

Groovier Spring, Part 1: Integration basics

See how Groovy can add flexibility to your Spring-based applications

Skill Level: Intermediate

[Scott Leberknight \(scott.leberknight@nearinfinity.com\)](mailto:scott.leberknight@nearinfinity.com)

Chief Architect

Near Infinity Corporation

06 Jan 2009

The Spring Framework provides a solid foundation for Web and enterprise applications. Spring's support for dynamic languages like Groovy adds capabilities that can make your application architecture more flexible and dynamic. In Part 1 of this [two-part series](#), you'll learn the basics of integrating Groovy into Spring applications.

Spring 2.0 introduced support for integrating dynamic languages into Spring-based applications. Out of the box, Spring supports Groovy, JRuby, and BeanShell. The portions of your application that you write in Groovy, JRuby, or any other supported language (including, of course, the Java™ language) will integrate seamlessly into your Spring application. The rest of your application's code doesn't need to know or care about the implementation language of individual Spring beans.

Spring's support for dynamic languages means your applications can gain flexibility and dynamic behavior with no strings attached; you can have your cake and eat it too. In Part 1 of this series, you'll see how to use Spring and Groovy together, and how that powerful combination can add interesting capabilities to your application. For example, you might need to make frequent changes to small chunks of business logic, to the text contained in e-mail messages your application sends out, or to the formatting and layout of PDFs that your application generates. Traditional application architectures might require a complete redeployment of the application in order to make these kinds of changes. Spring's support for Groovy lets you make such changes to a deployed application and have them take effect immediately. I'll

discuss the benefits this capability can bring to your application and the issues that might arise. The complete source code ([see Download](#)) for all of the article's examples is available for download.

Spring's dynamic language support

Dynamic-language support has changed Spring from a Java-centric application framework to a *JVM*-centric application framework. Now Spring doesn't just make Java development easier. It also makes development for the JVM easier by allowing code written in both static and dynamic languages to plug easily into the layered architectural approach Spring supports and facilitates. If you are already familiar with Spring, then you'll feel right at home: you can take advantage of all the features Spring already provides — inversion of control (IoC) and dependency injection, aspect-oriented programming (AOP), declarative transaction demarcation, Web and data-access framework integrations, remoting, and more — while using a flexible, dynamic language like Groovy.

Spring supports dynamic-language integration via the `ScriptFactory` and `ScriptSource` interfaces. The `ScriptFactory` interface defines the mechanism for configuration and creation of scripted Spring beans. Theoretically, any language that runs on the JVM can be supported, and you could create your own implementation for your particular language of choice. `ScriptSource` defines how Spring accesses the actual script source code; for example, via the file system or a URL. Groovy language integration is supported via the `GroovyScriptFactory` implementation of `ScriptFactory`.

Why Groovy?

According to the official Groovy site, Groovy is "an agile and dynamic language for the Java Virtual Machine" that "builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk" such as dynamic typing, closures, and support for *metaprogramming* (see [Resources](#)). It is a full-fledged object-oriented programming language that can be used as such, or it can be used purely as a scripting language. I like to think of it as the Java language minus all the code I don't like writing, plus closures and other features you find in dynamic languages.

Groovy makes a particularly compelling choice to use with Spring's dynamic-language support because it was designed specifically for the JVM and with tight Java integration in mind, making it easy for Groovy and Java code to interoperate. And its Java-like syntax feels natural to Java developers.

It's time to see how to integrate Groovy code into a Spring-based application.

Groovier Spring beans

Using Groovy beans in a Spring application is just as easy as using Java beans. (You have more options, however, in how you configure them, as you'll see later.) To start, you need to define an interface to serve as the contract that your Groovy beans follow. Although it's not strictly necessary to define an interface, most Spring applications define interactions and dependencies among application components via interfaces, rather than concrete implementation classes, to promote loose coupling and easier testing.

As an example, assume you have an interface that defines how PDFs are generated from `Invoice` objects, as shown in Listing 1:

Listing 1. PdfGenerator interface

```
public interface PdfGenerator {
    byte[] pdfFor(Invoice invoice);
}
```

The `PdfGenerator` interface serves as the contract for your Groovy implementation class. Groovy classes can implement interfaces in the same manner as Java classes, so this is quite easy. Listing 2 shows the Groovy implementation of `PdfGenerator` using the `iText` library (see [Resources](#)) to do the actual PDF generation; it returns a byte array containing the PDF contents:

Listing 2. GroovyPdfGenerator

```
class GroovyPdfGenerator implements PdfGenerator {
    String companyName

    public byte[] pdfFor(Invoice invoice) {
        Document document = new Document(PageSize.LETTER)
        ByteArrayOutputStream output = new ByteArrayOutputStream()
        PdfWriter.getInstance(document, output)
        document.open()
        Font headerFont = new Font(family: Font.HELVETICA, size: 24.0, style: Font.ITALIC)
        document.add(new Paragraph("$companyName", headerFont))
        document.add(new Paragraph("Invoice $invoice.orderNumber"))
        document.add(new Paragraph("Total amount: \${invoice.total}"))
        document.close()
        output.toByteArray()
    }
}
```

`GroovyPdfGenerator` is now ready for action. It defines a string property named `companyName`, which is used on the generated PDF invoice along with the order number and total. At this point, you're ready to integrate `GroovyPdfGenerator` into your Spring application. Beans written using the Java language must be compiled into `.class` files, but you have several options when using Groovy-based beans:

- Groovy class compiled into a normal Java class file
- Groovy class or script defined in a .groovy file
- Groovy script written inline in your Spring configuration file

Depending on which of these options you use for a Groovy bean, you can choose among several ways to define and configure your Groovy beans in a Spring application context. Next, I'll explore each of these configuration options.

Groovy bean configuration

Generally, you configure Spring beans written in Java code by using XML or — as of Spring 2.5 (see [Resources](#)) — using annotations, which significantly cuts down on XML configuration. When you configure your Groovy beans, the available options depend on whether you are using compiled Groovy classes or Groovy classes defined in .groovy files. The main point to remember is that you can implement beans using Groovy and either compile them as you normally would in Java programming, or you can implement them as classes or scripts in .groovy files and let Spring take care of compiling them upon creation of the application context.

If you choose to implement your beans in .groovy files, you do *not* compile them yourself. Instead, Spring reads the files to obtain the script source code, compiles them at run time, and makes them available in the application context. This provides more flexibility than straight compilation because the .groovy files don't necessarily need to be deployed in your application's JAR or WAR file and could instead come from somewhere in the file system or perhaps from a URL.

Now you'll take a look at the various configuration options in action. Keep in mind the difference between beans defined in Groovy classes you compile as part of your own build process and beans defined in .groovy scripts.

Configuring compiled Groovy classes

Configuring a Groovy bean that has already been compiled into a .class file is exactly the same as configuring a Java-based bean. Assuming you've already compiled `GroovyPdfGenerator` using the `groovyc` compiler, you can define the bean using normal Spring XML configuration, as shown in Listing 3:

Listing 3. Configuring precompiled GroovyPdfGenerator using XML

```
<bean id="pdfGenerator" class="groovierspring.GroovyPdfGenerator">
  <property name="companyName" value="Groovy Bookstore"/>
</bean>
```

No constructor-based injection on Groovy beans

Unfortunately, you currently cannot use constructor injection to set properties on Groovy beans — or any other dynamic-language beans such as JRuby beans. One reason this is not possible is that your scripts can define multiple implementation classes and logic to select a different implementation depending on run-time environment or other factors. In other words, the script itself might do the actual construction rather than Spring. Spring uses setter injection to set properties on the returned bean. [Listing 7](#) shows an example.

The configuration in [Listing 3](#) is a plain old Spring bean definition. The fact that it is implemented in Groovy is immaterial. Any other component in a Spring application that contains the `pdfGenerator` bean can use it without knowing or caring about its implementation details or language. You can also set properties on the bean as you normally would by using the `<property>` element. (Spring 2.0 introduced the `p` namespace as a more concise way to define properties, but I've stuck with `<property>` elements because I've found them to be more readable — strictly a matter of preference.)

Alternatively, you can use annotation-based configuration of `GroovyPdfGenerator` if you are using Spring 2.5 or later. In that case, you don't actually define the bean in your XML application context; instead, you annotate your class with the `@Component` stereotype annotation, as shown in [Listing 4](#):

Listing 4. Annotating GroovyPdfGenerator with @Component

```
@Component("pdfGenerator")
class GroovyPdfGenerator implements PdfGenerator {
    ...
}
```

And you enable annotation configuration and component scanning in the Spring application context XML configuration, as shown in [Listing 5](#):

Listing 5. Enabling Spring annotation configuration and component scanning

```
<context:annotation-config/>
<context:component-scan base-package="groovierspring"/>
```

Whether you configure a compiled Groovy bean using XML or using annotations, remember that the configuration is identical to configuring normal Java-based beans.

Configuring beans from Groovy scripts

Configuring Groovy beans from `.groovy` scripts is quite different from configuring

compiled Groovy beans. This is where things start to get much more interesting. The mechanism for converting Groovy scripts into beans involves reading the Groovy script, compiling it, and making it available as a bean in the Spring application context. The first step is to define a bean whose type is ostensibly `GroovyScriptFactory` and that points to the location of the Groovy script, as shown in Listing 6:

Listing 6. Defining GroovyScriptFactory bean

```
<bean id="pdfGenerator"
      class="org.springframework.scripting.groovy.GroovyScriptFactory">
  <constructor-arg value="classpath:groovierspring/GroovyPdfGenerator.groovy"/>
  <property name="companyName" value="Groovier Bookstore"/>
</bean>
```

In this listing, the `pdfGenerator` bean is defined as a `GroovyScriptFactory`. The `<constructor-arg>` element defines the location of the Groovy script you are configuring; note specifically that this points to a Groovy *script*, not a compiled Groovy class. You set properties on scripted objects using the same syntax you normally use when defining Spring beans. The `<property>` element in Listing 6 sets the `companyName` property just as you would expect.

The `GroovyPdfGenerator.groovy script` must contain at least one class that implements your interface. Usually it is best to follow standard Java practice by defining one Groovy class per `.groovy` file. However, you might want to implement logic within the script to determine which type of bean to create. For example, you could define two different implementations of the `PdfGenerator` interface in `GroovyPdfGenerator.groovy` and perform logic directly in the script to determine which of those implementations should be returned. Listing 7 defines two different `PdfGenerator` implementations and selects the one to use based on a system property:

Listing 7. Multiple class definitions in a Groovy script

```
class SimpleGroovyPdfGenerator implements PdfGenerator {
  ...
}

class ComplexGroovyPdfGenerator implements PdfGenerator {
  ...
}

def type = System.properties['generatorType']
if (type == 'simple')
  return new SimpleGroovyPdfGenerator()
}
else {
  return new ComplexGroovyPdfGenerator()
}
```

As this snippet shows, you could use a scripted bean to select a different

implementation based on a system property. When the `generatorType` system property is `simple`, the script creates and returns a `SimpleGroovyPdfGenerator`; otherwise, it returns a `ComplexGroovyPdfGenerator`. Because both the simple and complex implementations implement the `PdfGenerator` interface, code using the `pdfGenerator` bean from within your Spring application will neither know nor care what the actual implementation is.

Note that you can still set properties, as in [Listing 6](#), on the bean returned from your script. So if the script returns a `ComplexGroovyPdfGenerator`, the `companyName` property will be set on that bean. When you don't need the additional flexibility of defining multiple implementations, you can define just one class in the Groovy script file, as shown in [Listing 8](#). In this case, Spring finds this one class and instantiates it for you.

Listing 8. Typical Groovy script implementation

```
class GroovyPdfGenerator implements PdfGenerator {  
    ...  
}
```

At this point, you might be wondering why [Listing 6](#) defines the bean as a `GroovyScriptFactory`. The reason is that Spring creates scripted objects via a `ScriptFactory` implementation — in this case a Groovy factory — combined with a `ScriptFactoryPostProcessor` bean, which is responsible for replacing the factory beans with the actual object created by the factory. [Listing 9](#) shows the additional configuration that adds the postprocessor bean:

Listing 9. Defining the ScriptFactoryPostProcessor bean

```
<bean class="org.springframework.scripting.support.ScriptFactoryPostProcessor"/>
```

When Spring loads the application context, it first creates the factory beans (for example, the `GroovyScriptFactory` beans). Then the `ScriptFactoryPostProcessor` bean goes through and replaces all the factory beans with the actual scripted objects. For example, the configuration shown in [Listing 6](#) and [Listing 9](#) results in a bean named `pdfGenerator` whose type is `groovierspring.GroovyPdfGenerator`. (If you turn on debug-level logging in Spring and watch the application context start up, you'll see that Spring first creates a factory bean named `scriptFactory.pdfGenerator`, and later the `ScriptFactoryPostProcessor` creates the `pdfGenerator` bean from that factory bean.)

Now that you know the low-level details involved in configuring scripted Groovy beans using `GroovyScriptFactory` and `ScriptFactoryPostProcessor`, I'll

show you a much simpler and cleaner way to accomplish the same result. Spring provides the `lang` XML schema specifically for creating scripted beans. Listing 10 defines the `pdfGenerator` bean using the `lang` schema:

Listing 10. Defining a scripted bean using `<lang:groovy>`

```
<lang:groovy id="pdfGenerator"
  script-source="classpath:groovierspring/GroovyPdfGenerator.groovy">
  <lang:property name="companyName" value="Really Groovy Bookstore"/>
</lang:groovy>
```

This code results in the same `pdfGenerator` bean as the more verbose configuration in [Listing 6](#) and [Listing 9](#) but is cleaner and more concise, and it makes the intent more clear. The `<lang:groovy>` bean definition requires the `script-source` attribute; this tells Spring how to locate the Groovy script source code. In addition, you can use the `<lang:property>` element to set properties for scripted beans. Defining Groovy-based beans using `<lang:groovy>` is a better option and more clear to anyone reading your Spring configuration.

Configuring inline Groovy scripts

For the sake of completeness, I'll mention that Spring also supports writing Groovy scripts directly in your bean definitions. Listing 11 creates the `pdfGenerator` bean using an inline script:

Listing 11. Defining a scripted bean inline

```
<lang:groovy id="pdfGenerator">
  <lang:inline-script>
    <![CDATA[
      class GroovyPdfGenerator implements PdfGenerator {
        ...
      }
    ]]>
  </lang:inline-script>
  <lang:property name="companyName" value="Icky Groovy Bookstore"/>
</lang:groovy>
```

This snippet defines the `pdfGenerator` bean using `<lang:groovy>` and the `<lang:inline-script>` tag, which contains the Groovy script defining the class. You can set properties as before using `<lang:property>`. As you might have guessed, I don't recommend defining scripted beans inside XML configuration files (or any type of code inside XML files for that matter).

Configuring beans using the Grails Bean Builder

The Grails Web framework relies on Spring under the covers. Grails provides the Bean Builder, a cool feature that gives you a way to define Spring beans *programmatically* using Groovy code (see [Resources](#)). Defining beans

programmatically potentially provides more flexibility than XML configuration because you can embed logic in the bean-definition script, which is impossible to do in XML. Using Bean Builder, you can create bean definitions for both compiled Groovy classes and scripted Groovy beans. Listing 12 defines the `pdfGenerator` bean using a compiled Groovy class:

Listing 12. Using Bean Builder to define compiled Groovy bean

```
def builder = new grails.spring.BeanBuilder()
builder.beans {
    pdfGenerator(GroovyPdfGenerator) {
        companyName = 'Compiled BeanBuilder Bookstore'
    }
}
def appContext = builder.createApplicationContext()
def generator = context.pdfGenerator
```

The code in [Listing 12](#) first instantiates a `BeanBuilder` and then makes method calls to create the beans. Each method call and optional closure argument defines a bean and sets bean properties. For example, `pdfGenerator(GroovyPdfGenerator)` defines a bean named `pdfGenerator` of type `GroovyPdfGenerator`, and the code inside the closure sets the `companyName` property. You can of course define multiple beans inside the `beans` closure.

Using Bean Builder, you can also create beans from Groovy scripts, as opposed to compiled Groovy classes. However, Bean Builder has no equivalent to the syntactic sugar you saw with the `<lang:groovy>` configuration, so you need to define the beans as `GroovyScriptFactory` and also create a `ScriptFactoryPostProcessor` bean. Listing 13 shows an example of using Bean Builder to configure a scripted Groovy bean:

Listing 13. Using Bean Builder to define scripted Groovy bean

```
def builder = new grails.spring.BeanBuilder()
builder.beans {
    pdfGenerator(GroovyScriptFactory,
        'classpath:groovierspring/GroovyPdfGenerator.groovy') {
        companyName = 'Scripted BeanBuilder Bookstore'
    }
    scriptFactoryPostProcessor(ScriptFactoryPostProcessor)
}
def appContext = builder.createApplicationContext()
def generator = context.pdfGenerator
```

The code in [Listing 13](#) is logically equivalent to the XML configuration in [Listing 6](#) and [Listing 9](#), though of course [Listing 13](#) uses Groovy code to define the beans. To define the `pdfGenerator` bean, [Listing 13](#) specifies the type as `GroovyScriptFactory`. The second argument specifies the location of the script source and, as before, sets the `companyName` property inside the closure. It also

defines a bean named `scriptFactoryPostProcessor` of type `ScriptFactoryPostProcessor`, which will replace the factory beans with the actual scripted objects.

Which configuration option is best?

You've now seen several different ways to configure Groovy-based beans, whether they are compiled or scripted. If you are simply using Groovy to replace the Java language as the primary language in your application, then configuring the beans is no different from configuring Java-based beans. You can use either XML or annotation-based configuration for compiled Groovy classes.

For scripted Groovy objects, even though you can configure them in several ways, the `<lang:groovy>` option is the cleanest way to configure them and makes the intent most clear, as opposed to configuration using `GroovyScriptFactory` and `ScriptFactoryPostProcessor` or using `<lang:inline-script>`.

You also saw the Grails Bean Builder, which is a completely different way to create a Spring application context from what most Spring applications use. If you want to create all your beans in Groovy and be able to add logic into the bean-building process, the Bean Builder might fit the bill nicely. On the other hand, defining Groovy beans using Bean Builder requires you to define the beans using `GroovyScriptFactory` and `ScriptFactoryPostProcessor`.

Using Groovy beans

Bean configuration and the various options available to you is the hard part of integrating Groovy and Spring (though as you've seen it isn't really very difficult). Actually using Groovy beans in your Spring application is easy. In fact, Spring's dynamic-language support makes using the beans completely transparent to application code, which neither knows nor cares about the implementation details. You write application code as you normally do in a Spring application, and you can take advantage of all the features Spring normally provides such as dependency injection, AOP, and integration with many third-party frameworks.

Listing 14 shows a simple Groovy script that creates a Spring application context from an XML configuration file, retrieves the PDF generator bean, and uses it to generate a PDF version of an invoice:

Listing 14. Using Groovy beans in a script

```
def context = new ClassPathXmlApplicationContext("applicationContext.xml")
def generator = context.getBean("pdfGenerator")

Invoice invoice = new Invoice(orderNumber: "12345", orderDate: new Date())
invoice.lineItems = [
```

```
        new LineItem(quantity: 1, description: 'Groovy in Action (ebook)', price: 22.00),
        new LineItem(quantity: 1, description: 'Programming Erlang', price: 45.00),
        new LineItem(quantity: 2, description: 'iText in Action (ebook)', price: 22.00)
    ]

    byte[] invoicePdf = generator.pdfFor(invoice)

    FileOutputStream file = new FileOutputStream("Invoice-${invoice.orderNumber}.pdf")
    file.withStream {
        file.write(invoicePdf)
    }
    println "Generated invoice $invoice.orderNumber"
```

In [Listing 14](#), most of the code is related to creating the Spring `ApplicationContext`, creating the invoice, and writing it out to a file. Using the `pdfGenerator` bean to generate the invoice is only one line of code. In typical Spring applications, you bootstrap the application context once, at application startup, and from that point on your components simply use the dependencies that Spring provided them. In Spring Web applications, you can configure a servlet context listener, and Spring is bootstrapped when the application starts. As an example, a PDF invoice-generation service might be defined as shown in [Listing 15](#):

Listing 15. Service class that uses a PDF generator

```
@Service
public class InvoicePdfServiceImpl implements InvoicePdfService {

    @Autowired
    private PdfGenerator pdfGenerator;

    public byte[] generatePdf(Long invoiceId) {
        Invoice invoice = getInvoiceSomehow(invoiceId);
        return pdfGenerator.pdfFor(invoice);
    }

    // Rest of implementation...
}
```

The `InvoicePdfServiceImpl` class in [Listing 15](#) happens to be implemented as a Java class that depends on a `PdfGenerator`. It could just as easily been implemented as a Groovy bean. You can use the `GroovyPdfGenerator` implementation with any of the compiled or scripted bean configurations, and `InvoicePdfServiceImpl` is none the wiser. So, you can see that using Groovy (or any dynamic language) beans is transparent to application code. This is a good thing because it results in loose coupling between components, makes unit testing much easier, and allows you to use the most appropriate implementation language for the job.

Conclusion to Part 1

You've now seen several different ways to configure Groovy language beans and

how easy it is to use them in your Spring-based applications. You can use compiled Groovy classes, which is exactly the same as using Java classes. You've also seen several different ways to configure scripted Groovy objects. The option you should choose depends on how you are using Groovy in your application. You can also combine compiled and scripted Groovy beans in the same application. In fact, if you really wanted, you could have Java, Groovy, JRuby, and BeanShell beans all in the same application, though I wouldn't recommend doing this. As a developer, you must weigh the pros and cons of having multiple languages in your application.

Groovy's flexibility as a language when compared to the Java language makes it an attractive choice, even if you only choose to compile your Groovy classes. Spring's ability to integrate scripted dynamic language beans makes Groovy an even more compelling choice because you can introduce additional logic and flexibility in scripted beans. For example, as you saw earlier, you can add logic to determine the type of bean to instantiate at application startup based on business logic. Or you can gain flexibility in Web application deployment by deploying scripted objects in .groovy files that are in the application CLASSPATH or somewhere in the file system but are not packaged in the WAR file.

Everything you've seen so far adds flexibility and power to your Spring toolkit. But probably the most compelling feature in Spring's dynamic-language support is the ability to monitor and detect changes to dynamic-language scripts *while your application is running* and *automatically reload* the changed beans in the Spring application context. [Part 2](#) thoroughly explores this capability, which adds significantly more flexibility than using a static configuration containing beans that can't change at run time.

Downloads

Description	Name	Size	Download method
Sample code	j-groovierspringcode	8.5 MB	HTTP

[Information about download methods](#)

Resources

Learn

- [Spring Framework](#): Spring from the source.
- [The Spring Series](#) (Naveen Balani, developerWorks, 2005): This four-part series introduces the Spring framework.
- ["What's New in Spring 2.5?"](#) (Mark Fisher, InfoQ, November 2007): Read about the latest improvements in the Spring Framework.
- [Spring Dynamic Language Support](#): This chapter in the Spring reference guide covers dynamic language support in Spring.
- [Groovy](#): Visit the home of the Groovy language.
- [Practically Groovy](#): This developerWorks series is dedicated to exploring the practical uses of Groovy and teaching you when and how to apply them successfully.
- [Programming Groovy](#) (Venkat Subramaniam, The Pragmatic Programmers, 2008): This book contains lots of advice for using the metaprogramming features in Groovy.
- [Grails Bean Builder](#): Programmatically build your Spring beans.
- [iText PDF Library](#): Use the iText library to create and work with PDF documents easily.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.
- Browse the [technology bookstore](#) for books on these and other technical topics.

Get products and technologies

- [Spring](#): Download the latest Spring release.
- [Groovy](#): Download the latest Groovy release.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Scott Leberknight

Scott Leberknight has been developing software professionally for 14 years. He is

Chief Architect at Near Infinity Corporation and speaks frequently at the No Fluff Just Stuff conference series and other developer conferences.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.