In this Practically Groovy article, Scott Davis tackles a topic that strikes fear into most server-side Java™ developers: Swing. As you'll learn, Groovy's SwingBuilder takes some of the sting out of this powerful yet complex GUI framework.

View more content in this series

I recently interviewed Ted Neward, author of the IBM developerWorks article series The busy Java developer's guide to Scala (see Related topics). We talked about an interesting Twitter library he's been building in the series called Scitter (Scala + Twitter). Scitter highlights Scala's Web services and XML-parsing capabilities, but Ted admitted that he might never get around to putting a front end on the API. This, of course, got me thinking about what a Twitter GUI written in Groovy might look like. Doesn't Gwitter (Groovy + Twitter) have a nice ring to it?

About this series

Groovy is a modern programming language that runs on the Java platform. It offers seamless integration with existing Java code while introducing dramatic new features like closures and metaprogramming. Put simply, Groovy is what the Java language would look like had it been written in the 21st century.

The key to incorporating any new tool into your development toolkit is knowing when to use it and when to leave it in the box. Groovy can be extremely powerful, but only when applied properly to appropriate scenarios. To that end, the Practically Groovy series explores the practical uses of Groovy, helping you learn when and how to apply them successfully.

I won't tackle Scala + Groovy integration in this article, although there is a lot of potential synergy between the two languages. Instead, I'll take a look at a corner of the Java ecosystem that is often overlooked by Java developers: Swing. But before I do, I'll show you how Groovy's XmlSlurper makes short work of Twitter's Atom feed.

The Twitter Search API

Looking at the online documentation for the Twitter Search API (see Related topics), it appears that you can make a simple HTTP GET request to search Twitter. The query is passed in via a q
parameter in the query string, and the results can be returned in either Atom (an XML syndication format) or JavaScript Object Notation (JSON). So to get a set of Atom results for all mentions of *thirstyhead*, you need to make an HTTP GET request like this: http://search.twitter.com/search.atom?q=thirstyhead.

As shown in Listing 1, the results come back as a series of `<entry>` elements nested inside of a `<feed>` element:

**Listing 1. Twitter search Atom results**

```xml
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
  <entry>
    <id>tag:twitter.com,2007:http://twitter.com/thirstyhead/statuses/3419507135</id>
    <published>2009-08-20T02:54:54+00:00</published>
    <updated>2009-08-20T02:54:54+00:00</updated>
    <link type="text/html" rel="alternate" href="http://twitter.com/thirstyhead/statuses/3419507135" />
    <link type="image/jpeg" rel="image" href="http://s3.amazonaws.com/twitter_production/profile_images/73550313/flame_normal.jpg" />
    <author>
      <name>ThirstyHead.com</name>
      <uri>http://www.thirstyhead.com</uri>
    </author>
  </entry>
  <entry>...</entry>
  <entry>...</entry>
  <!-- snip -->
</feed>
```

In "Practically Groovy: Building, parsing, and slurping XML," you saw how easy it is to rip through XML results using Groovy's `XmlSlurper`. Now that you know what these specific results look like, create a file named `searchCl.groovy`, as shown in Listing 2:

**Listing 2. A Groovy script to parse the Atom results**

```groovy
if(args){
  def username = args[0]
  def addr = "http://search.twitter.com/search.atom?q=${username}"
  def feed = new XmlSlurper().parse(addr)
  feed.entry.each{
    println it.author.name
    println it.published
    println it.title
    println "-"*20
  }
} else{
  println "USAGE: groovy searchCl <query>"
}
```

Type `groovy searchCl thirstyhead` at the command line, and revel in your 13-line domination of the Atom results, as shown in Listing 3:
Listing 3. Running the searchCli.groovy script

$ groovy searchCli thirstyhead

thirstyhead (ThirstyHead.com)
2009-08-20T02:54:54Z
New series from Andrew Glover:
Java Development 2.0 http://bit.ly/bJX5i
--------------------
kung_foo (kung_foo)
2009-08-18T12:33:32Z
ThirstyHead interviews Venkat Subramaniam:
http://blip.tv/file/2484840 "Groovy and Scala are good friends..."
(via @mittie). very good.

//snip

Create the initial Gtwitter classes

While Groovy scripts are great for informal utilities and proof-of-concept experiments, writing
Groovy classes isn't that much more difficult. And as a reward, you can compile the Groovy
classes and call them from Java code.

For example, create Tweet.groovy as shown in Listing 4:

Listing 4. Tweet.groovy

class Tweet{
    String content
    String published
    String author

    String toString(){
        return "${author}: ${content}"
    }
}

As you already know, this Plain Old Groovy Object (POGO) is a drop-in replacement for the
significantly more verbose Plain Old Java Object (POJO).

Now convert the search script in Listing 2 to Search.groovy, as shown in Listing 5:

Listing 5. Search.groovy

class Search{
    static final String addr = "http://search.twitter.com/search.atom?q="

    static Object[] byKeyword(String query){
        def results = []
        def feed = new XmlSlurper().parse(addr + query)
        feed.entry.each{entry->
            def tweet = new Tweet()
            tweet.author = entry.author.name
            tweet.published = entry.published
            tweet.content = entry.title
            results << tweet
        }
        return results as Object[]
    }
}

Practically Groovy: SwingBuilder and the Twitter API, Part 1
Normally, I'd just leave results as a java.util.ArrayList. But the javax.swing.JList you will use later in the article needs an Object[], so consider this a wee bit of foreshadowing.

Notice that I took away the main() method in Search.groovy. How can you interact with this class now? Why, from a unit test, of course! Create SearchTest.groovy, as shown in Listing 6:

**Listing 6. SearchTest.groovy**

```groovy
class SearchTest extends GroovyTestCase{
    void testSearchByKeyword(){
        def results = Search.byKeyword("thirstyhead")
        results.each{
            assertTrue it.content.toLowerCase().contains("thirstyhead") ||
            it.author.toLowerCase().contains("thirstyhead")
        }
    }
}
```

If you type `groovy SearchTest` at the command prompt and see `OK (1 test)` (as shown in Listing 7), you have successfully converted your simple search script into a set of reusable classes:

**Listing 7. Results of a successful test run**

```
$ groovy SearchTest
  Time: 4.64
OK (1 test)
```

Now that the underlying infrastructure is in place, the next step is to start putting a pretty face on it.

**Introducing SwingBuilder**

Swing is an incredibly powerful GUI toolkit. Unfortunately, sometimes the power is overwhelmed by the complexity. If you are new to Swing, it can feel like learning to fly a Boeing 747 when what you really need is a single-engine Cessna — or a hang glider.

Groovy's SwingBuilder doesn't reduce any of the intrinsic complexity of tasks such as choosing the right LayoutManager or properly handling threading issues. What it reduces is the syntactic complexity. As you'll see in just a moment, Groovy's named argument / vararg constructors are perfectly suited for the various JComponents that you need to instantiate and then immediately configure with a series of setters. (For more details on SwingBuilder, see Related topics.)

But equally valuable is Groovy's use of closures. My longstanding problem with Swing is that a natural hierarchy seems to get lost amidst the implementation details. In Java code, you end up with a disjointed set of components and no real sense of what belongs to what. You can declare a JFrame, a JPanel, and a JLabel in any order. In code, this makes them look like peers, when in fact the JFrame contains the JPanel, which in turn contains the JLabel. See Listing 8 for an example:
Listing 8. HelloJavaSwing.java

```java
import javax.swing.*;

public class HelloJavaSwing {
    public static void main(String[] args) {
        JPanel panel = new JPanel();
        JLabel label = new JLabel("Hello Java Swing");
        JFrame frame = new JFrame("Hello Java Swing");
        panel.add(label);
        frame.add(panel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 300);
        frame.setVisible(true);
    }
}
```

Compiling this code (`javac HelloJavaSwing.java`) and running it (`java HelloJava`) should yield an application as shown in Figure 1:

**Figure 1. HelloJavaSwing**

Listing 9 shows the same application written in Groovy. As you can see, the SwingBuilder’s judicious use of closures allows you to see the chain of ownership clearly and declaratively in code.

Listing 9. HelloGroovySwing.groovy

```groovy
import groovy.swing.SwingBuilder
import javax.swing.*

def swingBuilder = new SwingBuilder()
swingBuilder.frame(title:"Hello Groovy Swing",
    defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
    size:[200,300],
    show:true) {
    panel(){
        label("Hello Groovy Swing")
    }
}
```

Type `groovy HelloGroovySwing` to see the application shown in Figure 2:

**Figure 2. HelloGroovySwing**

Notice in Listing 9 that the leading `J` is dropped from all component names — much as the superfluous `get` and `set` are dropped from method names. Next, notice the named argument constructor for `frame`. Behind the scenes, Groovy is calling the no-argument default constructor
and then calling the setter methods — no different from the previous Java example. But the fact that they are all gathered together in the constructor keeps things neat and tidy, and dropping the set prefix and the parentheses at the end reduces visual noise considerably.

If you don’t already know Swing, this probably still looks complicated. But if you have even the slightest bit of Swing experience, it most likely looks like Swing boiled down to its essence: clean, clear, and efficient.

Just as you did in the preceding section, take what you’ve learned from the script and convert it to a class. Create a file named Gwitter.groovy, as shown in Listing 10. This is the humble start of your Groovy + Twitter client UI.

Listing 10. The skeleton of the Gwitter UI

```groovy
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

class Gwitter{
  static void main(String[] args){
    def gwitter = new Gwitter()
    gwitter.show()
  }

  void show(){
    def swingBuilder = new SwingBuilder()
    swingBuilder.frame(title:"Gwitter",
      defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
      size:[400,500],
      show:true) {
    }
  }
}
```

Type `groovy Gwitter` to verify that the empty frame appears. If everything works as expected, the next step is to add a simple menu to the application.

Adding a menubar

Creating menus in Swing offers another example of components that have a natural hierarchy. You create a `JMenuBar` that contains one or more `JMenu`s, which in turn contain(s) one or more `JMenuItem`s.

To create a `File` menu with an `Exit` menu item, add the code in Listing 11 to Gwitter.groovy:

Listing 11. Adding a `File` menu to Gwitter

```groovy
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

class Gwitter{
  static void main(String[] args){
    def gwitter = new Gtwitter()
    gtwitter.show()
  }

  void show(){
    def swingBuilder = new SwingBuilder()
    swingBuilder.frame(title:"Gwitter",
      defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
      size:[400,500],
      show:true) {
      }
  }
}
```
void show()
{
    def swingBuilder = new SwingBuilder()

    def customMenuBar = {
        swingBuilder.menuBar{
            menu(text: "File", mnemonic: 'F') {
                menuItem(text: "Exit", mnemonic: 'X', actionPerformed: { dispose() })
            }
        }
    }

    swingBuilder.frame(title:"Gwitter",
                        defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                        size:[400,500],
                        show:true) {
        customMenuBar()
    }
}

Notice the nested hierarchy of the customMenuBar closure. I pulled it out separately here for readability, but I could have just as easily defined it inline in the middle of the frame. Once the closure is defined, I call it inside of the frame closure. Type groovy Gwitter once again to verify that the File menu appears, as shown in Figure 4. Choose File > Exit to shut down the application.

**Figure 4. Gwitter’s File menu**

Look again at Listing 11. Notice that the actionPerformed handler is defined as a closure instead of an anonymous class. Doesn’t it make the code clean and easy to read compared to the Java alternative?

Now it's time to add some form elements to enable searching.

**Adding the search panel**

Seasoned Swing developers are adept at assembling finished applications out of discrete JPanels. These container components are an easy way to group similar, related components together.

For example, Gwitter needs a JTextField to allow users to type in search criteria, and a JButton for submitting the request. It makes sense to group these two objects together in a searchPanel closure, as shown in Listing 12:

**Listing 12. Adding the search panel**

```java
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

class Gwitter{

    addPanel{
        def searchPanel = {
            JPanel{
                add(TextField(placeholder:"Search..."))
                add(Button(text:"Search"))
            }
        }
    }
}
```
def searchField

static void main(String[] args){
    def gwitter = new Gwitter()
    gwitter.show()
}

void show(){
    def swingBuilder = new SwingBuilder()
    def customMenuBar = {
        swingBuilder.menuBar{
            menu(text: "File", mnemonic: 'F') {
                menuItem(text: "Exit", mnemonic: 'X', actionPerformed: {dispose() })
            }
        }
    }
    def searchPanel = {
        swingBuilder.panel(constraints: BorderLayout.NORTH){
            searchField = textField(columns:15)
            button(text:"Search", actionPerformed:{ /* TODO */ } )
        }
    }
    swingBuilder.frame(title:"Gwitter",
                       defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                       size:[400,500],
                       show:true) {
        customMenuBar()
        searchPanel()
    }
}

Once you start dealing with panels, the issue of choosing an appropriate LayoutManager arises. By default, a JPanel uses a FlowLayout. This means that the textField and the button will be placed horizontally, one after the next.

The contentPane of a JFrame is slightly different — it uses BorderLayout by default. This means that when adding the searchPanel to the frame, you need to specify which area you'd like it to appear in: NORTH, SOUTH, EAST, WEST, or CENTER. (In a nod to those who failed Geography 101, you can also use PAGE_START, PAGE_END, LINE_START, LINE_END, and CENTER.) For more on the various LayoutManagers available in Swing, see Related topics.

Notice that the searchField variable is declared at the class level. This is so other components like the button have access to it. The rest of the components are anonymous. A quick glance at the list of class attributes hints at the relative importance of certain components over the others.

You probably already noticed that the button's actionPerformed listener doesn't do anything ... yet. There's really nothing for it to do right now. Before you can implement it, you need to add another panel to the application: one to hold the search results.

Adding the results panel

As shown in Listing 13, you define a resultsPanel in a nested closure just as you did with the searchPanel. Only this time, you nest another container inside of the panel: a JScrollPane. This
component allows horizontal and vertical scrollbars to appear and disappear as needed. The results of the `Search.byKeyword()` method call are displayed in a `JList` named `resultsList`. (The `JList.setListData()` methods takes an `Object[]`— remember that is exactly what you set the `Search.byKeyword()` method to return.)

**Listing 13. Adding the `resultsPanel`**

```groovy
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

class Gwitter{
    def searchField
    def resultsList

    static void main(String[] args){
        def gwitter = new Gwitter()
        gwitter.show()
    }

    void show(){
        def swingBuilder = new SwingBuilder()

        def customMenuBar = {
            swingBuilder.menuBar{
                menu(text: "File", mnemonic: 'F') {
                    menuItem(text: "Exit", mnemonic: 'X', actionPerformed: {dispose() })
                }
            }
        }

        def searchPanel = {
            swingBuilder.panel(constraints: BorderLayout.NORTH){
                searchField = textField(columns:15)
                button(text:"Search", actionPerformed:{
                    resultsList.listData = Search.byKeyword(searchField.text) }
                )
            }
        }

        def resultsPanel = {
            swingBuilder.scrollPane(constraints: BorderLayout.CENTER){
                resultsList = list()
            }
        }

        swingBuilder.frame(title:"Gwitter",
            defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
            size:[400,500],
            show:true ) {
            customMenuBar()
            searchPanel()
            resultsPanel()
        }
    }
}
```

Notice that `resultsList` variable, like `searchField`, is defined at the class level. Both variables are used in the button's `actionPerformed` handler in the `searchPanel`.

With the addition of the `resultsPanel`, Gwitter is now functional. Type `groovy Gwitter` at the command prompt and verify that it works as expected. Searching on `thirstyhead` should yield results similar to those in Figure 5:
You could quit now and claim success, but I'd like to address two more issues before taking the victory lap. The first issue is the potential threading nightmare innocently introduced by the search button's `actionPerformed` handler. The other is the general homeliness of the application. Both are solved in the next two sections.

**The event dispatch thread**

The cruel irony of Swing is that it either expects graphic designers to be comfortable with multithreading issues better suited for software engineers, or software engineers to understand the subtle nuances of graphic design and general usability issues.

I couldn't possibly cover a topic as complex as threading issues in a Swing application in a few short paragraphs. Suffice it to say that Swing applications are essentially single-threaded out of the box. Everything happens on the event dispatch thread (EDT). When users grumble about how
sluggish and unresponsive their Swing application is, it is usually because a novice developer made a lengthy, computationally intensive database query or Web services call on the EDT — the same thread that handles refreshing the screen, menu clicks, and so on. I had you unwittingly do exactly that sort of thing in the search button's `actionPerformed` handler. (You can see how easy it is to make this common mistake.)

Thankfully, the `javax.swing.SwingUtilities` class offers a few aptly named convenience methods — `invokeAndWait()` and `invokeLater()` — that ease some of the threading concerns. The two methods allow you to perform actions either synchronously or asynchronously on the EDT. (See Related topics for more on the `SwingUtilities` class.) SwingBuilder makes it easy to call either of these two methods, and it gives you a third option: the ability to spin up a new thread effortlessly to perform potentially expensive actions.

To perform a synchronous call on the EDT (`SwingUtilities.invokeAndWait()`), you can wrap the call in an `edt{}` closure. To perform an asynchronous call on the EDT (`SwingUtilities.invokeLater()`), wrap the call in a `doLater{}` closure. But I'm going to have you exercise the third option: spinning up a new thread to handle the `Search.byKeyword()` method call. To do so, wrap the code in a `doOutside{}` closure, as shown in Listing 14:

```
Listing 14. Using the doOutside closure

```def searchPanel = {
  swingBuilder.panel(constraints: BorderLayout.NORTH){
    searchField = textField(columns:15)
    button(text:"Search", actionPerformed:{
      doOutside{
        resultsList.listData = Search.byKeyword(searchField.text)
      }
    })
  }
}
```
Listing 15. Returning HTML in the toString() method

class Tweet{
    String content
    String published
    String author

    String toString(){
        //return "${author}: ${content}"

        return """<html>
        <body>
        <p><b><i>${author}:</i></b></p>
        <p>${content}</p>
        </body>
        </html>"
    }
}

The next tweak is a little more subtle. A common GUI trick is to stripe long lists or tables. By alternating colors on even and odd rows, it makes the list easier to scan. I searched for JList stripes in a search engine and followed the advice in the first article I found. The author recommended creating a custom DefaultListCellRenderer. I nodded sagely as I read each paragraph, and then shamelessly stole his example code verbatim (see Related topics for the full article).

Because Groovy syntax is a superset of Java syntax, I was able to copy and paste the Java code into a Groovy file unchanged. If I had a full build system in place that compiled both Java and Groovy code, I could have just as easily left it as a Java file. But by giving it a .groovy extension, I can run all of the Gwitter code uncompiled. Once again, I am taking advantage of the seamless integration between the Java language and Groovy. You can use any Java solution in a Groovy application unchanged.

Create a file named StripeRenderer.groovy and add the code shown in Listing 16:

Listing 16. Creating a zebra-stripe CellRenderer

import java.awt.*;
import javax.swing.*;

class StripeRenderer extends DefaultListCellRenderer {
    public Component getListCellRendererComponent(JList list, Object value,
        int index, boolean isSelected, boolean cellHasFocus) {
        JLabel label = (JLabel) super.getListCellRendererComponent(list, value,
            index, isSelected, cellHasFocus);

        if(index%2 == 0) {
            label.setBackground(new Color(230,230,255));
        }

        label.setVerticalAlignment(SwingConstants.TOP);
        return label;
    }
}

With the StripeRenderer class in place, the last thing you need to do is have the JList take advantage of it. Adjust the resultsPanel as shown in Listing 17:
Listing 17. Adding the custom CellRenderer to the JList

```groovy
def resultsPanel = {
    swingBuilder.scrollPane(constraints: BorderLayout.CENTER){
        //resultsList = list()
        resultsList =
            list(fixedCellWidth: 380, fixedCellHeight: 75, cellRenderer:new StripeRenderer())
    }
}
```

Type `groovy Gwitter` one last time at the command prompt. Searching on `thirstyhead` should give you results similar to those shown in Figure 6:

**Figure 6. Striped results**

thirstyhead (ThirstyHead.com):
New series from Andrew Glover: Java Development 2.0
http://bit.ly/bJX5i

kung_foo (kung_foo):
ThirstyHead interviews Venkat Subramaniam:
http://blip.tv/file/2484840 "Groovy and Scala are good friends..." (via @mittle). very good.

jmerritt (Jonathan Merritt):
@venkat_s: very nice ThirstyHead interview; so nice to hear such a coherent set of arguments. I hope you appear on the JavaPosse soon!

jmerritt (Jonathan Merritt):
RT @mittle: listening to: ThirstyHead interviews Venkat Subramaniam: http://blip.tv/file/2484840 "Groovy and Scala are good friends..."

mittle (Dierk König):
listening to RT @scottdavis99: ThirstyHead interviews Venkat Subramaniam: http://blip.tv/file/2484840 "Groovy and Scala are good friends..."

grailspodcast (grailspodcast):
RT @scottdavis99: ThirstyHead interviews Venkat Subramaniam: http://blip.tv/file/2484840 "Groovy and Scala are good...

I could spend considerably more time polishing Gwitter's look and feel, but I hope you're impressed with what about 50 lines of Swing code (not including the supporting classes, of course) accomplished.
Conclusion

As you've learned in this article, Groovy doesn't reduce Swing's intrinsic complexity, but it dramatically reduces the syntactic complexity. This frees you up to fight other, more important battles.

If this article piqued your interest in Groovy and Swing, you owe it to yourself to check out the Griffon project (see Related topics). It offers all of the scaffolding and the convention over configuration of a Grails project, but it is built on the shoulders of SwingBuilder and Groovy rather than Spring MVC and Hibernate. The project is still in its early stages — 0.2 is the most recent release as of this writing — but it was robust enough to win the Scripting Bowl for Groovy at JavaOne 2009. And one of the sample projects it ships with is Greet: a full Twitter client, implemented in Groovy.

Next time, you'll start adding more features to Gwitter. You'll learn how to handle basic HTTP authentication, take advantage of a cousin of XmlSlurper called ConfigSlurper, have Gwitter request and parse your friends timeline, and add a tabbed view. In Part 3, you'll top Gwitter off by adding HTTP POST capability so you can send your own tweets. Until then, I hope that you find plenty of practical uses for Groovy.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code for article examples</td>
<td>j-groovy09299.zip</td>
<td>24KB</td>
</tr>
</tbody>
</table>
Related topics

- Interview with Ted Neward: Scott Davis discusses the Scitter library and other topics with the author of the The busy developer's guide to Scala series.
- Twitter Search API: Locate the syntax your applications need for searching Twitter.
- "Java Web Start Persistence and JList Striping" (Joshua Marinacci, java.sun.com, November 2006): This article provided the zebra-striping solution implemented in Gwitter.
- Groovy Recipes (Scott Davis, Pragmatic Programmers, 2008): Learn more about Groovy and Grails in Scott Davis' latest book.
- Mastering Grails: Scott Davis's companion series focuses on this Groovy-based platform for Web development.