Grails
A Quick-Start Guide

The Pragmatic Programmers

Grails

A Quick-Start Guide

Dave Klein

Edited by Colleen Toporek

ISBN-10: 1-934356-46-8

ISBN-13: 978-1-934356-46-3

Printed on acid-free paper.

B6.0 printing, July 28, 2009

Version: 2009-8-7

Chapter 1

Introduction

1.1 Let Me Tell You About Grails...

Web development is a very rewarding experience. Building an application that can run from anywhere in the world is pretty awesome. Even in a corporate environment, you can deliver new features to your users, no matter where they are located, without ever touching their computer. It's a beautiful thing. Consider also what you can build: the potential for creativity on the web is unlimited.

The Java platform brings even more power to the party. The Java Servlet API and the plethora of libraries and frameworks in the Java ecosystem make it possible to include almost any feature you could want in a web application. It is an exciting time to be a web developer. However, it's not all sweetness and light.

With all this power comes a level of complexity that can be daunting. With most Java-based web frameworks, there are multiple XML configuration files to deal with, along with classes to extend and interfaces to implement. As a project grows, this complexity seems to increase exponentially.

Many web application frameworks have been created to address this problem. So many Java web frameworks have been developed that you might ask, "Why Grails? Why another framework?" That was my thought when I first heard about Grails.

I was at a conference which featured sessions on an array of Java-related technologies, planning to attend several talks on Java Server Faces, which is what I was working with at the time. During one of the time slots where there was nothing JSF-related, I wandered into a session on Grails by Scott Davis. And I have to say, I was impressed. But not convinced.

In the past, I had worked with so-called *rapid application development* tools on the desktop, and had seen the trade-off that you had to make to get these "applications in minutes." As soon as you needed to do more than the tool was designed for, you were stuck. I didn't want to go down that road again. Still, Grails did look like it would be a good choice for *small* applications. So I gave it a try.

After using Grails to build a website for our local Java user group, I was hooked. By day, I was struggling with JSF and EJB (Enterprise Java Beans); by night, I was having a blast building a website with Grails. I began to look for ways to take advantage of the brilliant simplicity of Grails in my day job. After all, I worked in a Java shop, and Grails is a fully compliant JEE[1] framework. It would produce a standard .war file, which could be deployed on our commercial JEE application server. Finally, an opportunity presented itself.

It was a small but important public-facing web application, planned as a six-week JSF/EJB project. With Grails, it was done in three weeks—and it turned out to be a little less trivial than we thought, because we needed to integrate with an existing EJB server. We found that the Grails "magic" was great for most of the application, and provided significant productivity boosts. We also found that when we needed to do something Grails didn't handle "out of the box,"[2] it was easy to dip into the underlying technologies and do what we needed. There were no black boxes or brick walls. It wasn't "the Grails way or the highway."

We went on to use Grails to rescue another, much larger project that was in trouble, with similar results. Grails is definitely not *just* for small applications!

1.2 How Does Grails Do It?

Grails takes a set of successful frameworks, each of which have made their own strides toward addressing the complexity of building web applications, and makes them all simpler, easier to use, and ultimately more powerful.

1. Java Enterprise Edition.
2. I use this term with some hesitation—http://dave-klein.blogspot.com/2008/08/out-of-box.html.

Grails bundles Spring, Hibernate, Sitemesh, HSQLDB, Jetty, and a host of other battle-hardened frameworks, and following the principle of *convention over configuration*,[3] it removes the complexity for most use cases. And it uses the dynamic Groovy programming language to magically give us easy access to the combined power of these tools.

Recall from my story that on the projects that I was involved in, Grails was a replacement for both JSF and EJB. JSF, like Struts before it, and JSP before that, is intended to address the web tier (the front end). EJB was the framework we were using to provide persistence, transactions, and various other services (the back end). Grails addresses the whole application, and more importantly, it allows *us* to address the whole application. Using the frameworks mentioned earlier, Grails gives us a complete, seamless MVC[4] framework that is really more of a web application platform than just another framework.

1.3 Why This Book?

The idea for this book came about while working on the projects I mentioned earlier. I had been working with Grails for a while, but there were four other developers working with me, and we really could have used a book to help bring them up to speed quickly. They didn't need a reference book yet, but something more than a collection of articles and blog posts (as helpful as those are).

As Grails' exposure and acceptance continues to grow, and as more and more developers have their "Wow!" moments, it will become even more important to have a resource to help them get started quickly. That's the goal of this quick-start guide. It is not intended to be a reference, or the only Grails book on your shelf. In this book I'll help you get started and become productive with Grails, but you will no doubt want to go beyond that. To help you dig deeper, I've included lists of books, websites, blogs and other helpful resources from the Groovy/Grails community in Appendix B, on page 205.

This book is, however, intended to be more than a cursory introduction. We will cover all of the basics of Grails and a few advanced topics as well. When we have finished our time together here, you will understand Grails well enough to use it in real projects. In fact, you will have

3. http://en.wikipedia.org/wiki/Convention_over_Configuration
4. Model View Controller. See http://en.wikipedia.org/wiki/Model-view-controller.

already used it in a real project, as that is what we are going to do together. More on that later.

1.4 Who Should Read This Book

This book is aimed at web developers looking for relief from the pain brought on by the complexity of modern web development. If you dream in XML and enjoy juggling multiple layers of abstraction at a time, or if you are in a job where your pay is based on the number of lines of code you write, then Grails may not be for you. If, on the other hand, you are looking for a way to be more productive, a way to be able to focus on the heart of your applications instead of all the technological bureaucracy, then you're in the right place.

I am assuming an understanding of web application development, but you don't need to be an expert to benefit from Grails and from this book. An understanding of Java or another object-oriented programming language would be helpful. If you have experience with Spring and Hibernate you are ahead of the curve, but if you've never even heard of them you'll do fine. You can go quite far with Grails and be using Spring and Hibernate extensively without even realizing it. Finally, the language of Grails is Groovy. I won't assume that you have any experience with Groovy, and you won't need a great deal of it to get going with Grails. However, some knowledge of Groovy syntax and constructs will be helpful, so we'll now embark on a brief tutorial.

1.5 Enough Groovy to Be Dangerous

Groovy is a dynamic language for the Java Virtual Machine (JVM). Of all the JVM languages, Groovy has the best integration with Java and probably the lowest barrier to entry for Java developers. Java is considered by many to be in the "C family" of languages; that is to say that its syntax borrows heavily from the C language. Other languages in this family are C++, C#, and, by its close relationship to Java, Groovy. Without getting into a debate on whether that syntax family is a good one, it is one that millions of developers are familiar with. That means that there are millions of developers that can quickly pick up Groovy!

Groovy, like Spring, Hibernate, and the other frameworks used in Grails, is included in the Grails install. You do not need to install Groovy to use Grails. However, Groovy is a great multi-purpose language, and

I encourage you to download it and take it for a spin. You will quickly become more productive in areas like XML processing, database access, file manipulation, and more. You can download the Groovy installation and find more information on the Groovy website.[5] There are also some excellent books available on Groovy such as Venkat Subramaniam's *Programming Groovy* [Sub08], Scott Davis' *Groovy Recipes: Greasing the Wheels of Java* [Dav08], and *Groovy in Action* [Koe07] by Dierk König and friends.

We're going to discuss the Groovy features that are most often used in a Grails application. But first, for the benefit of Java developers, we'll look at some of the differences between Java and Groovy.

1.6 Groovy Syntax Compared to Java

Despite the overall syntactic similarities, there are some differences between Groovy and Java that are worth noting. The first thing you'll notice in a block of Groovy code is the lack of semicolons; in Groovy, semicolons are optional. Return statements are also optional. If there is no return statement in a method, then the last statement evaluated is returned. Sometimes this makes sense, especially in the case of small methods that simply return a value or perform a single calculation. Other times it can be confusing. That's the beauty of the word "optional": when **return** makes code more readable, use it; when it doesn't, don't.

Parentheses for method calls are optional in most cases, the exception being when calling a method without any arguments. Here are some examples:

```
x = someMethodWithArgs arg1, arg2, arg3
y = someMethodWithoutArgs()
```

Methods without arguments need the parentheses so that Groovy can tell them apart from *properties*. Groovy provides "real" properties. All fields in a Groovy class are given getters and setters at compile time. When you access a field of a Groovy class, it may look like you are directly accessing the field, but behind the scenes, the getter or setter is being called. If you're not convinced, you can call them explicitly. They'll be there even though you didn't code them.

5. http://groovy.codehaus.org

Download introduction/get_property.groovy

```
class Person {
    String name
}
def person = new Person()
person.name = 'Abigail'
assert person.getName() == 'Abigail'
person.setName('Abi')
assert person.name == 'Abi'
```

If you *explicitly declare* a get or set method for a property, it will be used as expected.

Download introduction/explicit_set_property.groovy

```
class Person {
    String name

    void setName(String val){
        name = val.toUpperCase()
    }
}

def person = new Person(name:'Sarah')
assert person.name == 'SARAH'
```

This last snippet shows a few other differences in Groovy. First, all Groovy classes automatically get a named-args constructor. This is a constructor that takes a Map and calls the set method for each key that corresponds to a property.[6] You can easily see how this might save several lines of code with larger classes. Grails takes advantage of this feature to assign the values from a web page to a new object instance. Second, in Groovy, types are optional. Instead of giving a variable an explicit type, we can use the **def** keyword to designate that this variable will be *dynamically* typed. The third difference is the use of == in the assert statements. In Groovy, == is the same as calling the equals() method on the left hand operand.

Now, the toUpperCase() method we just used is the same as in Java. But for a little fun, we can modify that last example to try out one of the many methods that Groovy adds to the String class.[7]

6. Any elements in the map that do not correspond to a property are ignored by the named-args constructor.
7. You can find more goodies in the API docs at http://groovy.codehaus.org/groovy-jdk/java/lang/String.html.

Download introduction/reverse.groovy

```
class Person {
    String name

    void setName(String val){
        name = val.toUpperCase().reverse()
    }
}

Person p = new Person(name:'Hannah')

assert p.name == 'HANNAH'
```

It worked. Trust me.

Not only does Groovy enhance the java.lang.String class, but it also adds an entirely new one.

1.7 Groovy Strings

Groovy adds a new string known as a GString. A GString can be created by declaring a literal with double quotes; a string literal with single quotes is a java.lang.String. A GString can be used in place of a Java String. If a method is expecting a String and is given a GString, it will be cast at runtime.

The beauty and power of the GString is its ability to evaluate embedded Groovy expressions. Groovy expressions can be designated in two ways. For simple values that are not directly adjacent to any plain text, you can just use a dollar sign, like this:

```
"Hello $name"
```

For more involved expressions, you can use the dollar sign and a pair of curly braces:

```
"The 5th letter in 'Encyclopedia' is ${'Encyclopedia'[4]}"
```

There can be any number of expressions in a given GString, and single quotes can be embedded without any escaping. This comes in handy when generating HTML, as we'll see later. For now, let's take a look at the GString in action.

Download introduction/hello_groovy_string.groovy

```
def name = 'Zachary'
def x = 3
def y = 7
def groovyString = "Hello ${name}, did you know that $x x $y equals ${x*y}?"
```

```
assert groovyString == 'Hello Zachary, did you know that 3 x 7 equals 21?'
```

1.8 Groovy Closures

A Groovy Closure, in simple terms, is an executable block of code that can be assigned to a variable, passed to a method, and executed.[8] Many of the enhancements Groovy has made to the standard Java libraries involved adding methods that take a Closure as a parameter.

A Closure is declared by placing code between curly braces. It can be declared as it is being passed to a method call, or it can be assigned to a variable and used later. A Closure can take parameters by listing them after the opening curly brace and separating them from the code with a *dash-rocket* (->), like so:

```
def c = {a, b ->  a + b}
```

If no parameters are declared in a Closure, then one is implicitly provided: it's called it. Take a look at the following example.

Download introduction/closure_times.groovy

```
def name = 'Dave'
def c = {println "$name called this closure ${it+1} time${it > 0 ? 's' : ''}"}
assert c instanceof Closure
5.times(c)
```

There's a fair bit of new stuff in these three lines of code. Let's start at the top. The variable name is available when the Closure is executed. Anything that is in scope when the Closure is created will be available when it is executed, even if it is being executed by code in a different class. The Closure is being assigned to the variable c, and has no declared parameters. It does have and use the implicit parameter it. The code in this Closure takes advantage of another Groovy shortcut. What would be in Java System.out.println() is now just println(). When you look at the text of the GString that follows, it becomes obvious that this code will only work if whatever calls this Closure passes it a single parameter that is a number. That's just what the times() method, which Groovy adds to Integer, does. The parentheses are not required for the times() method, but I added them to emphasize that the Closure

8. There has been much discussion and some confusion over the definition of a "closure" in programming languages. Some argue that what Groovy defines as a closure isn't. If you're ever in town we can discuss it over a cup of coffee, but for our purposes, we'll be referring to closures as defined at http://groovy.codehaus.org/Closures.

was being passed in as a parameter. The output from this code looks like this:

```
Dave called this closure 1 time
Dave called this closure 2 times
Dave called this closure 3 times
Dave called this closure 4 times
Dave called this closure 5 times
```

There is much more to Closures than we can cover here, and I highly recommend the coverage of this topic in *Programming Groovy* [Sub08]. We will see more examples of Closures in action as we look at Groovy collection classes.

1.9 Groovy Collections

Groovy offers many enhancements to the standard Java collection classes. We'll take a look at the three collection types that are most used in Grails. The List, Map and Set are powerful tools, and Groovy gives them a new edge. I know—technically Map is not a Collection; that is, it does not implement the Collection interface. But for our purposes, it is a *collection* in that it holds objects. So leaving semantic sensitivities aside, let's look at what Groovy has done for these classes.

List

One of the first interesting things to learn about the List in Groovy is that it can be created with a literal declaration.

Download introduction/groovy_list.groovy

```
def colors = ['Red', 'Green', 'Blue', 'Yellow']
def empty = []

assert colors instanceof List
assert empty instanceof List
assert empty.class.name == 'java.util.ArrayList'
```

A comma separated list inside of square brackets is an intialized List. It can contain literal numbers, strings, or any other objects. This is a good time to point out that in Groovy, *everything* is an object. Even simple data types such as int or boolean are auto-boxed objects. (That's why we were able to call the times() method on the literal 5 in our earlier example.) The last line of this example shows that the default List implementation in Groovy is a java.util.ArrayList.

Groovy has also added a host of helpful methods to the List interface. One of the most useful is each(). This method is actually added to all objects in Groovy, but it is most useful with collection types. The each() method on List takes a Closure as a parameter and calls that Closure for each element in the List, passing in that element as the single "it" parameter.

Download introduction/groovy_list.groovy

```
def names = ['Nate', 'Matthew', 'Craig', 'Amanda']

names.each{
  println "The name $it contains ${it.size()} characters."
}
```

This example will print the following output to the console:

```
The name Nate contains 4 characters.
The name Matthew contains 7 characters.
The name Craig contains 5 characters.
The name Amanda contains 6 characters.
```

Another handy set of methods added by Groovy are min() and max().

Download introduction/groovy_list.groovy

```
assert names.min() == 'Amanda'
assert names.max() == 'Nate'
```

Groovy also provides a few easy ways to sort a List. The simple sort() will provide a natural sort of the elements in the List. The sort() method can also take a Closure. If the Closure has no explicit parameters, then the implied it parameter can be used in an expression to sort on. You can also give the Closure two parameters to represent two List elements, and then use those parameters in a comparison expression. Here are some examples:

Download introduction/groovy_list.groovy

```
def sortedNames = names.sort()
assert sortedNames == ['Amanda','Craig','Matthew','Nate']

sortedNames = names.sort{it.size()}
assert sortedNames == ['Nate','Craig','Amanda','Matthew']

sortedNames = names.sort{obj1, obj2 ->
  obj1[2] <=> obj2[2]
}
assert sortedNames == ['Craig','Amanda','Nate','Matthew']
```

The first example performs a natural sort on the names. The second example uses a Closure to sort the names based on their size(). The last example, though admittedly contrived, is the more interesting one. In that example we pass a Closure to the sort(). This Closure takes two parameters which represent two objects to be compared. In the body of the Closure we use the comparison operator[9] to compare some aspect of the two objects; in this case, and this is the contrived part, we compare the third character in the name with [2]. This type of sort would make more sense when the List elements are a more complex type and you need to sort on a combination of properties or a more complex expression—but you get the point.

Another useful feature of List is that the left shift operator "<<"< can be used in place of the " can be used in place of the add() method.

Download introduction/groovy_list.groovy

```
names << 'Jim'
assert names.contains('Jim')
```

Map

The Map class contains a collection of key/value pairs. It also can be created with a literal declaration, like so:

Download introduction/groovy_map.groovy

```
def family = [boys:7, girls:6, Debbie:1, Dave:1]
def empty = [:]

assert family instanceof Map
assert empty instanceof Map
assert empty.getClass().name == 'java.util.LinkedHashMap'
```

The Map class in Groovy also has the each() method. When it is given a Closure without any parameters, the implicit it will be a Map.Entry containing key and value properties. The more common approach is to give the Closure two parameters: the first parameter will hold the key and the second parameter will hold the value.

Download introduction/groovy_map.groovy

```
def favoriteColors = [Ben:'Green',Solomon:'Blue',Joanna:'Red']
favoriteColors.each{key, value ->
    println "${key}'s favorite color is ${value}."
}
```

The output from this code would be:

```
Ben's favorite color is Green.
```

9. <=> is a short-cut for the compareTo() method.

```
Solomon's favorite color is Blue.
Joanna's favorite color is Red.
```

In Groovy, Map entries can be accessed using dot notation, as if they were properties. You may have noticed that in our first Map example, we had to use empty.getClass().name instead of the Groovy shortcut empty.class.name. That's because empty.class would have looked for a key in empty called class. Other than a few edge cases like that, this is the preferred way to access Map values.

Download introduction/groovy_map.groovy

```
assert favoriteColors.Joanna == 'Red'
```

There is no overridden left shift operator for Map, but adding an element is still a snap. Assigning a value to a key that doesn't exist will add that key and value to the Map.

Download introduction/groovy_map.groovy

```
favoriteColors.Rebekah = 'Pink'
assert favoriteColors.size() == 4
assert favoriteColors.containsKey('Rebekah')
```

Set

The Set class also implements the Collection interface, so most of what we saw with List applies to it as well.Set is the default type for one-to-many associations in Grails, so we'll be working with it often. There are a couple of notable differences between Set and List: first, a Set can't contain duplicates, and second, it can't be accessed with the subscript operator ([]). This last difference can be a hindrance, but it is easy to overcome with the toList() method.

Download introduction/groovy_set.groovy

```
def employees = ['Susannah','Noah','Samuel','Gideon'] as Set
Set empty = []

assert employees instanceof Set
assert empty instanceof Set
assert empty.class.name == 'java.util.HashSet'

employees << 'Joshua'

assert employees.contains('Joshua')

println employees.toList()[3]
```

In this example we create a Set with three names in it. Since we didn't declare employees with a type, we need to cast it as a Set. (The default type for a literal declaration like this is ArrayList.) We could have just declared the type explicitly, as we do with empty on the next line. Then we add another item to the Set using the handy left shift operator, and assert() that it is there. Finally, we show that there are now four items by printing the fourth one with println employees.toList()[3]. The output from the last line of that example is: Susannah. This brings up another point about Set: You have no control of the order in which elements are stored. If you need to specify an order, either sorted or creation order, you can use a SortedSet or List.

There are many more methods added to these classes that we don't have space to cover here. To become more productive in Groovy (and to have more "Wow!" moments), check out the Groovy JDK docs at http://groovy.codehaus.org/groovy-jdk.

1.10 Where To From Here?

Now that you have some Groovy basics under your belt, we are ready to get into Grails. Over the next eleven chapters we will be exploring most areas of the Grails framework. We won't spend a great deal of time on any one feature, and we may not cover every aspect of Grails. The goal is to give you the knowledge and experience necessary to start working effectively and productively with Grails, and to point you to the resources you'll need as you continue.

"Experience?" you say. "How do I get experience from a book?" This book is not meant only to be read, but to be *used*. In the Groovy tutorial I showed some code snippets and explained them. In the rest of the book, we will be working together on a real project. By the time you finish this book, you will have developed and deployed your first full-featured web application with Grails.

Finally, at the end of the book, there is an appendix containing resources (websites, blogs, mailing lists) available in the thriving Groovy and Grails community.

Let's get started.

1.11 Acknowledgements

First and most of all I thank my creator and savior, Jesus Christ. Without Him I could do nothing and I know that every good thing I have comes from Him (James 1:17). I am also very grateful to the many