GRIFFON
IN ACTION

Andres Almiray
Danno Ferrin
James Shingler

FOREWORD BY **Dierk König**

MANNING

Griffon in Action

by Andres Almiray
Danno Ferrin
James Shingler

Chapter 1

brief contents

v

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.
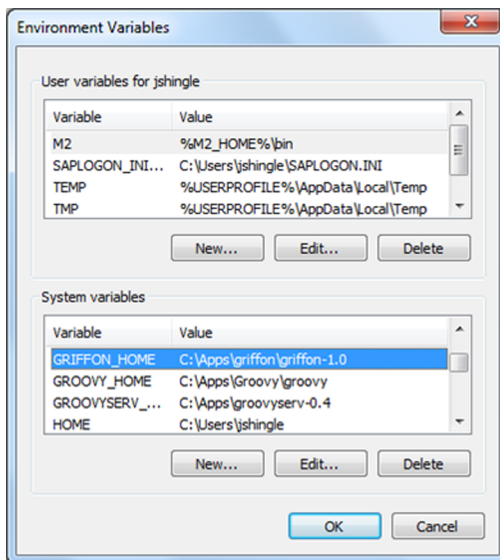

Figure 1.1   Updating variable settings on Windows

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.
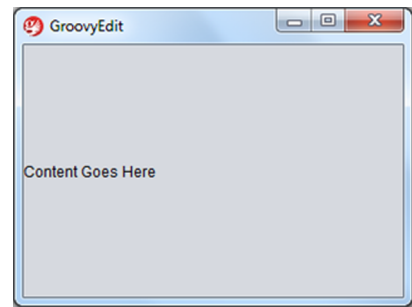
---

[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.

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.

Figure 1.3   Your first application running in Web Start mode

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.

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.

Figure 1.4   The GroovyEdit application running in applet mode

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
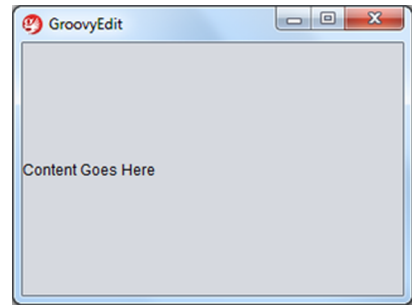
---

[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 multi-threaded 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:
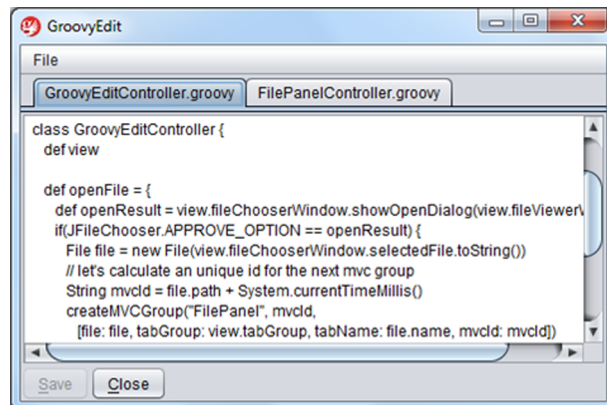

Figure 1.5   Finished GroovyEdit application displaying two tabs with its own source code

- 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 (`JFile-Chooser`), 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                                            ❶ Declare FileChooser
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') {                                        Declare menuBar and
            menuItem 'Open'                                   menu structure
            separator()
            menuItem 'Quit'
        }
    }
    borderLayout()                                            ❷ Declare reference
    tabbedPane(id: 'tabGroup', constraints: CENTER)              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'))
}
```
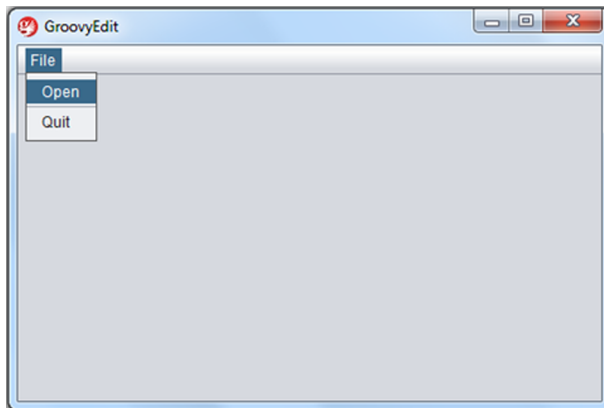

Figure 1.6   The GroovyEdit application now has a menu.

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                         ◁──────────  Set open action
         separator()
         menuItem quitAction                         ◁──────────  Set quit action
      }
   }
   borderLayout()
   tabbedPane id: 'tabGroup', constraints: CENTER
}
```
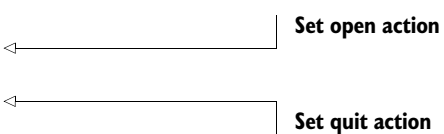
Run the application again, and you'll see the newly defined properties of the menu items as shown in figure 1.7. They now include mnemonics and accelerators, honoring the current operating system.

What about adding some behavior to the menu items? A controller is the appropriate location for behaviors.
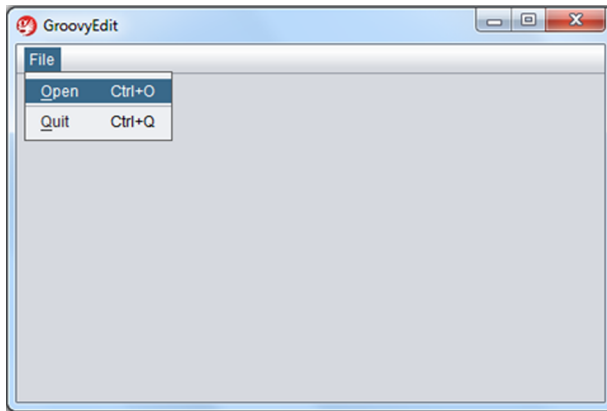
Figure 1.7   Native menu accelerators (Windows)

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 Groovy-EditModel. 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 {                                    To be defined
    def view                                                      shortly

    def openFile = {                          Define file action
        def openResult = view.fileChooserWindow.showOpenDialog()
        if( JFileChooser.APPROVE_OPTION == openResult ) {
        }
    }
                                                              ❶  Define quit
    def 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) {      ◁─┐  Define
    panel(title: tabName, id: 'tab') {                          ❶  tab
        borderLayout()
        scrollPane(constraints: CENTER) {
            textArea(id: 'editor')
        }
        hbox(constraints: SOUTH) {
            button 'Save'
```

```
          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 tabbed-pane 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 `create-MVCGroup()`. 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
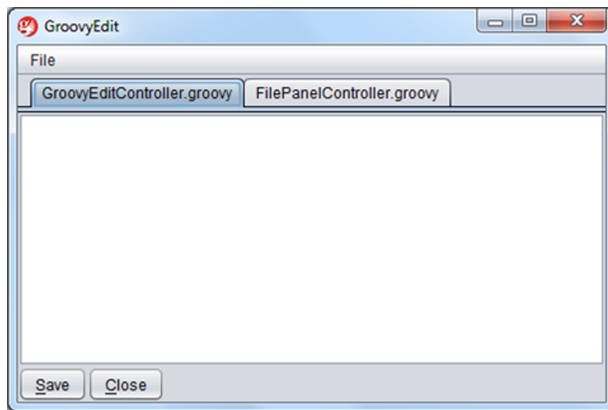
Figure 1.8   GroovyEdit displaying
two tabs. But where is the content?

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, mediat-
ing 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 `Property-ChangeEvent`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)
    }
}
```

❶ Inject model and view properties

❷ Run outside EDT

Read file

Run in ❸ 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 ❶, 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},                       Action
        name: 'Save',                                      set
        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) {   Content area
    panel(title: tabName, id: 'tab') {
        borderLayout()
        scrollPane(constraints: CENTER) {
            textArea(id: 'editor', text: bind {model.fileText})
        }
        hbox(constraints: SOUTH) {                         Button
            button saveAction                              area
            button closeAction
        }
    }
}
bean(model, dirty: bind {editor.text != model.fileText})   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`):

```java
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?

```groovy
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();          // Swing components must
        frame.setVisible(true);            // be created in EDT
      }
    });
  }

  private static JFrame buildUI() {
    JFrame frame = new JFrame("JavaFrame");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);   // Instantiate frame
    frame.getContentPane().setLayout(                       // and set properties
      new GridLayout(3,1)                                   // 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(){      Add event listener
      public void actionPerformed(ActionEvent event) {    on button
        output.setText(input.getText());
      }
    });
    frame.getContentPane().add(input);
    frame.getContentPane().add(button);
    frame.getContentPane().add(output);
    frame.pack();
    return frame;
  }
}
```

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 `Property-ChangeListeners` 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.
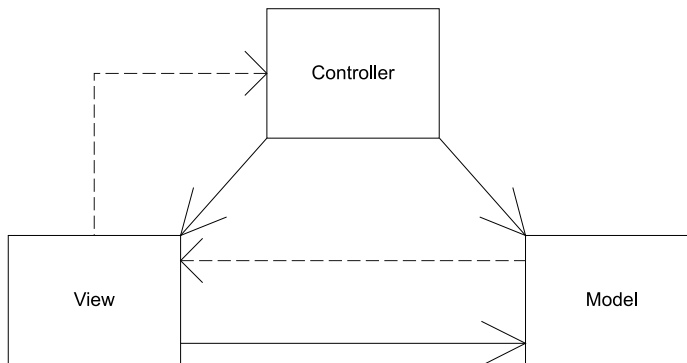

Figure 1.9   A diagram of the Model-View-Controller design pattern

---

[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
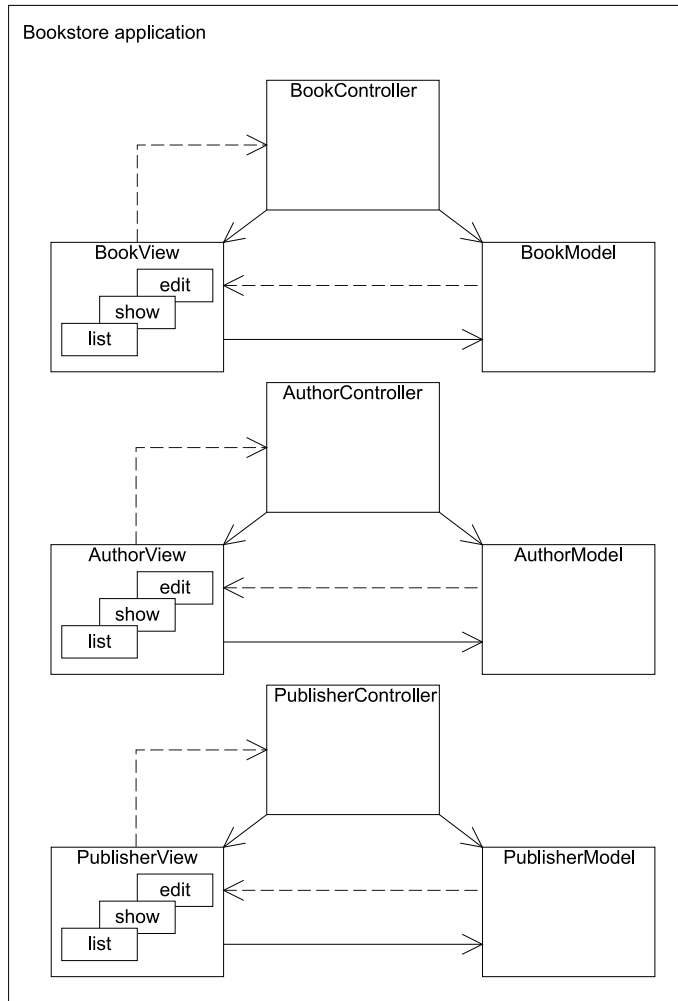
Figure 1.10   Bookstore application MVC model

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.

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

Free eBook
SEE INSERT

> **"**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

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]