(color: color('white'))
        customNode(SphereNode, cx: 125, cy: 125)
     }
  }
}
```

1 Custom node definition

[1] Of course, everything you see in this book and your screen are 2D primitives. Still, the sphere looks like you can almost grab it.

As you can see, the view code is similar to the first GfxBuilder example. This time, there's a new node ❶ used to render a sphere object. The `customNode` node takes either a class or an instance of `CustomNode`: in this case `SphereNode`, whose entire definition is found in the next listing.

Listing 12.10 `SphereNode` definition

```
import java.awt.Color
import griffon.builder.gfx.Colors
import griffon.builder.gfx.GfxBuilder
import griffon.builder.gfx.GfxAttribute
import griffon.builder.gfx.DrawableNode
import griffon.builder.gfx.CustomGfxNode

class SphereNode extends CustomGfxNode {
  @GfxAttribute(alias="r") double radius = 90
  @GfxAttribute double cx = 100
  @GfxAttribute double cy = 100
  @GfxAttribute Color base     = Colors.get("blue")
  @GfxAttribute Color ambient  = Colors.get(red: 6, green: 76, blue: 160,
  ➥alpha: 127)
  @GfxAttribute Color specular = Colors.get(r: 64, g: 142, b: 203, a: 255)
  @GfxAttribute Color shine    = Colors.get("white")

  SphereNode() {
    super("sphere")
  }

  DrawableNode createNode(GfxBuilder builder) {
    double height = radius * 2
    builder.group(borderColor: 'none') {
            circle(cx: cx, cy: cy+radius,
            ➥radius: radius, sy: 0.3, sx: 1.2) {
        radialGradient {
          stop(offset: 0.0f, color: color('black'))
          stop(offset: 0.6f, color: color('black').derive(alpha: 0.5))
          stop(offset: 0.9f, color: color('black').derive(alpha: 0))
        }
      }

      circle(cx: cx, cy: cy, radius: radius) {
        multiPaint {
          colorPaint(color: base)
          radialGradient(radius: radius) {
            stop(offset: 0.0f, color: ambient)
            stop(offset: 1.0f, color: rgba(alpha: 204))
          }
          radialGradient(cy: cy + (height*0.9),
                         fy: cy + (height*1.1)+20,
                         radius: radius) {
            stop(offset: 0.0f, color: specular)
            stop(offset: 0.8f, color: specular.derive(alpha: 0))
            transforms{ scale(y: 0.5) }
          }
          radialGradient(fit: false, radius: height/1.4,
                         fx: radius/2, fy: radius/4){
```

❶ **Must extend from CustomNode**

❷ **Define observable property**

❸ **Must implement with custom drawing code**

❹ **Define shadow's circle and gradient**

❺ **Define sphere's circle and gradients**

```
                stop(offset: 0.0f, color: shine.derive(alpha:0.5))
                stop(offset: 0.5f, color: shine.derive(alpha:0))
            }
        }
    }
  }
 }
}
```

These are the code highlights. First, every custom node must extend the `CustomGfxNode` class ❶. Second, creating observable properties on custom nodes is similar to creating observable properties on `Model` classes ❷. The only difference is the usage of `@GfxAttribute` instead of `@Bindable`. Third, every custom node must have drawing code that renders the node ❸; you'll find that you can use the same nodes as if they were regular drawings—even other custom nodes! Then the gradients are created at ❹ and ❺. In particular, the second of these ❺ shows a unique feature: multipaints. If it were not for multipaints, which are a series of paint instructions all applied to the same shape, you'd have to define a circle for each gradient. This would complicate matters, as some of those gradients are scaled and transformed according to the base circle. It's easier to calculate the transformations this way.

12.3.5 Additional builders

There are more builders and plugins than the ones we just explained, and it's possible that more builders have been added since this book saw print. You'll find a detailed list of available builders at http://griffon.codehaus.org/Builders.

A quick survey of the builders that were left out of the book's scope follows.

FLAMINGOBUILDER

The Flamingo (http://java.net/projects/flamingo/) component suite was created and is maintained by Kirill Grouchnikov (www.pushing-pixels.org)—the same mastermind behind the Substance look and feel project that makes Swing applications look great. Perhaps the most interesting component found in this suite is JRibbon, a pure Java implementation of the Ribbon component found in Microsoft Office 2007. There's also SVG support for creating icons and other goodies that complement the Ribbon component.

MACWIDGETSBUILDER

MacWidgets (http://code.google.com/p/macwidgets) is a project started by Kenneth Orr, whose aim is to provide components that follow Apple's Human Interface Guidelines. They're 100 percent Java but blend seamlessly with Apple's look and feel, even when not running on Apple's OS X! That's right—with MacWidgets you can create an application that looks like a native OS X application but runs on Windows or Linux. How crazy is that? Amazing!

TRIDENTBUILDER

Trident (http://kenai.com/projects/trident/pages/Home) is a general-purpose animation library created and maintained by the powerhouse that is Kirill Grouchnikov.

Trident takes off where Chet Haase's Timing Framework ends, and then adds many more things. Trident follows Timing Framework in spirit, but it isn't based on the latter's codebase at all. Kirill has put a lot of effort into this animation library. It's small and practical, to the point that it's now the one used by both Substance and Flamingo to drive their animation needs. Be sure to follow Kirill's blog and watch some of the videos related to Trident, Flamingo, and Substance that he has created over the years (http://vimeo.com/kirillcool).

SwingXtrasBuilder

This is a collection of projects that don't warrant a builder of their own because their individual sets of components are rather small. Here you'll find xswingx, which we discussed earlier in this chapter. There's also L2FProd Commons (http://l2fprod.com/common), providing task panes, an outlook bar, and a properties table. SwingX's task panes are based on L2FProd's—the code was contributed from one project to the other. There's also Balloon tip (http://java.net/projects/balloontip/). If you ever wanted to have a friendly pop-up like the ones you see appearing on your operating system's task bar, then the components provided by Balloon tip are the way to make it happen.

AbeilleForm Builder

This builder was discussed briefly in chapter 4. It's the workhorse behind the Abeille Forms plugin. You may recall that Abeille Forms Designer is an open source alternative for designing forms and panels that relies on JGoodies FormLayout or the JDK's GridBagLayout to place components on a Swing container.

12.4 Summary

In this chapter, you learned how to enhance a view script. You can do it by adding new nodes, for which there are also a number of alternatives: there's the direct approach of registering a node factory on a view script; then there's the more reusable option of using an implicit addon, which has the advantage of reusable nodes across scripts; and finally, there's the "make your own builder" option, which gives you the advantage of using nodes outside of Griffon applications.

Then we reviewed the different strategies `FactoryBuilderSupport` exposes to customize the node build cycle without needing to modify existing factories. There are four types of delegates that come into play at specific points of the build cycle. Each one can complement the others, depending on what your needs are.

Last, we surveyed the list of extensions in the form of builders. There are many third-party component suites out there. These builders reduce the time you'd have to spend hunting for custom components; they also reduce your learning curve, because they follow the same conventions as SwingBuilder.

In the next chapter, we'll unite Griffon and Grails to build a Griffon Bookstore application that uses Grails web services as a persistence mechanism.

13 Griffon in front, Grails in the back

This chapter covers

- Building a Grails server application
- Building a Griffon UI
- Connecting Grails and Griffon via REST

It's hard to find a web developer these days who hasn't come across an Ajax- or RIA-powered website. These technologies have become so ubiquitous that we can't go back to the times when Web 2.0 didn't exist. There are myriad options for building a web application that has Ajax built in or that presents a rich interface, in both the frontend and backend tiers. Grails happens to be one of the front runners when dealing with the JVM.

We've mentioned Grails a few times already in this book. If you're a developer working on web applications and you haven't given Grails a try, you owe it to yourself to do so. We can guarantee you won't be disappointed.

Grails is a full-stack web development platform whose foundations lie in Spring and Hibernate, so it shouldn't be hard for a Java developer to pick it up and get to work. But what really makes it revolutionary is its choice of default development language: Groovy, the same as in Griffon.

Grails has another ace up its sleeve: a ready-for-business command tool that's also extensible via plugins. It's thanks to this plugin system that building a Grails

application is a breeze. Need a way to search through your data? Install the Searchable plugin. Your requirements ask for CouchDB instead of a traditional SQL store? No problem, install the CouchDB plugin. What's that? You need to protect certain parts of an application using security realms? The Shiro plugin is here to help. You get the idea.

Out of the immense set of Grails plugins (by the team's current count, it's over 500), you'll find a good number that deal with Ajax and RIAs. They're pretty good. But no matter which one you pick, there will be times when you require a feature that can't be implemented because of a browser limitation. That's when it's time to look outside of the browser window at the space that allows you to run the browser. Yes, that's your computer's desktop environment. This is where Griffon comes in.

In this chapter, you'll see how to take advantage of Grails' powerful features to build a backend, in literally minutes, followed up by building a frontend with Griffon. The trick is finding a proper way to communicate between the two ends. We'll show you one of the many options you can use to connect a Grails server application with a Griffon desktop application.

First, though, you need to set up your environment, starting with Grails.

13.1 Getting started with Grails

Setting up Grails is as easy as setting up Griffon:

1. Point your browser to http://grails.org/Download, pick the latest stable release zip, and unpack it in the directory of your choice.
2. Set an environment variable called `GRAILS_HOME`, pointing to your Grails installation folder.
3. Add GRAILS_HOME/bin to your path. In OS X and Linux this is normally done by editing your shell configuration file (such as ~/.profile) by adding the following lines:

   ```
   export GRAILS_HOME=/opt/grails ex-
   port PATH=$PATH:$GRAILS_HOME/bin
   ```

   In Windows you'll need to go into the System Properties window to define a `GRAILS_HOME` variable and update your path settings.

Done? Perfect. You can verify that Grails has been installed correctly by typing `grails help` in your command prompt. This should display a list of available Grails commands, similar to the following:

```
$ grails help
| Environment set to development.....

Usage (optionals marked with *):
grails [environment]* [target] [arguments]*

Examples:
grails dev run-app
grails create-app books
```

This will confirm that your `GRAILS_HOME` has been set correctly and that the `grails` command is available on your path.

With installation out of the way, you're ready to start building a Grails application.

13.2 Building the Grails server application

We'll pick the familiar book/author domain because of its simplicity. Creating a Bookstore application is done with a simple command:

```
$ grails create-app bookstore
```

This command will create the application's structure and download a minimum set of plugins if you're running Grails for the first time.

That command looks oddly similar to Griffon's, doesn't it? Remember that Griffon was born as a fork of the Grails codebase. You'll put that claim to the test now. Besides some unique concepts to Grails and Griffon, almost all commands found in both frameworks provide the same behavior.

You can run the application now, but because it's empty you won't see anything of use. Next you'll fill it up a bit.

13.2.1 Creating domain classes

Domain classes reveal some of the differences between Grails and Griffon. One of the big differences is that Grails domain classes have access to Grails' object relational mapping (GORM) implementation, and Griffon doesn't provide that functionality by default.

Create two domain classes: `Book` and `Author`. Make sure you're in the application's directory before invoking the following commands:

```
$ grails create-domain-class Author
$ grails create-domain-class Book
```

This set of commands creates a pair of files under grails-app/domain/bookstore, aptly named `Author.groovy` and `Book.groovy`. These two classes each represent a domain object of your domain model. They both are empty at the moment. By contrast, Griffon doesn't support domain classes out of the box. This is a concern that's left to plugins, as domain classes aren't a generic attribute of all Griffon applications.

Go ahead and open the two classes in your favorite editor, or, if you prefer, in your favorite IDE. Grails isn't picky and will gladly work with any IDE or editor you throw at it. Make sure that the contents of the `Author` and `Book` domain classes match the ones in the following listings.

First, let's look at the Grails Author domain (grails-app/domain/bookstore/Author.groovy):

```
package bookstore
class Author {
    static hasMany = [books: Book]
    static constraints = {
        name(blank: false)
```

One-to-many relationship

```
        lastname(blank: false)
    }

    String name
    String lastname

    String toString() { "$name $lastname" }
}
```

Now let's look at the Grails Book domain (grails-app/domain/bookstore/Book.groovy):

```
package bookstore
class Book {
    static belongsTo = Author
    static constraints = {
        title(unique: true)
    }

    String title
    Author author

    String toString() { title }
}
```

Without going into much detail, the `Author` and `Book` classes define a pair of entities that can be persisted to the default data store. If this is the first time you've encountered a Grails domain class, don't worry, they don't bite. Besides the simple properties in each class, you'll notice that there's a one-to-many relationship from `Author` to `Book`. You could make it a many-to-many relationship to more closely reflect the real world, but let's keep things simple for the moment.

There's another step that must be performed before you attempt to run the application for the first time. You must expose the domain classes to the user in some way.

13.2.2 Creating the controllers

With a framework other than Grails, you'd need to write HTML or use some other templating mechanism to expose your domain classes. With Grails you can let the framework take over, as long as you stick to the conventions. If you only create a pair of controller classes, one per domain class, nothing else needs to be done. Hurray for scaffolding!

Go back to your command prompt, and type the following:

```
$ grails create-controller Author
$ grails create-controller Book
```

Locate each controller under grails-app/controllers/bookstore and edit it, carefully copying the following code into the appropriate file.

First, here's the Grails Author controller (grails-app/controllers/AuthorController.groovy):

```
package bookstore
class AuthorController {
    static scaffold = true
}
```

Next, the Grails Book controller (grails-app/controllers/BookController.groovy):

```
package bookstore
class BookController {
    static scaffold = true
}
```

That's all you need for now. Don't be fooled, though—there's a full-blown Spring MVC-powered component behind each of these controllers.

Perfect. You're good to go.

13.2.3 Running the Bookstore application

Run the application with the following command:

```
$ grails run-app
```

After a few seconds, during which the command compiles and packages the application, you'll be instructed to visit the following address: http://localhost:8080/bookstore. Use your favorite browser to navigate to that URL. You should see a page that looks like the one in figure 13.1.

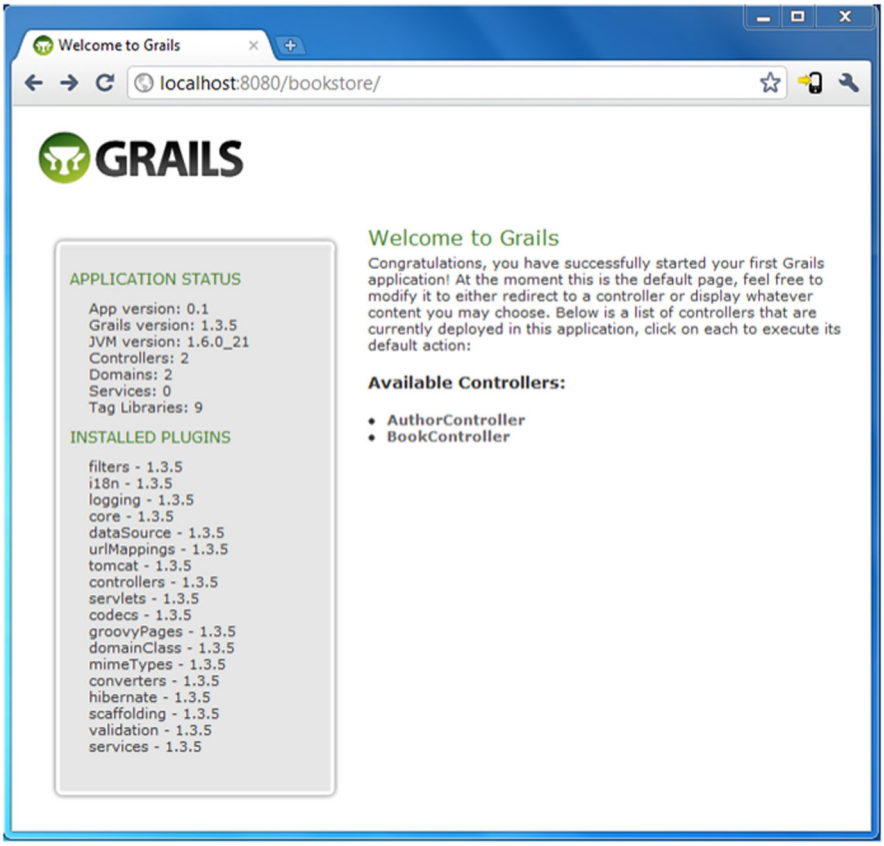


Figure 13.1 Bookstore webpage

You'll see a default page listing some internal details of the application, like the currently installed plugins and their versions. You'll also notice a pair of links that point to the controller you wrote.

Click the AuthorController link. You're now on the starting page for all create, read, update, and delete (CRUD) operations that affect an author. How is this possible? This is the power of conventions. When you instructed each controller that a domain class should be scaffolded, it generated (at compile time) the minimum required code and templates to achieve basic CRUD operations on that specific domain class.

You can play around with authors and books now. Try creating and removing some. You may end up with a screen that resembles figure 13.2.

Now that you've mastered the basics, let's step it up a notch and expose the domain classes to the outside world using a REST API. This is where picking up a good remoting strategy to interface with the Griffon frontend pays off.

13.3 To REST or not

There's no shortage of options for exposing data to the wild. You could go with a binary protocol like RMI or Hessian, or you could pick a SOAP-based alternative. We'd argue that a REST style is perhaps the simplest one. It doesn't hurt that many Web 2.0 sites have chosen this style (or variants of it) to give developers access to the services they provide. As you'll see in just a few moments, exposing domain classes in a RESTful way with Grails is a piece of cake.

13.3.1 Adding controller operations

Love it or hate it, XML is a popular choice among Java developers for externalizing data. Another format that gained momentum in Web 2.0 is JSON. There are a couple of ways to produce and consume both formats in Grails, but let's pick JSON for its simplicity.

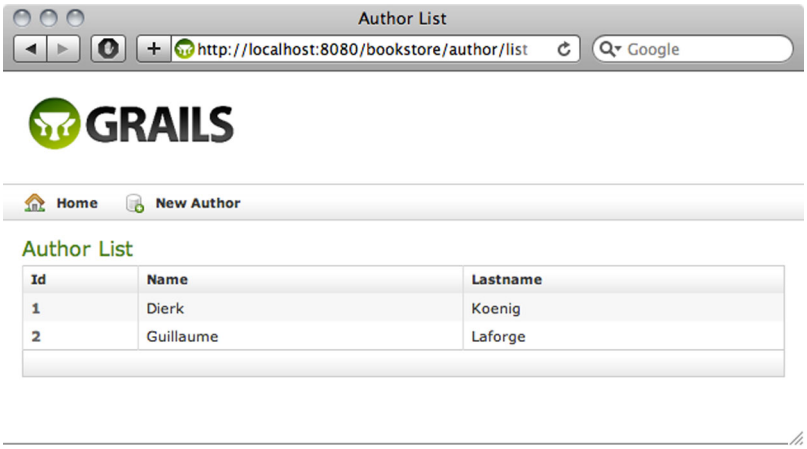


Figure 13.2 The Bookstore application showing a list of two authors that were created by clicking on the New Author link and filling in the generated form

What you want to do is expose each domain class with three operations:

- *List*—Returns a collection of all instances of the domain available in the data store
- *Show*—Returns a single instance that can be found with a specific identifier
- *Search*—Returns a collection of all domain instances that match certain criteria

All the results will be returned in JSON format.

Go back to `AuthorController`, and make the necessary edits so that it looks like the following listing.

Listing 13.1 RESTful implementation of `AuthorController`

```
package bookstore
import grails.converters.JSON
class AuthorController {
    static defaultAction = 'list'

    def list = {                                          <- List action
        render(Author.list(params) as JSON)
    }

    def show = {                                          <- Show action
        def author = Author.get(params.id)
        if(!author) {
            redirect(action: 'list')
        } else {
            render(author as JSON)
        }
    }

    def search = {                                        <- Search action
        def list = []
        if(params.q) {
            list = Author.findAllByNameLike("%${params.q}%")
            if(!list) list = Author.findAllByLastnameLike("%${params.q}%")
        }
        render(list as JSON)
    }
}
```

Gone is the default scaffolding, which has been replaced by specific actions that match the operations you want to expose to the outside world. The code is pretty straightforward, but you'll notice that there are calls to static methods on the `Author` class that aren't defined. These methods are added by the framework.

Remember we spoke about Groovy's metaprogramming capabilities? Well, here's proof that they're put to good use. All Grails domain classes possess the ability to invoke dynamic finder methods that magically match their own properties. For example, in the search action, you can look up all authors using a `like` query on its `name` or `lastname` property. These dynamic finder methods closely follow the operators and rules that you find in SQL.

One last point is that all results are returned in JSON format by using a type conversion. Grails will figure out the proper content type to send to the client by inspecting the format and data.

The updated `BookController` class looks similar to the author class, as shown in the next listing.

Listing 13.2 RESTful implementation of `BookController`

```
package bookstore
import grails.converters.JSON
class BookController {
    static defaultAction = 'list'

    def list = {
        def list = Book.list(params)
        render(list as JSON)
    }

    def show = {
        def book = Book.get(params.id)
        if(!book) {
            redirect(action: 'list')
        } else {
            render(book as JSON)
        }
    }

    def search = {                                    (1) Search action
        def list = []
        if(params.q) {
            list = Book.findAllByTitleLike("%${params.q}%")
        }
        render(list as JSON)
    }
}
```

Here, too, you'll find a dynamic finder on the `Book` class (1). This one operates on the book's `title` property. Summarizing the added behavior, you can search authors by `name` and `lastname`, and books can be searched by `title`. Both domain instances can be listed in their entirety and looked up by a particular identifier.

You must perform one final change before you can test the application.

13.3.2 Pointing to resources via URL

The REST style states that resources (domain info) should be available via a URL naming convention. There are many variations on the original guidelines, because REST doesn't impose strict rules on the conventions, so we'll pick one that's easily recognizable. The root of the URL path must be the application name; Grails takes care of that. The next element in the path will be the name of the domain class, followed by the action you want to invoke, with optional parameters. Here are some examples for the Author domain:

```
/author -> default action, which in our case lists all entities
/author/search -> calls the search action on authors
/author/42 -> fetches the author with id = 42
```

With these examples in mind, look for a file named `UrlMappings.groovy` located in `grails-app/conf`. Copy the contents of the following snippet into that file:

```
class UrlMappings {
    static mappings = {
        "/author/"(controller: 'author', action: 'list')
        "/author/search"(controller: 'author', action: 'search')
        "/author/list"(controller: 'author', action: 'list')
        "/author/$id"(controller: 'author', action: 'show')
        "/book/"(controller: 'book', action: 'list')
        "/book/search"(controller: 'book', action: 'search')
        "/book/list"(controller: 'book', action: 'list')
        "/book/$id"(controller: 'book', action: 'show')
        "/"(view:"/index")
        "500"(view:'/error')
    }
}
```

Given that you'll use the application as the data provider for the desktop application, it makes sense to start with some predefined domain instances, don't you think? Locate `BootStrap.groovy`, also located in `grails-app/conf`, and fill it with the contents of the following listing.

Listing 13.3 Adding some initial data to the application via `Bootstrap.groovy`

```
import bookstore.Author
import bookstore.Book
class BootStrap {
    def init = { servletContext ->
        def authors = [
            new Author(name: 'Octavio', lastname: 'Paz'),
            new Author(name: 'Gabriel', lastname: 'Garcia Marquez'),
            new Author(name: 'Douglas R.', lastname: 'Hofstadter')
        ]

        def books = [
            new Book(title: 'The Labyrinth of Solitude'),
            new Book(title: 'No One Writes to the Coronel'),
            new Book(title: 'Goedel, Escher & Bach'),
            new Book(title: 'One Hundred Years of Solitude')
        ]

        authors[0].addToBooks(books[0]).save()
        authors[1].addToBooks(books[1]).save()
        authors[2].addToBooks(books[2]).save()
        authors[1].addToBooks(books[3]).save()
    }
}
```

❶ **Injected methods**

You can appreciate a pair of new methods ❶ that you needn't write; the compiler and the Grails framework can inject them for you.

You can run the application now, but don't use your browser to navigate to the various URLs. Use a command-line browser like curl or Lynx, which makes for quicker debugging, or you could fire up groovysh if you don't have a command-line browser.

For example, the following command

```
$ curl http://localhost:8080/bookstore/author/1
```

results in the following output (formatted here for clarity):

```
{
    "class":"bookstore.Author",
    "id":1,
    "books":[{"class":"Book","id":1}],
    "lastname":"Paz",
    "name":"Octavio"
}
```

This command

```
$ curl http://localhost:8080/bookstore/book/search?q=Solitude
```

should give you the following results (also formatted for clarity):

```
[
    {
        "class":"bookstore.Book",
        "id":1,
        "author":{"class":"Author","id":1},
        "title":"The Labyrinth of Solitude"
    },{
        "class":"bookstore.Book",
        "id":4,
        "author":{"class":"Author","id":2},
        "title":"One Hundred Years of Solitude"
    }
]
```

TIP We highly recommend you pick up a Grails book, like the excellent *Grails in Action* by Glen Smith and Peter Ledbrook (Manning, 2009), if you found any of the steps so far to be a bit confusing. Those guys packed a lot of tips and tricks into that book, and it can help you get up to speed with Grails in no time.

You're done with the server side of the application. Time to look at the other half.

13.4 Building the Griffon frontend

You're back on familiar ground. Your target is to build an application that resembles what figures 13.3 and 13.4 depict. The first shows a tabbed view of the application's domain classes.

Figure 13.4 displays an elaborate search screen. A text box captures the search string, a check box specifies which domain the search will act upon, and a table displays the search results.

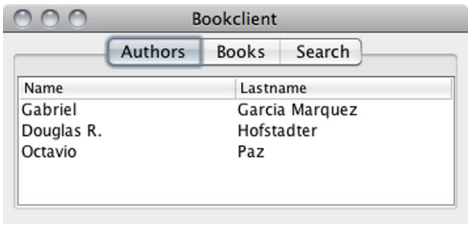


Figure 13.3 A tabbed view of all instances of Author domain classes after querying the Grails backend. The Books tab does the same for Book domain classes.

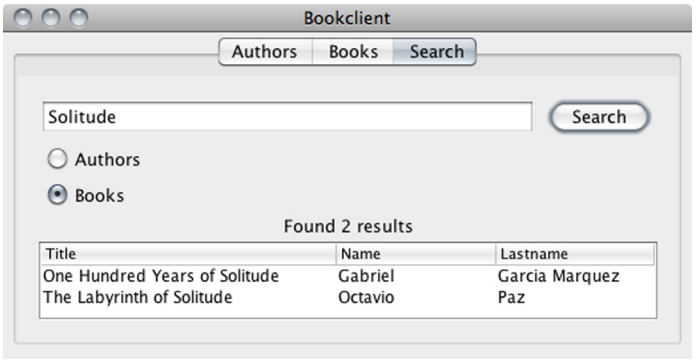


Figure 13.4 The Search tab for the Bookclient application. Users can search the bookstore backend by querying Authors or Books.

Let's get started. First create an application named `bookclient`—you already know the drill:

```
$ griffon create-app bookclient
```

This gives you a shell of an application. Now you'll spruce up the views.

13.4.1 Setting up the view

Swing is a vast toolkit—you know that already. You can do many things with it, or you can frustrate yourself by using it alone. The Swing classes found in the JDK are good as a starting point, but they fall short of providing a modern user experience. And let's not talk about layouts, especially GridBagLayout (http://madbean.com/anim/totallygridbag). You'll install a few plugins right away.

MigLayout (http://miglayout.com) is the first on the list. It's been said that using this layout is like using CSS to position elements on a page.

Next on the list is Glazed Lists (http://publicobject.com/glazedlists), which simplifies working with lists, tables, and trees. It does so by providing a missing key from the JDK's `List` class: a `List` that produces events whenever its contents change.

You can install all of these plugins with the following commands:

```
$ griffon install-plugin miglayout
$ griffon install-plugin glazedlists
```

While you're installing plugins, you can install one that will allow you to query the server side via a REST client. The plugin is aptly named rest:

```
$ griffon install-plugin rest
```

Now let's turn back to the view script and modify it to make it look like the screens shown in figures 13.3 and 13.4. The next listing contains all the code that you need to write to get the UI working.

Listing 13.4 The view in all its glory

```
package bookclient
makeTableTab = { params ->
  scrollPane(title: params.title) {
    table {
      def tf = defaultTableFormat(columnNames: params.columns)
      eventTableModel(source: params.source, format: tf)
      installTableComparatorChooser(source: params.source)
}}}

application(title: 'Bookclient', size: [480, 300],
  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]) {
  tabbedPane {
    makeTableTab(title: 'Authors', columns: ['Name', 'Lastname'],
                 source: model.authors)
    makeTableTab(title: 'Books', columns: ['Title'],
                 source: model.books)
      panel(title: 'Search') {
        migLayout(layoutConstraints: 'fill')
        textField(columns: 30, text: bind('query', target: model))
        button('Search', actionPerformed: controller.search,
               enabled: bind{ model.enabled }, constraints: 'wrap')
        buttonGroup(id: 'choice')
        radioButton('Authors', buttonGroup: choice,
                    selected: true, constraints: 'wrap',
                    actionCommand: BookclientModel.AUTHORS)
        radioButton('Books', buttonGroup: choice, constraints: 'wrap',
                    actionCommand: BookclientModel.BOOKS)
        label(text: bind{ model.status },
              constraints: 'span 2, center, wrap')
        scrollPane(constraints: 'span 2, growx, growy' ) {
          table {
            def columns = ['Title', 'Name', 'Lastname']
            def tf = defaultTableFormat(columnNames: columns)
            eventTableModel(source: model.results, format: tf)
            installTableComparatorChooser(source: model.results)
}}}}}
```

1 Reusable table tab-building code

2 Nodes contributed by Glazed Lists

3 Create tab

2 Nodes contributed by Glazed Lists

Recall that you can define any Groovy construct within a view script, because a view script is also a valid Groovy script. That's why you'll spot a closure at the beginning 1. This closure will be used to build the first two tabs 3, as their construction is identical; they only differ in the data source that feeds them. The third tab is a bit more elaborate than the other two.

Every child node of a `tabbedPane` must have a `title` property; that's how the `tabbedPane` knows what name should be used for the tab. You can easily spot the `title` properties on each tab and their values (Authors, Books, and Search). The first and second tabs make use of nodes provided by the Glazed Lists plugin ❷.

These nodes build a `TableModel` out of some sort of data source (which will be revealed to be a `List` that produces events, also from Glazed Lists). The model is then added to its parent table. Finally, a sorting element is added to the table. Clicking on the table headers will sort the data accordingly.

Onward to the third tab. The main node is a `panel` whose title is Search. Inside this `panel` is a `migLayout` definition. All elements inside the `panel` will be attached to it using MigLayout's settings. The six visible elements inside this `panel` are listed in table 13.1.

Table 13.1 The Search tab's visible elements

Element	Purpose
Text field	The `text` property is bound to a model property named `query`.
Button	Triggers the controller's search action.
Two radio buttons	Specifies on which domain the search should be performed.
Label	The text is bound to a model property named `status`.
Table wrapped in a `scrollPane`	Displays search results.

The second table also relies on nodes ❷ provided by the Glazed Lists support found in Griffon. These nodes operate by following a naming convention to name the properties in each element of the source list that feeds the table. We'll soon come back to how these conventions are put to work. For now, make a mental note of the names of the columns used to build each of the `tableFormat` nodes.

You can't run the application just now. You're missing a few properties on the model and the definition of the search action on the controller. You'll get a nasty runtime exception if you attempt running the application at this stage. You'll fill those holes next.

13.4.2 Updating the model

You might have noticed in listing 13.4 that the view expects a couple of properties to be available in the model. A pair of constants must be defined in it as well. Some of those properties are expected to be some kind of list. But not any list implementation will do—you need a special one. An observable list, to be exact.

We mentioned earlier that the Glazed Lists base building block is a `List` implementation that can trigger events whenever its contents change in some way. Those events aren't only triggered when an element is added or removed from the list, but also when an existing element is updated internally. How cool is that?

The following listing shows all the code that you must write to get the model ready for this application.

Listing 13.5 The model with all the properties required by the view

```
package bookclient
import groovy.beans.Bindable
import griffon.transform.PropertyListener
import ca.odell.glazedlists.EventList
import ca.odell.glazedlists.BasicEventList
import ca.odell.glazedlists.SortedList
class BookclientModel {
    @PropertyListener(enabler)
    @Bindable String query
    @Bindable String status = ''
    @Bindable boolean enabled = false

    static final AUTHORS = 'author'
    static final BOOKS = 'book'

    EventList authors = new SortedList(new BasicEventList(),
        {a, b -> a.lastname <=> b.lastname} as Comparator)
    EventList books = new SortedList(new BasicEventList(),
        {a, b -> a.title <=> b.title} as Comparator)
    EventList results = new SortedList(new BasicEventList(),
        {a, b -> a.title <=> b.title} as Comparator)

    private enabler = { evt ->
        enabled = evt.newValue?.trim()?.size() ? true : false
    }
}
```

❶ Shortcut for change listeners

❷ Observable Glazed Lists

The model contains three lists, as expected. They will hold authors, books, and the search results ❷. There are also other properties needed for the bindings. You might remember the `@PropertyListener` annotation from previous chapters. If not, here's a quick reminder of its function: it's an AST transformation that generates a `PropertyChangeListener` around a closure or a closure field. In this case ❶, it turns out to be a private field found on the same class. Whenever the query property changes value, the enabler listener will be called.

You're almost done. The next and last step is to finish up the logic.

13.5 Querying the Grails backend

You've reached the point where you can connect the Griffon frontend with the Grails backend. It's the job of the controller (and perhaps of a helper service) to send REST calls to Grails in order to get a list of each domain class type and to execute the search queries that the user types.

Let's encapsulate all the network-related code—the REST calls—in a service.

13.5.1 Creating a service

Implementing the REST calls in a service allows you to keep the controller as a simple entity that collects data from the model and updates the view with new data obtained from the service and saved once more in the model. Type the following at your command prompt:

```
$ griffon create-service bookstore
```

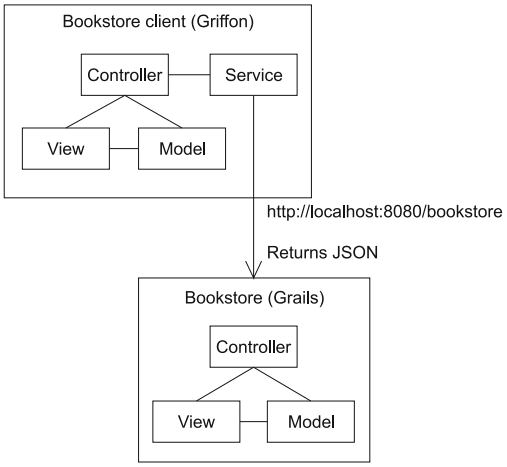


Figure 13.5 Griffon Bookstore client calling Grails Bookstore

This will create a service class named `BookstoreService` inside `griffon-app/services/bookstoreclient` (see the following listing). But you knew this already, didn't you? For a quick refresh on Griffon services, feel free to look back at chapter 5.

Listing 13.6 Bookstore client with all required REST calls

```
package bookclient
class BookstoreService {
  List searchAuthors(model) {
    withRest(id: 'bookstoreREST') {                          (1) Reuse REST client
      def response = get(path: 'author/search', query: [q: model.query])
      response.data.inject([]) { list, author ->
        author.books.id.collect(list) { bookId ->
          def book = model.books.find{it.id == bookId}
          [title: book.title, name: author.name, lastname: author.lastname]
  }}}}

  List searchBooks(model) {
    withRest(id: 'bookstoreREST') {                          (1) Reuse REST client
      def response = get(path: 'book/search', query: [q: model.query])
      response.data.collect([]) { book ->
        def author = model.authors.find{it.id == book.author.id}
        [title: book.title, name: author.name, lastname: author.lastname]
  }}}

  void populateModel(model) {
    withRest(id: 'bookstoreREST',                            (2) Set REST client for first time
      uri: 'http://localhost:8080/bookstore/'){
      def response = get(path: 'author')
      def authors = response.data.collect([]) { author ->
        [name: author.name, lastname: author.lastname, id: author.id]
      }
      execSync { model.authors.addAll(authors) }

      response = get(path: 'book')
      def books = response.data.collect([]) { book ->
```

```
            [title: book.title, id: book.id]
        }
        execSync { model.books.addAll(books) }
    }}
}
```

There are three service methods in this class. The first two ❶ will be used to search each of the domains given certain search criteria. If you recall from what you set up in the view, the user enters the search criteria on a text field, which is bound to a model property. In the implementation of each of the service methods is a call to a method named `withRest` that isn't defined by you. This method is provided by the REST plugin, and it's responsible for executing REST calls. The contents of this method are bound to an instance of `HTTPBuilder`, another handy builder that provides a higher-level API over Apache's HTTPClient. The third method ❷ will be used to populate the initial data during application startup.

You'll notice that the third method defines a URL that points to the server running the Grails app, whereas the other methods don't. It's your intention to call the third method first in order to set up the HTTPBuilder object and later reuse it for any subsequent queries. That's why there's an `id` property defined as well. When the `id` property is present, it means that a reference to the internal HTTPBuilder will be saved in an in-memory storage managed by the REST plugin. There's no need to pay a penalty for setting up a new HTTPBuilder for each query made, is there?

Finally, we can look at the controller now.

13.5.2 Injecting an instance of the service

The following listing shows all that there is to it. The controller relies on an injected instance of the service you just defined.

Listing 13.7 The controller and service working in unison

```
package bookclient
class BookclientController {
    def model
    def view
    def bookstoreService

    def search = {
        execInsideUISync {
            model.enabled = false
            model.status = ''
            model.results.clear()
        }

        String where = view.choice.selection.actionCommand

        try {
            List results = []
            switch(where) {
                case BookclientModel.AUTHORS:
                  results = bookstoreService.searchAuthors(model)
                  break
```

❶ Injected service instance

```
                case BookclientModel.BOOKS:
                  results = bookstoreService.searchBooks(model)
                  break
            }
            execInsideUISync {                                    ❷ Update view
                int count = results.size()                          inside UI thread
                model.status = "Found $count result${count != 1 ? 's': ''}"
                if(results) model.results.addAll(results)
            }
        } finally {
            execInsideUIAsync { model.enabled = true }
        }
    }

    def onStartupEnd = { app ->                                   ❸ Set up data before
        execOutsideUI { bookstoreService.populateModel(model) }     view is displayed
    }
}
```

All the pieces are finally coming together. And we don't mean just the Grails and Griffon part, but everything else that you've learned so far along the journey.

As a quick reminder of what we discussed in chapter 5, all services are automatically handled by Griffon as singletons, even if the Spring plugin isn't installed. These services will be automatically injected into MVC members by following a naming convention on the properties they expose. In this case, the controller has a property whose name matches the logical name of the service ❶.

Once the MVC group has been created, the application will switch from the startup to the ready phase. The controller reacts to that change by listening to an event ❸. During the handling of the event, it tells the service to load the data. This loading will occur outside of the UI thread, but the model will be updated inside the UI thread—the service has code to handle the latter case.

The controller's search action will be triggered once the user enters a query and clicks the Search button in the view. The query will be sent to the server outside of the UI thread once more, because that's the default setting for controller actions, as you might remember from chapter 7. Once the results come back, the controller updates the view via the model back inside the UI thread ❷.

Even though the description of the whole application takes a fair number of pages, the code takes just a few pages.

Now for the last piece you need to get this application running on its own.

13.5.3 Configuring the Bookstore application

The REST plugin will add dynamic methods to all controllers by default; after all, that's how it's configured. But you need those dynamic methods to be added to services instead. What can you do?

Back in the first chapter, we mentioned that Griffon encourages convention over configuration. This doesn't mean that configuration is completely gone. You can alter both the build-time and runtime configuration of a Griffon application. In chapter 2

we discussed the options at your disposal for configuration. Now's the time to put those claims to the test.

Locate the file `Config.groovy` inside griffon-app/conf. When you open it in an editor, you'll find logging configuration by means of a Log4j DSL. Add the following line to the file:

```
griffon.rest.injectInto = ['controller', 'service']
```

With this change, all REST dynamic methods should be added to both controllers and services alike. Go ahead, give it a whirl! Remember to have Grails running in the background; otherwise the REST calls will fail at startup. The following commands are enough to get the Grails backend running:

```
$ cd bookstore
$ grails run-app
```

Wait a few seconds, and then invoke the following commands at another command prompt:

```
$ cd bookclient
$ griffon run-app
```

Voilà! Run the application again, and you should be able to see the list of authors and books. You should also be able to query authors by name and last name. Feel free to play around with both applications. Maybe you feel like adding another search term or a third element to the domain model, like a publisher.

When the time is right, you'll need to package the applications and deliver them to your customers. Packaging in Grails is similar to Griffon—there's a specialized command that takes care of all the details. A Grails application can be packaged in a WAR file and then dropped into any JEE-compliant application server.

The command to be executed is aptly named `war`, and can be invoked like this:

```
$ grails war
```

Yes, it's that easy. After a while, you should see a WAR file that matches the name and version of the application stored in the default location, the target directory located at the root of the application's codebase. You might remember the command for packaging a Griffon application that you saw in chapter 10. Here it is again in its short form:

```
$ griffon package
```

This command will generate four packages: zip, jar, applet, and webstart. Pick whichever you think is best for your customers. Also remember that the Installer plugin is just a command invocation away; it provides more packaging targets that could be better for your needs.

That was quite a whirlwind ride, wasn't it? You might remember that we decided on a REST approach for these applications because of its simplicity, and we hope you agree that we accomplished the goal of keeping both applications simple. But REST is

just one of the many options you have at your disposal. Perhaps the most common would be a SOAP-based web service, as SOAP also facilitates communication between heterogeneous systems.

13.6 Alternative networking options

If SOAP is your game, you're in luck. Both Grails and Griffon have excellent support for SOAP! You only need to install the corresponding plugin and tweak the sources.

In the case of Grails, the recommended plugin is called xfire (http://grails.org/plugin/xfire). This plugin can expose a service using Apache XFire as the workhorse. The following snippet shows a simple example of its usage:

```
class SampleService {
    static expose = ['xfire']

    boolean myServiceMethod(String someValue) {
        someValue * 2
    }
}
```

The key to make this service available through a SOAP interface is in the `static expose` property. Notice that it takes a list of strings as its value. Though you only specified `xfire` as the single element for the time being, it's important to remember that you can define more values; you'll see when and why in a bit.

The Griffon plugin counterpart is Wsclient (http://artifacts.griffon-framework.org/plugin/wsclient). Like the REST plugin, this one will add dynamic methods that let components send a SOAP request. These methods are added by default to controllers, but you can change this preference via configuration in the same way you did before. Here's an example of how a Griffon controller could query the service:

```
String url = 'http://localhost:8080/exporter/services/sample?wsdl'
def result = withWs(wsdl: url) {
    myServiceMethod('griffon')
}
assert result == 'griffongriffon'
```

And that's all there is to it.

But your options don't stop with REST and SOAP. There are other formats and protocols for performing data exchange. Table 13.2 enumerates the plugins in both Grails and Griffon that can cover some of these additional options.

Table 13.2 Additional communication protocols supported by both Grails and Griffon

Grails	Griffon	Description
remoting	rmi	Java RMI protocol
remoting	hessian	Hessian/Burlap protocols by Caucho
xmlrpc	xmlrpc	XML-based RPC
protobuf	protobuf	Google's protocol buffers

Griffon goes a little further by supporting the following protocols and binary formats: Jabber, Avro, and Thrift.

13.7 Summary

Grails is by far the best option for building web applications in the JVM, enabling you to use features that can be found in popular Java libraries and features only found in the Groovy language. Griffon follows in Grails' footsteps and aims to provide the same productivity gains but in the desktop space. The two can be combined to build applications that touch desktop and server with the same approach to development: an approach aimed at high productivity and making programming fun again.

REST APIs are but one of the many options you can pick to allow both sides to collaborate with each other. Grails has other plugins that can expose domain objects and services via SOAP or remoting. Griffon similarly has plugins that can consume SOAP and remoting.

You got a good look at all the features offered by Griffon with a sample Bookstore application. We touched every default artifact provided by the framework. You installed a handful of plugins that enhanced the application's capabilities, either by providing new nodes to be used on views or dynamic methods ready to be called from controllers and services.

The application's life cycle made an appearance too, and you saw how to handle one of the many events it can trigger. You also tweaked the runtime configuration by editing one of the standard configuration files found in every application.

This exercise showed how closely related Griffon is to Grails, even though they target disparate running environments, such as desktop and web.

Now that you've had a taste of a more elaborate application, it's likely that the notion of tool support has come to mind. We've left the best for last: productivity tools and IDE integration will be the topics of the last stop on our journey.

14 Productivity tools

This chapter covers

- Setting up popular IDEs and Griffon
- Additional command-line tools

The poet John Donne once wrote, "No man is an island," reflecting on the fact that all of humanity is interconnected. The same reflection can be applied to our software tools and frameworks. In order to be really productive with one tool, you have to reach out to others. The Griffon framework is no different, and it's for that reason that it provides hooks for popular Java and Groovy tools to help you write, build, and deploy applications as part of a much larger ecosystem.

In this chapter, we'll look at popular software development tools such as IDEs. These tools let developers write, refactor, and debug applications. They even include build and deployment facilities. Not to be outdone by their visual brethren, command-line tools are more than adequate to build, package applications in specific environments, such as continuous integration servers, and even deploy such applications in a continuous delivery fashion.

We'll start with perhaps the most ubiquitous kind of tool that a Java developer will come across: IDEs.

14.1 Getting set up in popular IDEs

In the Java world, few start developing an application without the aid of an integrated development environment (IDE) or even a power editor. The advantages of such tools are clear to Java developers: the ability to refactor code without breaking the build, file history management, syntax highlighting, code completion and code suggestion, expandable macros, testing facilities, you name it. The bottom line is that IDEs make working with Java a less painful experience.

Because Groovy is closely related to Java, sometimes even being a substitute, you'd expect a similar degree of support for it in the same Java IDEs, which is the case, to varying degrees. At the time of writing, Groovy is on a clear path to becoming a first-class citizen in popular IDEs, such as Eclipse, IntelliJ IDEA, and NetBeans IDE. Power editors, such as jEdit and TextMate, aren't left behind. The latter is a popular choice for developers who like a certain fruity computer brand; it allows you to edit Groovy code and even manage Groovy-powered applications.

Let's begin with Eclipse.

14.1.1 Griffon and Eclipse

We'll discuss Eclipse first because it's the dominant IDE on the market by a large margin, judging by its install base. We'll cover the basic steps involved in getting a Griffon application up and running. The first step is installing the Groovy Eclipse plugin.

Installing the Groovy Eclipse plugin

The home page for this plugin can be found at http://groovy.codehaus.org/Eclipse+Plugin. You'll require this plugin if you're running a vanilla version of Eclipse or some other distribution that doesn't bundle the plugin.

> **NOTE** If you happen to have SpringSource Tool Suite (www.springsource.com/developer/sts), chances are that the Groovy plugin has already been configured in your settings. You can check the status of the Groovy plugin by bringing up the STS dashboard.

The plugin's documentation page provides a quick overview of what you need in order to install the plugin and what steps you need to perform. In most cases it's a simple matter of pointing your Eclipse instance to an update site, letting Eclipse figure out if additional dependencies must be met, and letting it download and install the plugin. You should be able to compile and run Groovy projects after a quick restart.

If that approach doesn't work for some reason, you can install the plugin the manual way. Locate the manual installation zip file on the Eclipse page, download it, and expand it inside the plugins directory in your Eclipse install directory.

Once that's done, you're ready to proceed with the next step—configuring your Groovy environment.

SETTING UP GROOVY AND GRIFFON

You must configure some environment settings in order to resolve classpath dependencies that point to your Griffon installation. Go to the Preferences window found on your Eclipse application. In the Java section locate Build Path and then Classpath Variables, as shown in figure 14.1. You must define two more variables: `USER_HOME` and `GRIFFON_HOME`. `USER_HOME` should point to the directory where your user account is located. For example, if your account name is `joecool`, the value should be similar to /home/joecool (Linux), /Users/joecool (Mac OS X), or C:\Documents and Settings\joecool (Windows). The `GRIFFON_HOME` variable should point to your Griffon installation directory.

You're all set up now for the next step: importing your Griffon application into Eclipse.

IMPORTING A GRIFFON APPLICATION

Importing a Griffon application is a three-step process.

First you must generate a pair of files that allows Eclipse to treat a Griffon project as an Eclipse one. In other words, you need to integrate the project with Eclipse. Fortunately, there's a Griffon command that relieves you of the burden of generating those files. Type the following at your command prompt to complete the first step, making sure your command prompt is already placed in the project's directory:

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

This generates a pair of files: .project and .classpath. The first identifies the project as an Eclipse project, whereas the second defines compile paths and libraries required by the project.

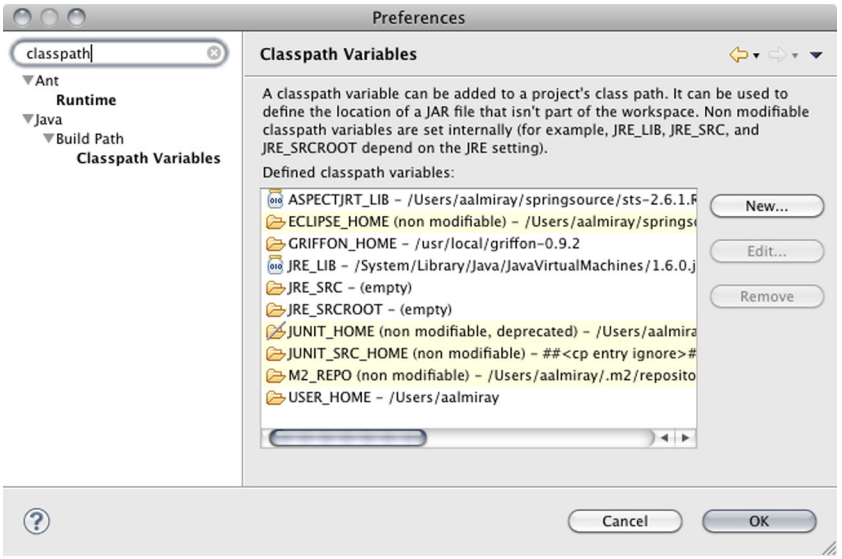


Figure 14.1 The Classpath Variables preferences dialog box, showing the configured variables after adding values for `USER_HOME` and `GRIFFON_HOME`

The second step is to install a plugin that allows you to keep the .classpath file up to date. Go back to your command prompt and type the following command:

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

Now the .classpath file will be up to date every time you install or uninstall a plugin. Remember to refresh the project inside Eclipse to get the latest version every time you install or uninstall a plugin.

Finally, the third step: importing the project into Eclipse. Bring up the Import dialog box and select the Existing Projects into Workspace option, as shown in figure 14.2.

Click the Next button and you'll be presented with the Import Projects page, as shown in figure 14.3. Type in or browse to select the directory that contains your Griffon application. You'll use a newly created application, named demo, as an example here. Figure 14.3 shows the page's state after typing in the root directory of a freshly created application with the name `demo`.

After this, nothing else needs to be configured. You can click the Finish button and let the wizard figure out the rest.

At this point, your Griffon application should be visible in the project explorer.

Running a Griffon application inside Eclipse

All your Groovy sources will automatically benefit from the Groovy support provided by the Groovy Eclipse plugin. This means you'll be able to see the structure of a Groovy class or script, block folding, syntax highlighting, and even call basic refactoring operations, as shown in figure 14.4.

All of this is great, but you'll surely want to test drive your Griffon application. You'll rely on Eclipse's Ant support, because at the time of writing there's no explicit Griffon support in Eclipse—this may change in the future.

The trick is letting Eclipse know about the Ant build file that should be located in your application's main directory. Hold on a second, there's no Ant build file yet! No

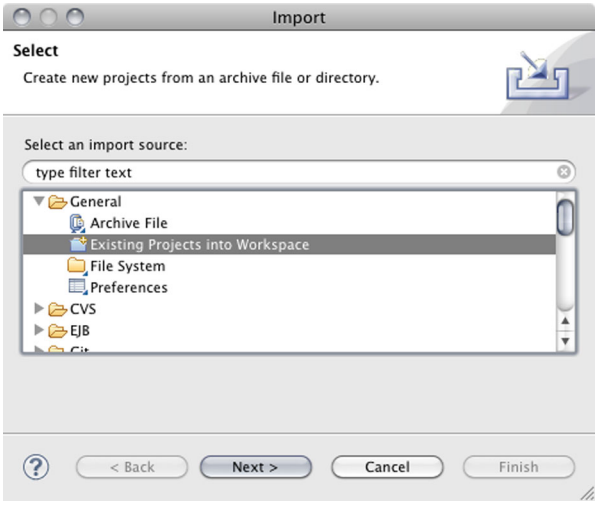


Figure 14.2 The Import dialog box showing the recommended option for importing a Griffon project into Eclipse

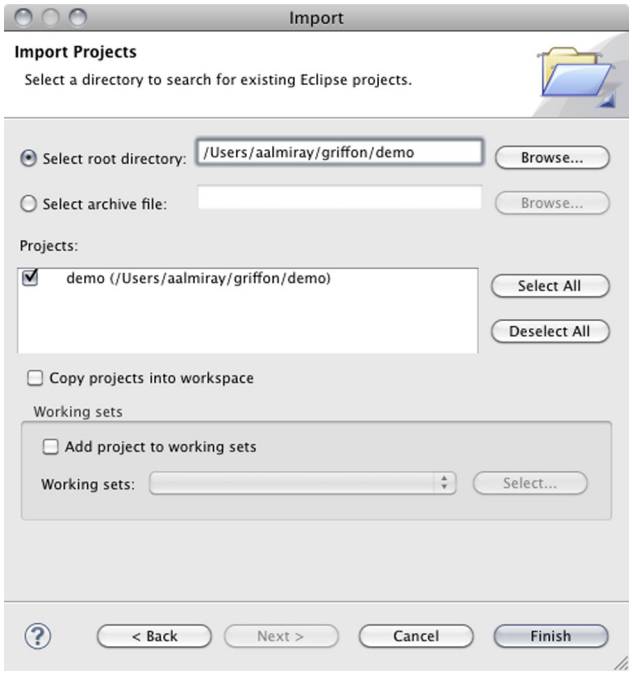


Figure 14.3 The last step for importing a Griffon project into an Eclipse workspace

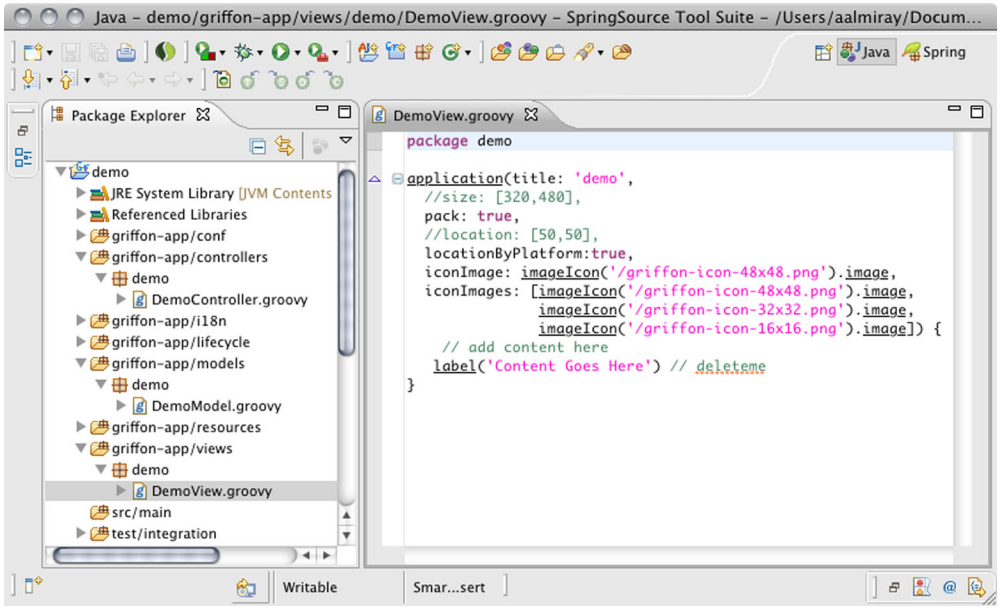


Figure 14.4 Eclipse, showing the structure of a Griffon application in the project explorer, as well as the class structure of the `DemoController`

worries—you can generate a suitable one by invoking the following command at the command prompt:

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

Now go back to Eclipse, and don't forget to refresh the project's contents! Open the Ant view if you don't have it open already: go to Window > Show Views > Ant. On that view, right-click to display a contextual menu, select the Add Buildfile option, locate your application's build file, and voilà! You should see a list of available targets in the Ant view, as shown in figure 14.5. Without further ado, double-click on any target, such as the run-app one, and after a few seconds your application should pop up. Notice that the target's output will be displayed in the Console view.

You're free now to explore what can be done with Eclipse. Remember that the Ant build file contains just a small selection of the targets that can be called. You can tweak it to meet your own needs.

We'll cover NetBeans next.

14.1.2 Griffon and NetBeans IDE

NetBeans IDE is another popular choice among Java developers. Though it was a latecomer in terms of Groovy support, that didn't prevent the NetBeans team from gaining ground at a tremendous pace, to the point where this IDE can be considered the second best when it comes to working with Groovy (and no, Eclipse isn't first on that list, yet).

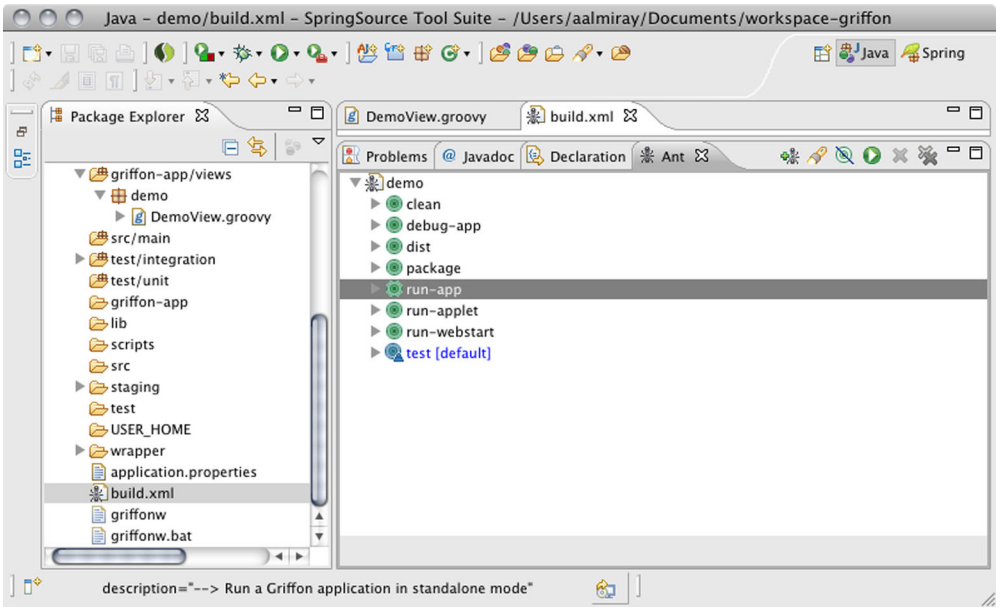


Figure 14.5 An Ant view of the application's targets, as shown by Eclipse's Ant support

Unlike Eclipse, a Groovy editor is a standard part of NetBeans IDE if you install the full version, meaning that you don't need to install any plugins to use Groovy in NetBeans IDE. In addition, there's a plugin that integrates the Griffon commands into NetBeans IDE, allowing you to create new Griffon applications right inside NetBeans IDE.

INSTALLING GROOVY AND GRIFFON IN NETBEANS IDE

From the NetBeans IDE download page (http://netbeans.org/downloads/index.html), download and install the latest version of NetBeans IDE, making sure that you get either the Java or the All download bundle. Once you've finished, you'll have an installation of NetBeans IDE that includes a Groovy editor, support for Grails, and many other features. If you have the Java edition, make sure you install the Groovy and Grails plugin before continuing with the next step.

Next, you need to install the Griffon plugin. This task is similar to what we described for Eclipse: you must tell the Plugin Manager to locate and install the Griffon plugin. To do so, first browse to http://mng.bz/48M6 and download the zip file with the latest version of the Griffon plugin. Back in NetBeans, go to Tools > Plugins. When the Plugins dialog box comes up, switch to the Downloaded tab and install all of the NBM files that are contained in the zip file you downloaded in previously. Figure 14.6 gives you an overview of what the Plugins dialog box should look like before the installation is finished.

Click on the Install button and, without needing to restart NetBeans IDE, you should be ready for business with Griffon in NetBeans IDE.

Now you're ready to create an application.

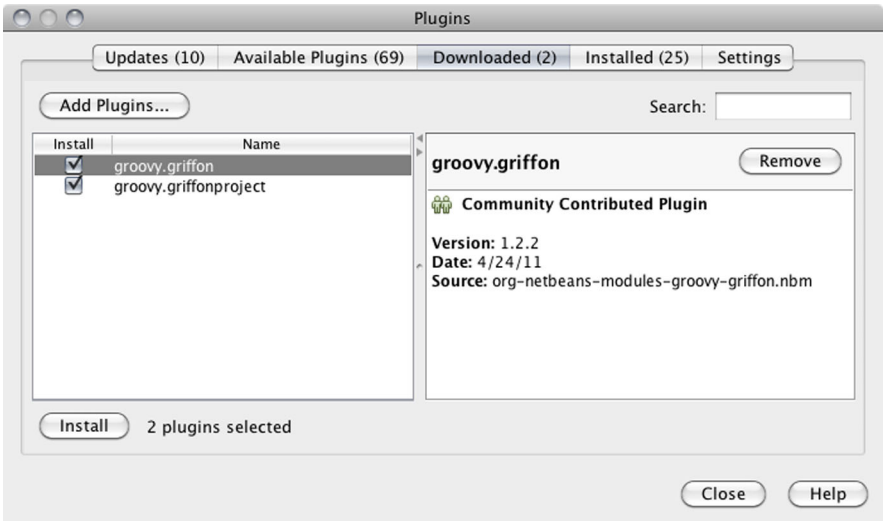


Figure 14.6 Both Griffon NetBeans modules are selected for installation.

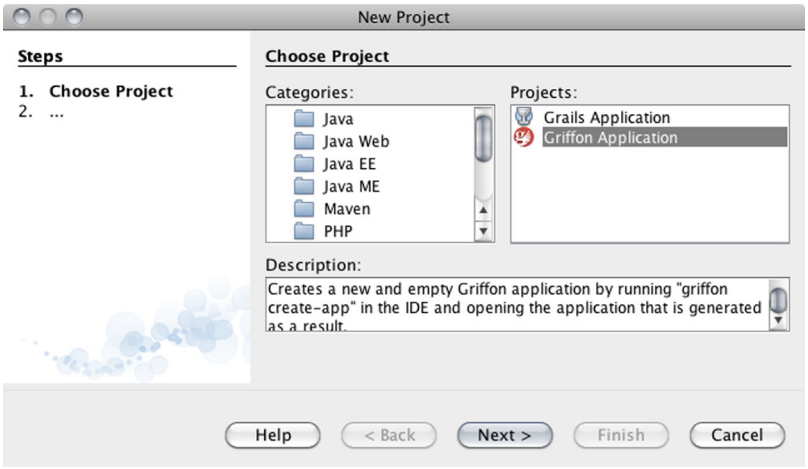


Figure 14.7 NetBeans' New Project wizard page with a project template selected, which is required for setting up a Griffon application

CREATING A GRIFFON APPLICATION

Unlike Eclipse, creating a Griffon application in NetBeans requires no previous configuration. Bring up the New Project dialog box, which has a Groovy category containing a Griffon Application project, as shown in figure 14.7.

On the next wizard page, you'll be asked for the location of your Griffon application. Type in or browse to select your application's main directory. If Griffon Home isn't set, you'll be able to set it in the wizard, as shown in figures 14.8 and 14.9.

From here you should be able to edit all your source files to your heart's content, using the outline view in the Projects window to work with your new Griffon application.

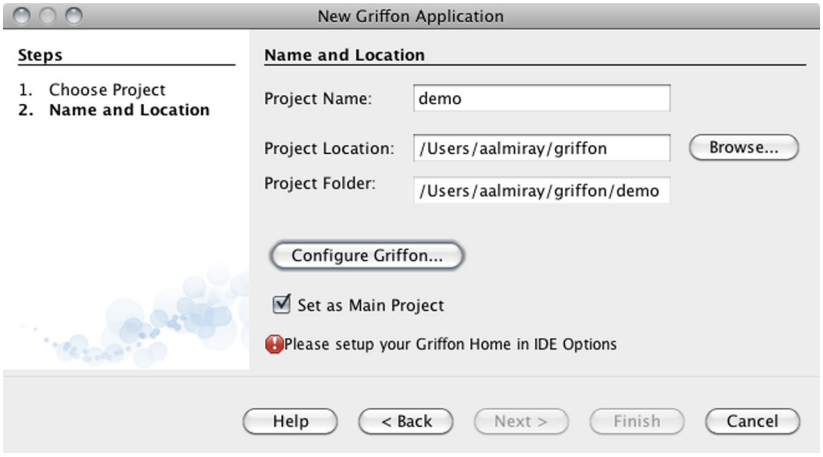


Figure 14.8 The second step of the New Griffon Application wizard

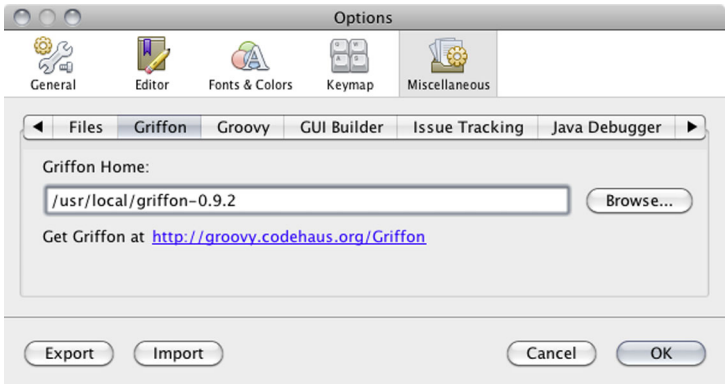


Figure 14.9 Configuring Griffon Home

NetBeans' Griffon plugin is aware of the Griffon conventions, so it shows each artifact in its own place, as you can see in figure 14.10.

The last bit of getting acquainted with NetBeans is running your application.

RUNNING A GRIFFON APPLICATION INSIDE NETBEANS

We've already given away the ending of this story. Switch back to the Projects tab, right-click the project, and choose one of the Run commands. Or select the project and click on the green arrow button on the toolbar. That was easy, wasn't it?

You can even call any Griffon command target available to the application by selecting Run Griffon Command from the context menu. This should pop up a dialog box that gives you a few choices. It even provides command name autocompletion—sweet! Figure 14.11 shows what the dialog box looks like after typing "create" in the command field.

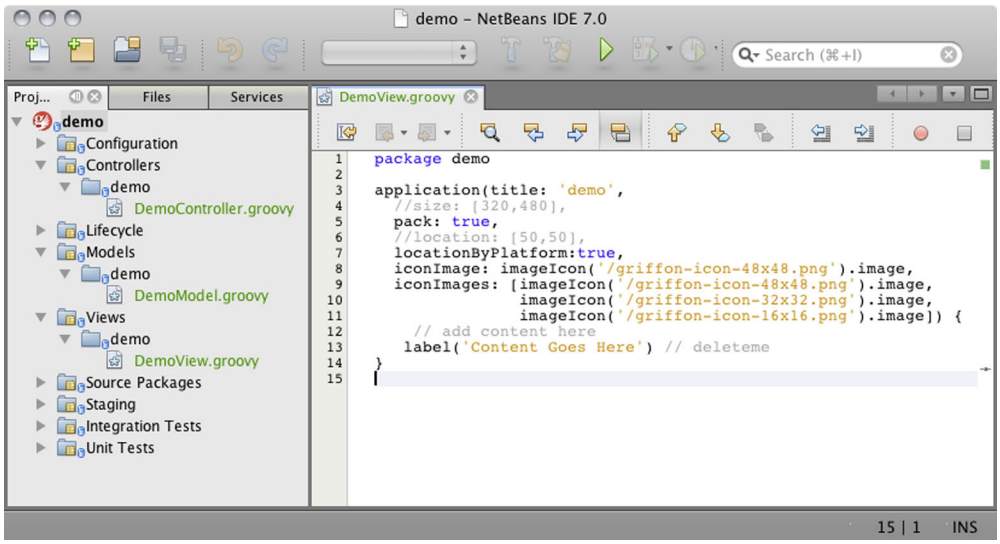


Figure 14.10 Project view with artifacts properly identified

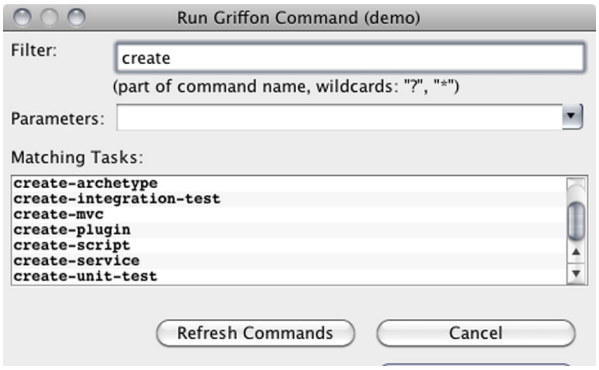


Figure 14.11 The Run Griffon Command dialog box, containing a list of suggestions based on what was typed in the Filter text field

A great thing about NetBeans' Griffon support is that it's able to open a Griffon project from the get-go. There's no need to generate integration files at all!

We're now up to the last IDE, which is IntelliJ IDEA. If you're wondering which Groovy IDE is the best around, you should look at this one.

14.1.3 Griffon and IDEA

IntelliJ IDEA is without a doubt the best Groovy IDE out there. JetBrains did a good job of quickly supporting Groovy, and they update it quite often. You can download a copy of IntelliJ IDEA from http://www.jetbrains.com/idea.

Since IDEA release 7 chances are that Groovy support is already included in the default set of plugins that come bundled in IDEA. Just to be sure, check that the JetGroovy plugin is installed by looking at the Plugin Manager. Make sure you have the Griffon plugin installed as well.

Creating a Griffon application

Once IDEA and the plugins are installed, you can safely invoke the New Project wizard, shown in figure 14.12. You'll observe that the last choice allows you to import an existing

Figure 14.12 IDEA's New Project wizard showing several options. The Griffon one is the last in the list.

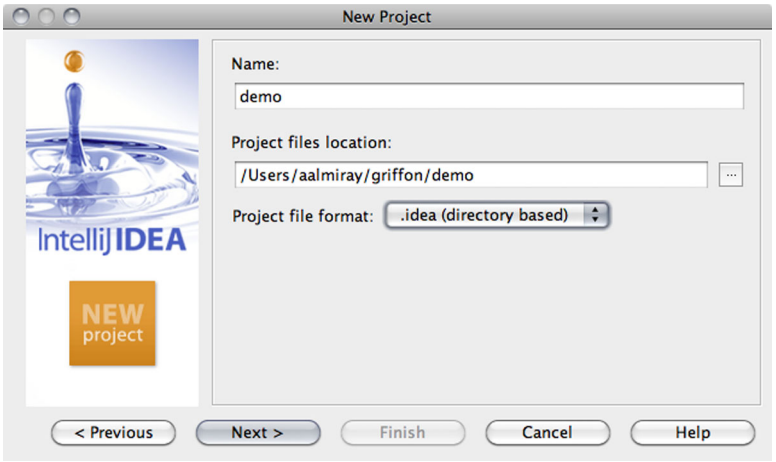


Figure 14.13 Configuring the demo application

Griffon project, which is great! Don't be fooled, though—this option also allows you to create a brand new Griffon project, which is exactly what you're going to do.

The next page of the wizard asks for the project name and its location. It also prompts you to select the Griffon SDK to use. If none are configured yet, you'll be asked to create one. Figure 14.13 shows the second wizard page, and figure 14.14 depicts the configuration of a Griffon SDK.

Click the Finish button once you've finished configuring the settings to your liking. IDEA should present a view similar to the one shown in figure 14.15. There you can see that each artifact has been identified by type, just as it happened in NetBeans.

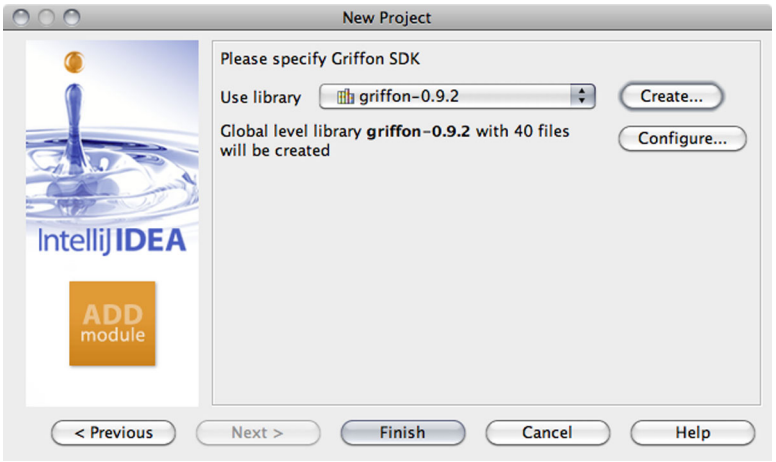


Figure 14.14 Selecting a Griffon SDK to be used with the demo application

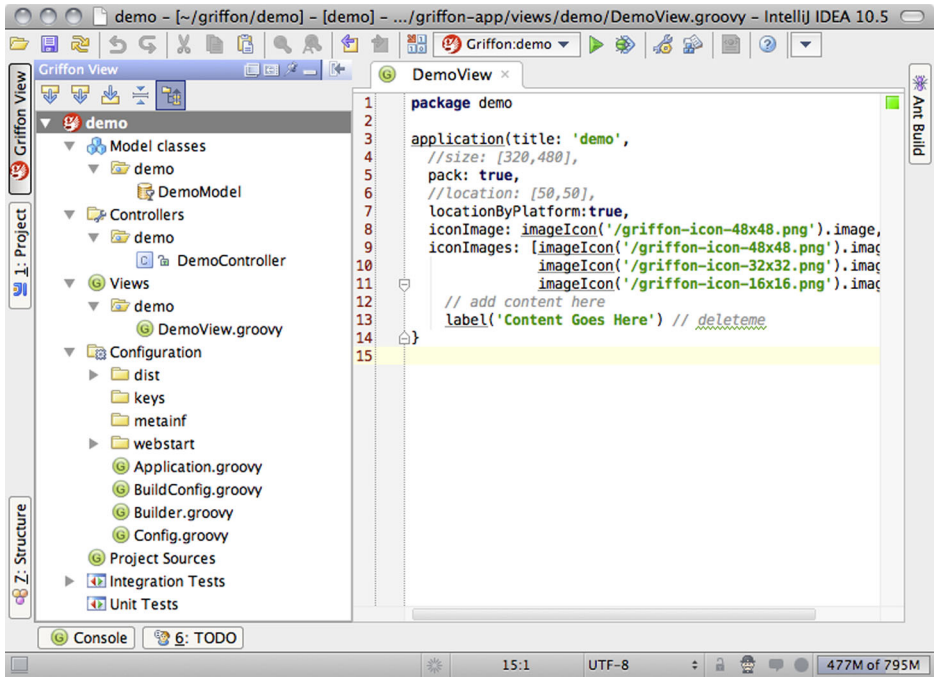


Figure 14.15 A Griffon project, as shown in IDEA's Griffon view

The Groovy editor provides syntax highlighting, code suggestions, and refactoring options, exactly what you'd expect from an IDE.

Running a Griffon application inside IDEA

Running a Griffon application in IDEA is a trivial task. See the green arrow at the center of the top toolbar, next to the project's name and the Griffon logo? Click on it, and the application will be launched in standalone mode, just as if you invoked the `run-app` command. In fact, IDEA invokes the `run-app` command for you, as you can verify by looking at the messages that appear in the output console.

IDEA also gives you the option to invoke any command. Right-click on the project name and select the Run Griffon Command option. This should bring up a dialog box similar to the one shown in figure 14.16. This dialog box prompts you for the command name and any arguments it may take. It even remembers previous invocations.

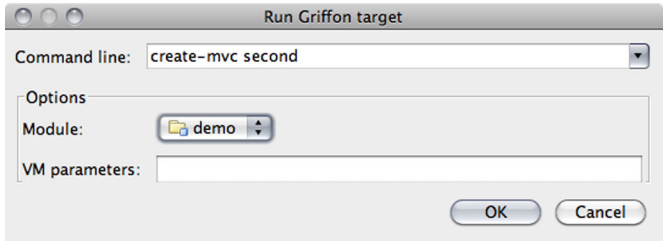


Figure 14.16 The dialog box for running any Griffon command. It takes the name of the command and any optional arguments as input.

In summary, all three major IDEs have good support for developing Griffon applications. Expect such support to be improved in each of these IDEs as new releases are published.

There's one more tool we'll cover, and though it's not a Java IDE per se, it's a powerful text editor, and a very popular one with Mac OS X users. Yes, we're referring to TextMate.

14.1.4 Griffon and TextMate

TextMate is a well-known text editor for Mac OS X, produced by MacroMates. You can download it directly from http://macromates.com. TextMate's functionality can by extended by installing new bundles. There are several bundles to be found out there, and there are a few ways to install them.

INSTALLING THE GRIFFON BUNDLE

Follow these instructions to install the Griffon bundle.

First, you need to create a directory where the bundle will reside. Go to your command prompt and execute the following command:

```
$ mkdir -p /Library/Application\ Support/TextMate/Bundles
```

The next step is easy if you have git installed. The popular GitHub website has some pointers on how to get it done; see http://help.github.com/ for reference. This step will also let you update the bundle effortlessly whenever a new version becomes available. Change into the directory you just created, and clone the bundle's repository using git, like this:

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

The bundle is now available. You can keep the bundle up to date by pulling the latest changes from the repository, as you'd do with any git repository:

```
$ cd /Library/Application\ Support/TextMate/Bundles
$ git update
```

If you don't have git installed on your system, there's an alternative. Point your browser to https://github.com/griffon/griffon.tmbundle. You'll notice that there's a button close to the top that lets you download a copy of the Griffon bundle repository in zip format; click on it. A file named something like griffon-griffon.tmbundle-749a7b6.zip should be downloaded. Don't worry if the numbers don't match—they're a reference to the last commit made to the repository. What's important is that now you have a snapshot of the whole bundle.

Now all you need to do is change into the bundle directory you created in the first step, unpack the file, and rename the directory created by unpacking it:

```
$ cd /Library/Application\ Support/TextMate/Bundles
$ unzip griffon-griffon.tmbundle-749a7b6.zip
$ mv griffon-griffon.tmbundle-749a7b6.zip griffon.tmbundle
```

Either way, you should now have a working Griffon bundle. Open TextMate, and inspect the Bundle menu: Groovy Griffon should appear in the list.

Follow the same steps to install the Groovy bundle. More details can be found at http://groovy.codehaus.org/TextMate.

Setting up the environment

Before you attempt to use the bundle, you must fix the path environment settings used by TextMate. Go to the Preferences menu, choose the Advanced options, and select the Shell Variables tab. Locate the entry named `PATH` and update its value. Make sure you enter the full path to the Griffon binaries. Figure 14.17 shows a typical setup.

It's time to give the bundle a try. Open any Griffon application, such as the demo application you used before. Figure 14.18 shows how TextMate presents the view and the rest of the project's contents in a sidebar.

The Groovy bundle provides syntax highlighting for all Groovy source files, as well as other features such as macros and code snippets that should let you write code in a flash.

Running a Griffon application inside TextMate

The main job of the Griffon bundle is to deliver a set of commands that can be invoked when the sources opened in the editor belong to a Griffon application.

Look at the Bundles menu again. Search for the Groovy Griffon menu item, and then select Commands. There are a few familiar names there, aren't there? You can run the application in any of the three development modes. Furthermore, there's an option to run any Griffon command available, including those provided by plugins and the application itself. You can also clean compiled artifacts or install a plugin from the editor.

Figure 14.19 shows the default commands that the bundle provided at the time of writing.

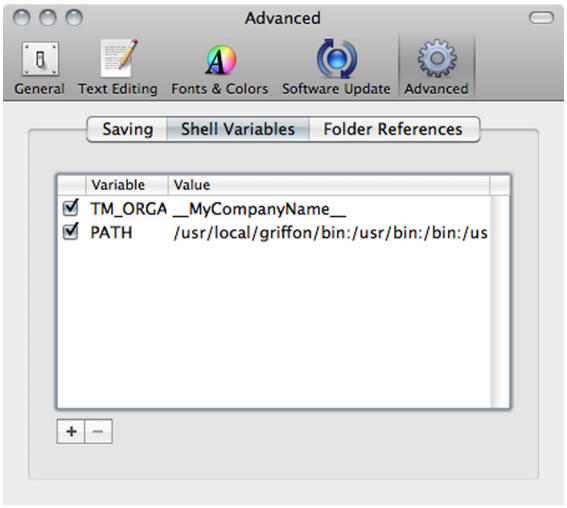


Figure 14.17 The `PATH` configuration, showing the full path of a Griffon installation being added at the beginning of the variable's value

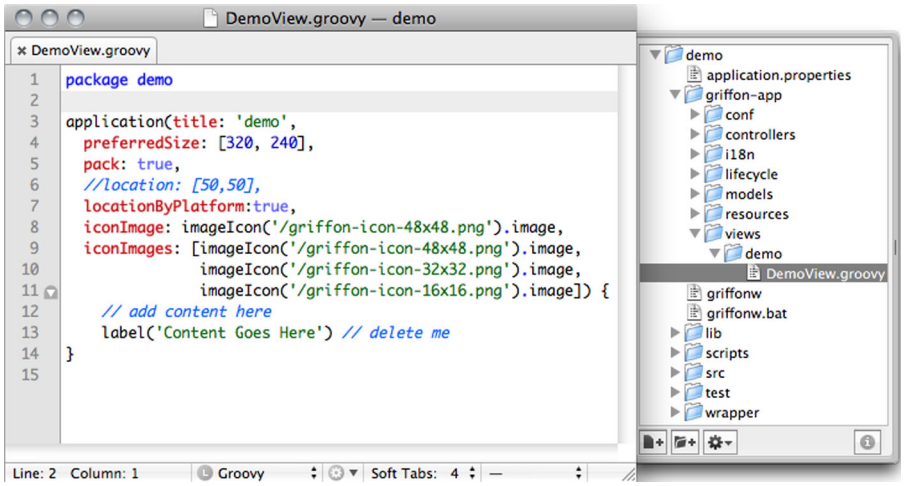


Figure 14.18 The view of the demo application. Notice that TextMate provides a detailed list of all files in a sidebar.

Select the Run App command. A new dialog box should pop up. It gives a view of the command output—output that should be familiar to you as it's the invocation of the `run-app` command applied to your application. Figure 14.20 shows a typical run of this command.

That's all the visual aids for now. But if command-line tools happen to be your cup of coffee, then no worries. We've got you covered with the next section.

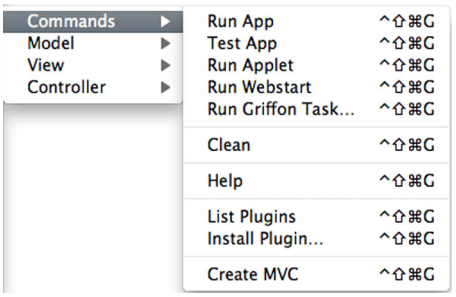


Figure 14.19 A list of default commands provided by the bundle. This list might be updated in future versions of the bundle.

14.2 Command-line tools

Way before Java IDEs and other visual tools emerged from the minds of their creators, we had the command line. This simple interface is the common denominator in all platforms. Tools that target the command line can be useful in situations where visual aids aren't reliable, might get in the way, or are too slow to get the job done. The following sections describe some of the common tools you'll encounter when dealing with the command line.

14.2.1 Griffon and Ant

Ant (http://ant.apache.org/) is perhaps the best-known command-line tool in the Java space, besides the Java compiler itself from the Java SDK. It's been around for many years now, and chances are that you've encountered an Ant build file sometime during your career. Despite its old age, some organizations still rely on Ant to do the job.

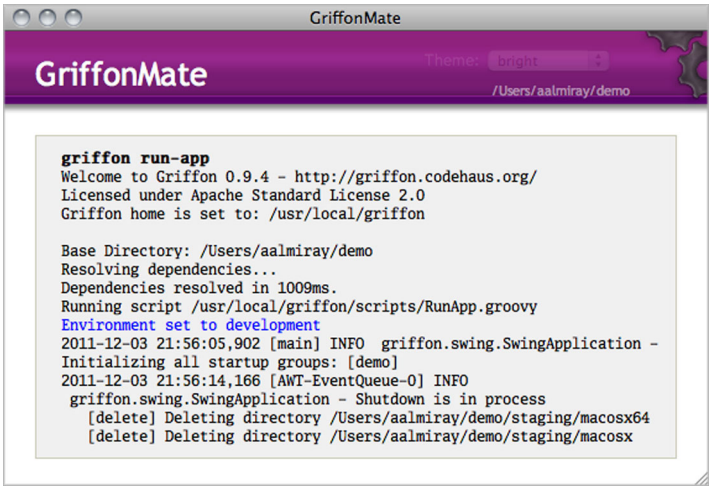


Figure 14.20 The output of invoking the Run App command from within the bundle. Notice that the output is the same as you'd get when invoking the command by regular means.

What happens if you're working with a team that relies on Ant for its build, and you must integrate a Griffon application into the mix? No problem, simply integrate it. You see, Griffon is aware of Ant's existence, and it understands Ant's conventions. Naturally it provides a way to generate a build script suitable for Ant.

If you read the Eclipse section earlier in this chapter, then you know what's going to happen. Execute the following command to integrate a Griffon application into an Ant build:

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

This command will generate a build.xml file with some Ant targets already configured. Running the `ant` command with the `-p` flag enabled will list all of the available targets, like this:

```
$ ant -p
Buildfile: /Users/aalmiray/demo/build.xml

Main targets:

 clean          --> Cleans a Griffon application
 debug-app      --> Run a Griffon application in standalone mode with
➥debugging turned on
 dist           --> Packages up Griffon artifacts in the Production
➥Environment
 run-app        --> Run a Griffon application in standalone mode
 run-applet     --> Run a Griffon application in applet mode
 run-webstart   --> Run a Griffon application in webstart mode
 test           --> Run a Griffon applications unit tests
Default target: run-app
```

Each one of those targets is backed by a specific Griffon command. For example, invoking `ant run-app` would be the same as invoking `griffon run-app`. Sweet!

You can create new Griffon-aware tasks too. Open the build file in your favorite editor, and paste in the following snippet:

```
<target name="zip" description="--> Zips the application">
  <griffon>
    <arg value="package"/>
    <arg value="zip"/>
  </griffon>
</target>
```

The new target should package the application using the zip deployment mode. Now, go back to your command prompt and invoke it:

```
$ ant zip
```

A few moments later, you should have the application packaged in zip mode, as expected. Be mindful that this setup requires you to have a valid JDK installed and a `JAVA_HOME` environment variable that points to the installation directory. The same rule applies for a Griffon installation and a `GRIFFON_HOME` variable.

14.2.2 Griffon and Gradle

Gradle (http://gradle.org/) is a newcomer compared to Ant, but it builds on the lessons Ant and Maven have learned. Gradle provides many advantages over Ant. For example, it has a full-blown object model of your project. Gradle can discern with 100% accuracy which tasks should be invoked and which can be skipped, for example. This decreases build time dramatically, as the tool won't waste time computing outputs for tasks that need not be run.

Gradle also follows the convention-over-configuration paradigm, meaning that a build file will contain only the configuration that deviates from the conventions set forth by the tool and any plugins applied to it. Furthermore, Gradle uses a real programming language (Groovy, in this case) instead of XML to describe what the build should do.

In order to integrate Griffon with Gradle, you need to use the Gradle Griffon plugin. You can either integrate an existing application or create a new one from the get-go.

INTEGRATING AN EXISTING APPLICATION

The process for integrating an existing application with Gradle is almost identical to the process for integrating an application with Ant. You'll use the same command, but with a different argument this time:

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

The output of this command is a `build.gradle` file that contains the minimal setup required by Gradle to build and run the application. At the moment of writing, the generated file looks like the following listing.

Listing 14.1 A default Gradle build file created by the Griffon command

```
buildscript {
    repositories {
        mavenCentral()
        mavenRepo url: 'http://repository.codehaus.org/'
```

```
        mavenRepo url: 'http://repo.grails.org/grails/core/'
        mavenRepo url: 'http://repository.springsource.com/maven/bundles/
     release'
        mavenRepo url: 'http://download.java.net/maven/2/'
    }

    dependencies {
        classpath('org.codehaus.griffon:griffon-gradle-plugin:1.1.0')
        classpath('org.codehaus.griffon:griffon-scripts:0.9.5')
    }
}

griffonVersion = '0.9.5'
version = '0.1'

apply plugin: 'griffon'

repositories {
    mavenCentral()
    mavenRepo url: 'http://repository.codehaus.org/'
}

dependencies {
    compile "org.codehaus.griffon:griffon-rt:$griffonVersion"
}
```

It's quite possible that by the time you give it a try, the template used to create this file will have been updated with additional tasks. Go ahead; take it for a spin by calling the `clean` command:

```
$ gradle clean
:clean
Resolving dependencies...
Dependencies resolved in 366ms.
Running pre-compiled script
Environment set to development
   [delete] Deleting directory /Users/aalmiray/demo/build/classes
   [delete] Deleting directory /Users/aalmiray/demo/build/plugin-classes
   [delete] Deleting directory /Users/aalmiray/demo/build/resources
   [delete] Deleting directory /Users/aalmiray/demo/build/test-classes
   [delete] Deleting directory /Users/aalmiray/demo/build/test-resources
   [delete] Deleting directory /Users/aalmiray/demo/staging

BUILD SUCCESSFUL
```

Good, it works. The plugin lets you invoke any Griffon command by prefixing `griffon-` in front of the command name. Say you'd like to run the application using Gradle—simply type the following at your command prompt:

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

It's useful as it is, but the Gradle support is in its early stages as we write this. It's likely that you'll experience better integration when you try it.

CREATING A NEW APPLICATION

We almost forgot to cover the other case regarding Gradle integration. You can bootstrap a new application by starting with just a Gradle build. See the code in listing 14.1? Save it in a new build.gradle build file, and call the following command:

```
$ gradle init
```

This will create the initial structure for a Griffon application. Of course, you shouldn't call it in an already existing application, lest you overwrite some files.

14.2.3 Griffon and Maven

What about Maven? It's also a very popular build tool among Java developers. Unfortunately, the Griffon team hasn't come up with definite support for Maven at the time of writing. It's possible that a Maven plugin providing Griffon integration will soon find its way into the Griffon toolbox. Keep your eyes peeled at the official Griffon website and mailing lists to find out more.

The tools we've reviewed so far have a wide range of features and differ in many areas, but they have one common attribute: they must be installed on your system before you can use them. What if you don't have the tool installed, not even Griffon, and you still want to build an application? Don't fret, there's an answer for that: the Griffon wrapper.

14.3 The Griffon wrapper

Quick question: How do you make use of a tool without having the tool at your disposal? You rely on another tool to reach the first, of course.

Imagine the following scenario. Bob and Alice are working on the same project, which happens to be a Griffon application. Alice has followed due diligence and has installed Griffon on her system and created the initial structure of the application, committed the source to a code repository, and sent a message to Bob letting him know he can grab a copy and continue the work.

Bob decided to skip setting up Griffon on his machine because he didn't want to mess with his environment settings. How will he be able to build the application? Alice, being a dutiful developer, made sure to include a few files that the `create-app` command added by default when the application was created. These are the files:

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

The first two are script files, and the last two are a binary and a configuration file. Combined, they form the Griffon *wrapper*. The wrapper lets you call any Griffon command as if you had a local Griffon distribution configured. As a matter of fact, one will

be configured for this particular project. With this tool, Bob is able to compile the application by typing the following at his command prompt:

```
$ ./griffonw compile
```

Next, he must create a new MVC group for the application. What should he do now? The answer is to invoke the `create-mvc` command target using the wrapper again:

```
$ ./griffonw create-mvc viewer
```

Excellent! Every command becomes available to Bob, thanks to the wrapper. Now extrapolate this scenario to a continuous integration server. You usually require administrative access to the server that hosts the CI environment in order to install a new tool. If you don't have the right access permissions, you might have to wait for IT to resolve the matter, which may take some time. But if your CI software lets you invoke an arbitrary command on the same sources you're about to build, then it's only a matter of configuring the Griffon wrapper. Your only requirement is to have a working JDK installed in the CI server, which should be there from the start.

14.4 Summary

Griffon provides its own toolchain by offering a command-line tool that has been discussed throughout this book. But there are other tools that can make a Java developer very productive when building applications. IDEs, visual editors, and additional command-line tools are among the most common choices.

Popular Java IDEs such as Eclipse, NetBeans IDE, and IntelliJ IDEA provide support for Griffon in various degrees. Most of them have an easy-to-follow setup. Once any of these tools has been configured, a developer should be able to move code around much faster, apply refactoring techniques, and invoke Griffon commands from within the IDE.

There are also other command tools, such as Ant and Gradle, that when used in combination can extend the reach of a particular build. These tools can permit a Griffon application to participate in a much larger build, for example.

Then there's the Griffon wrapper, the ultimate command-line tool for building Griffon applications without even having an existing Griffon distribution locally installed.

We're glad you stayed with us for the long ride. We hope you have enjoyed the journey as much as we did. We also hope you come back to these chapters when you need a quick recap of a particular feature or setting. And keep in mind that the Griffon framework continues to grow by means of its plugin system, not just by the features added to its core. Keep an eye on the plugin community.

There's no place on the desktop that Griffon can't reach. The sky is the limit!

appendix Porting a legacy application

You've seen how Griffon can be used to create desktop applications from scratch, and you've had a lot of fun while doing so. Sadly, all isn't fun and giggles when it comes to dealing with existing Java-based applications, because you can't go back in time and begin the project with Griffon as the starting tool. But as the saying goes, hope dies last, and there's hope for solving this problem. No, it's not a time machine, no matter how much we long for a TARDIS; rather, it's a set of tips and features that can ease the transition from a full Java Swing application to a Griffon-enabled one.

Let's begin with the visible aspect of an application.

Handling legacy views

You might recall that back in section 4.6, we discussed two options that allow you to adapt a legacy view into a Griffon application without changing the source of the existing view classes: the ability to wrap a Swing GUI Builder (formerly Matisse) view into a Groovy view script, or use the Abeille Forms Designer plugin if your views rely on AFD. Here's a quick summary of both approaches.

Swing GUI Builder views

A Swing GUI Builder view class is normally generated via templates. The most usual template creates a subclass of `JComponent`, registers a field for each visible component, and finally arranges all components using a complex set of instructions using GroupLayout's API. The Griffon script `generate-view-script` relies on these facts and generates a view script that exposes each of the visible components and returns the top-level container (the subclass of `JComponent`). This script expects the name

of the class to be converted, whose file should reside somewhere in the application's source directories. Where to put the file is up to you; we recommend src/main instead of griffon-app/views, because the class isn't a real Griffon view.

Here's how a wrapped view may look:

```
widget(new LoginDialog(), id:'loginDialog')
noparent {
  bean(loginDialog.usernameField, id:'usernameField')
  bean(loginDialog.passwordField, id:'passwordField')
  bean(loginDialog.okButton, id:'okButton')
  bean(loginDialog.cancelButton, id:'cancelButton')
}
return loginDialog
```

You can infer from this script that the top-level class `LoginDialog` is a subclass of `JComponent`. You can also infer that the four beans declared afterward are visible components that belong to said class. Notice the usage of the `noparent` node to avoid exposing the components to unsuspecting containers. Once all components are available in a view, you can apply them to the same techniques and tricks applicable for all other nodes, such as binding.

`generate-view-script` comes bundled with the default distribution; there's no need to install additional software to make it work. But you do need to add to your application whatever dependencies the legacy view class requires, such as grouplayout.jar or swing-application.jar.

Abeille Forms Designer views

Unlike the previous option, you do need to install additional software in order to work with AFD views. Fortunately, that software comes bundled as a Griffon plugin, which means installing it is just a command invocation away:

```
$ griffon install-plugin abeilleforms-builder
```

AFD views are stored in either XML or a custom binary format. Regardless of which one is used, the plugin should be able to expose the contents of a form. Given that form definitions aren't code, it's better to place them under griffon-app/resources; otherwise, use your best judgment depending on your preferences. The only constraint is that the form definitions must be available in the application's classpath.

Here's how the previous view might look with AFD. We also added a few customizations for binding and registering actions on the buttons:

```
formPanel("login.xml", id: 'loginPanel')
noparent {
  bean(model, username: bind{ usernameField.text })
  bean(model, password: bind{ passwordField.text })
  bean(okButton, actionPerformed: controller.loginOk)
  bean(cancelButton, actionPerformed: controller.loginCancel)
}
return loginPanel
```

The code is simple. It follows the same approach as the Swing GUI Builder view wrapper: exposing the top-level form as a container plus all of its children components.

But what happens if your view can't be wrapped with any of these options? No problem. Assuming the view class follows a design similar to the one used by Swing GUI Builder views, you can manually create a view script that exposes the top-level container and its children, using a combination of `widget`, `container`, and `bean` nodes. The last approach we'll cover is a full-blown Java view, no Groovy at all.

Custom Java-based views

If for some reason you feel the need to build a view in 100% Java or you're unable to use Groovy, don't worry; you can still take advantage of Griffon. For the first case, you only need to write a class that provides the same behavior as a Griffon view, including the hooks to the application's life cycle. If you're thinking already about how to accomplish such a task, then don't bother too much; it already exists, and here's how you can take advantage of it.

First you'll convert an existing Groovy-based view into a Java source file. Say, for example, that you have an existing application with an MVC group named `sample`. By convention, it will have a view script named `SampleView` located in the `sample` package in griffon-app/views. Armed with this knowledge, you can invoke the following command:

```
$ griffon replace-artifact sample.SampleView -type=view -fileType=java
```

Looking carefully at the command's arguments, you can expect the source file griffon-app/views/sample/SampleView.groovy to be replaced by another file with the same name but with .java as its extension. Its contents are also different from the original file. This is how the view looks after it has been converted from Groovy to Java:

```
package sample;

import java.awt.*;
import javax.swing.*;
import java.util.Map;

import griffon.swing.SwingGriffonApplication;
import org.codehaus.griffon.runtime.core.AbstractGriffonView;

public class SampleView extends AbstractGriffonView {
    // these will be injected by Griffon
    private SampleController controller;
    private SampleModel model;

    public void setController(SampleController controller) {
        this.controller = controller;
    }

    public void setModel(SampleModel 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(final Map<String, Object> args) {
        execSync(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("sample");
        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
        (SampleView.class.getResource(path));
    }
}
```

A quick survey of the code reveals that the new view extends from a particular class (`AbstractGriffonView`), which you can assume implements all the required behavior of a Griffon view. There are also properties that match the other two MVC members of the group, the controller and the model. You can delete those if they aren't needed. Next you find the hook into the application's life cycle. Recall from chapter 6 that the `mvcGroupInit` method is called by the application once all members of a group have been instantiated but before the building of the group has been completed. This is the perfect time to customize the view. Also of important note is the usage of the

`execInsideUISync` method to guarantee that the UI components are built inside the UI thread. This method is one of the threading options we discussed back in chapter 7. The default template suggests that you build the UI using a private method named `init`; this is a standard practice with generated code. The key aspect is that you're free to implement this method as you see fit.

XML-based views

There's one last alternative to building an UI declaratively: using XML. For some strange reason, XML has been the preferred go-to format for externalizing almost every aspect of a Java application since the early days. We won't engage in a discussion of whether XML is good or bad, but the moment you step outside of declarative programming by adding behavior, well … let's just say there are better ways to spend your time than battling the angle-bracket monster.

Java-based Griffon views come with a ready-to-use mechanism for dealing with XML. You may have noticed that the `AbstractGriffonView` class exposes a pair of methods named `buildViewFromXml`: one takes a map as a single argument, and the other takes a map and a string. Both methods will read an XML definition, but the first relies on the convention over configuration precept to determine the name of the file while the other takes the name as an argument. What could be the convention here? If you guessed the name of the view class, then you're on the correct path again.

Suppose you have a Java-based view whose full qualified class name is `com.acme.SampleView`. By convention, the XML file should be named SampleView.xml, and it should be somewhere in the classpath under a directory structure equal to com/acme. XML-based views are typically placed in griffon-app/resources, but this isn't a strict requirement. The code needed to read and bootstrap a simple externalized view looks like the following snippet:

```
package com.acme;

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);
    }
}
```

Pay attention to the bolded section—that's all you need to instruct the view to read the default XML file. The `args` parameter contains all the elements the view's builder might require, such as the controller and model. The XML file could look like this:

```
<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>
```

It resembles a Groovy view, doesn't it? That's because this feature directly translates the XML text into a Groovy representation that the view's builder can understand. As a matter of fact, it generates an inline script that is equivalent to the following:

```
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)
}
```

Notice the usage of Groovy expressions in the XML. It's like having a ready-made expression language! Also, be sure to use escape literal values with single or double quotes, as is done here for the `id` properties of each text field; otherwise the builder will complain. You might be thinking, "If the XML looks so close to what you would write in Groovy, why bother with the XML in the first place?" We wonder that too; alas, some people still prefer XML over a scripting language. It's just the way the world works.

Full Java MVC members

As you just saw in the previous section, the ability to change the source type of a view is useful and isn't restricted to views; you can replace a controller, a model, or even a service in the same manner. You can also create a new group using Java as the source type from the start—just make sure you specify the `fileType` parameter, as demonstrated in the following example:

```
$ griffon create-mvc custom -fileType=java
```

This command will generate all MVC members using a Java-based template instead of the default Groovy one. You've seen the view already, so let's take a peek at the controller and model. First comes the controller:

```
package sample;

import java.awt.event.ActionEvent;
import org.codehaus.griffon.runtime.core.AbstractGriffonController;

public class CustomController extends AbstractGriffonController {
  private CustomModel model;
  public void setModel(CustomModel model) { this.model = model; }
  public void action(ActionEvent e) {}
}
```

This class must comply with the contract of a Griffon controller—that's why it extends from a specific class (`AbstractGriffonController`). The template shows how actions can be defined. Because you're working with Java now, using closures is out of the question, but actions can be defined as public methods. You can change the type of the parameter; `ActionEvent` is the most usual case, which is why it's suggested by the template.

Next is the model:

```
package sample;

import org.codehaus.griffon.runtime.core.AbstractGriffonModel;

public class CustomModel extends AbstractGriffonModel {
  private String input;

  public String getInput() { return input; }
  public void setInput(String input) {
    firePropertyChange("input", this.input, this.input = input);
  }
}
```

As with the previous two artifacts, the model requires a custom superclass (`AbstractGriffonModel`). The template shows how a simple observable property can be defined. It must be done in this way because `@Bindable` can't be used with Java code.

It's worth mentioning that there's a specific interface for each artifact type provided by Griffon. Following the naming conventions, their class names are as follows:

```
griffon.core.GriffonModel
griffon.core.GriffonView
griffon.core.GriffonController
griffon.core.GriffonService
```

You're required to implement the corresponding interface for a particular artifact. Given that the behavior is similar for all artifacts, Griffon provides base implementations for each interface; these are the abstract classes you saw in the previous code snippets. But nothing prevents you from implementing any of these interfaces from scratch.

If you're wondering whether a different source file type can be applied to the initial MVC group without resorting to calling `replace-artifact`, then the answer is yes. Just be sure to specify the `fileType` argument when issuing the call to `create-app`.

Preferring services over controllers

You know now that every MVC member can be written in Java too. Although that ability can be useful, sometimes it's better to keep the encapsulation of legacy code and not mix it with specific Griffon artifact behavior. This is where services come to your aid. You can use a service as a bridge between controllers and the target legacy code. In this way, you can take full advantage of Groovy in your controller, models, and views while still accessing all the behavior from the legacy classes from a safe vantage point.

Keep in mind that services are treated as singletons and are automatically instantiated by the framework. Services are also automatically registered as application event listeners, which can help at times to initialize any components belonging to the legacy code.

Using events to your advantage

Recall from chapters 2 and 8 that Griffon provides a powerful event mechanism that's paired with the application's life cycle. This enables you to register initialization code for any legacy component that can be performed at the right time. Be aware that addons fire a new set of events when they're being processed by the application while it's starting up; you'll have more options to micromanage the instantiation and customization of legacy code.

Finally, remember that any component can be transformed into an event publisher if it's annotated with `@EventPublisher` or, in the case of Java code, if it relies on `EventRouter` to send out notifications. Events make it simple to communicate disparate components, which is always a good thing to have in mind when you're dealing with legacy code.

index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

JAVA

GRIFFON IN ACTION

A. Almiray • D. Ferrin • J. Shingler

You can think of Griffon as Grails for the desktop. It is a Groovy-driven UI framework for the JVM that wraps and radically simplifies Swing. Its declarative style and approachable abstractions are instantly familiar to developers using Grails or JavaFX.

With **Griffon in Action** you get going quickly. Griffon's convention-over-configuration approach requires minimal code to get an app off the ground, so you can start seeing results immediately. You'll learn how SwingBuilder and other Griffon "builders" provide a coherent DSL-driven development experience. Along the way, you'll explore best practices for structure, architecture, and lifecycle of a Java desktop application.

What's Inside

- Griffon from the ground up
- Full compatibility with Griffon 1.0
- Using SwingBuilder and the other "builders"
- Practical, real-world examples
- Just enough Groovy

Written for Java developers—no experience with Groovy, Grails, or Swing is required.

Andres Almiray is the project lead of the Griffon framework, frequent conference speaker, and Java Champion. **Danno Ferrin** is cofounder of Griffon and an active Groovy committer. **James Shingler** is a technical architect, conference speaker, open source advocate, and author.

To download their free eBook in PDF, ePub and Kindle formats, owners of this book should visit manning.com/GriffoninAction

"A thorough source of information ... the definitive guide."

—From the Foreword by Dierk König, author of *Groovy in Action*

"If you think building desktop apps is complex, this awesome book will change your mind!"

—Guillaume Laforge
Groovy project lead

"Brings life back into Java desktop application development."

—Santosh Shanbhag
Monsanto Company

"Griffon makes Java GUI programming easy. *Griffon in Action* makes it fun."

—Michael Kimsal, publisher of GroovyMag

ISBN 13: 978-1-935182-23-8
ISBN 10: 1-935182-23-4
54499
9 781935 182238

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]