Getting Groovy in an SOA

Grandeur

Study

Projectgroup	Grandeur
Date	January 21, 2009
School	Hogeschool van Arnhem en Nijmegen
Training	Informatie Communicatie Acedemie
Minor	Enterprise Application Development
Subject	Getting Groovy in an SOA
Supervisor	Sander Leer
Version	1.3

Summary

This paper is aimed at answering the following question:

'What are the characteristics of Groovy (and Grails) and what impact do they have for an implementation in an SOA within enterprise applications?'

Groovy is a dynamic language and utilizes the Java Virtual Machine. Its dynamic nature lies in the ability to alter its classes at runtime, thus allowing for constant change. Due to its foundation in Java, it can cooperate with and enrich the existing Java libraries.

One of the more important features that Groovy provides, is the Meta-Object Protocol, which allows Groovy to perform its dynamic capabilities. In essence, each object has an accompanying Meta-Object which contains all properties in the form of a Map, thus allowing it to scale at runtime as needed.

When using Groovy for constructing webservices, some additional modules can be used. When using SOAP as the desired transport mechanism, the GroovyWS-module provides functionality that abstracts all of the low level transport operations through a simple interface. However, it is still in development and support is therefore minimal. Furthermore, only a very small part of the WS-Security stack is implemented which minimizes the developers choice for securing the webservices.

For RESTful webservices, Groovy's webframework Grails can be used. It has full support for all of the HTTP-request methods and provides url mapping. For exposing the SOA to its target audience, the Grails framework offers a quick start for developing a web interface that can interact with the SOA. Scaffolding allows Grails to generate both controllers and views based on the domain classes in the model, enabling rapid development. By default, all domain classes are persisted to a datastore through Grails Object Relational Mapping.

We believe that Groovy is mature enough to be used in a production environment, however Groovy's modules do not provide all needed functionality for SOA development, and therefor Java frameworks are still required.

Table of contents

Foreword

The world of IT is a constant revolving cycle where new and old ideas are used into ever emerging technologies, where some find a silent end and others find great success. Java was one of those success stories, and has gained a large community over time, supplying it with numerous frameworks that enchant its functionality and strength, making it one of the most common programming languages in use these days.

Java has also inspired people to develop alternatives, modifying and replacing ideas of Java and other languages as they see fit, creating the new generation of programming languages. One of these languages is Groovy. An interesting fact about Groovy, that might make it more interesting then other fresh and emerging languages, is that Groovy code compiles to Java byte code. This, and the rather intriguing name were enough for us to use it as a research subject.

The result of this research is a paper of approximately fifty pages, in the form of the document you are reading right now. But before we end this foreword and start writing about the core questions that have driven our research, there are a few people we would would like to thank for their contribution and support through the entire project:

- Rody Middelkoop, lecturer at the HAN
- Sander Leer, supervisor and lecturer at the HAN
- Peter Schuszler, lecturer at the HAN
- Paul Bakker, trainer/consultant at Info Support

We believe to have written a decent paper that will answer a few important questions about Groovy, and show you how it can contribute to faster and more fun development along the way. We hope our readers will feel the same, give us feedback if they desire, and might even introduce a bit a Groovyness in their everyday life.

Best regards,

Grandeur

1

Introduction

" Personal beauty is a greater recommendation than any letter of introduction. "
- Aristotle

This study is the result of five months of research we have done on the young programming language called Groovy.

The research has been performed by a project group called “Grandeur”. This group consists of the following members:

- Youssef El Messaoudi
- Gaya Kessler
- Marco Kuiper
- Jaap Mengers
- Bart van Zeeland

Subject

This research is aimed at the programming language Groovy, a dynamic language that works on top of the Java Virtual Machine. Groovy, developed in 2003 by James Strachan and Bob McWhirter [28], is a language that is very similar to languages like Ruby, Smalltalk, Python and Perl.

One of the aspects that this research is aimed at, is the usage of Groovy in an SOA. In this context, the position of Groovy is investigated as a service provider and consumer.

Other aspects that will be covered, are the characteristics of Groovy; the dynamic nature of the language and remarkable syntax that makes Groovy unique.

Finally, the Grails-framework will be covered. This framework helps developers build web applications based on Groovy's variant of the Java Server Pages.

Boundaries

Covering all aspects of Groovy, SOA, dynamic languages, Java and others, would be beyond the scope of this project . Therefore, the following boundaries are set:

- This study focusses solely on Groovy. Although some comparisons are made with Java – after all Groovy runs on the JVM – the emphasis is on Groovy. Other programming languages are outside the scope of this research.
- Groovy can provide a solution for several software problems. This study is only targeted at using Groovy as a software solution inside an SOA in the form of a webservice.
- There are several modules available for Groovy. To support the main question of the research, we'll only cover the following modules for Groovy:
    - Groovy SOAP
    - GroovyWS
    - GSP
    - Grails
    - GORM

Expected knowledge

Readers are expected to have the following knowledge before they will be able to grasp the covered subjects.

- A general understanding of the SOA paradigm.
- Basic knowledge of Java.

Problem Definition

This research is aimed at Groovy and its role in an enterprise environment between large and established platforms like Java and .NET. Because of Groovy's relative young age, not much experience is gathered about Groovy in large, enterprise-wide SOA's.

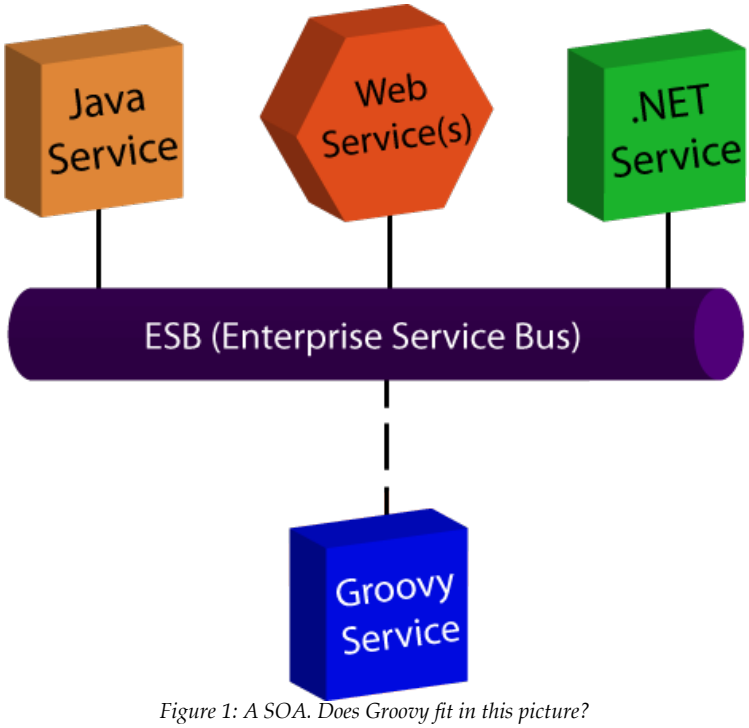

Figure 1: A SOA. Does Groovy fit in this picture?

1.1 Objective

The goal of this research is to investigate whether Groovy and Grails can be used as alternatives for common platform like Java and .NET [29] inside an enterprise environment..

This objective came about since Groovy is a fairly new language. Languages like Java and .NET have already proven themselves that they can be used inside a SOA, but does this count for Groovy (Figure 1: A SOA. Does Groovy fit in this picture?)?

1.2 Research Questions

With this objective in mind, we can conclude the following research question:

'What are the characteristics of Groovy (and Grails) and what impact do they have for an implementation in an SOA within enterprise applications?'

There are separate questions made to support the main question to come to an accurate answer.

1. What is Groovy?
    1. What are the characteristics of Groovy?
    2. What is the MOP (Meta Object Protocol)?
1. Which mechanisms does Groovy provide for communication with web services?
    3. How does Groovy handle XML documents?
    4. Which mechanisms does Groovy provide for SOAP-based services?
    5. Which mechanisms does Groovy provide for RESTful services?
2. Which features does Groovy provide for securing webservices?
3. Which possibilities does Groovy provide as a server side technology to build web applications?

All of the answers to the questions above can be found in this document.

2

Groovy

" Any fool can write code that a computer can understand. Good programmers write code that humans can understand. "
- Martin Fowler

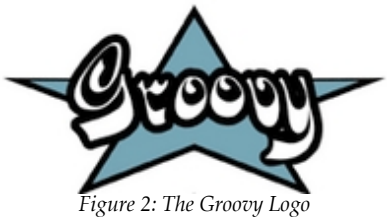

Figure 2: The Groovy Logo

> " Groovy...
>
> - is an agile and **dynamic language** for the **Java Virtual Machine**
> - builds upon the strengths of Java but has **additional power features** inspired by languages like Python, Ruby and Smalltalk
> - makes **modern programming features** available to Java developers with **almost-zero learning curve**
> - supports **Domain-Specific Languages** and other compact syntax so your code becomes **easy to read and maintain**
> - makes writing shell and build scripts easy with its **powerful processing primitives**, OO abilities and an Ant DSL
> - increases developer productivity by **reducing scaffolding code** when developing web, GUI, database or console applications
> - **simplifies testing** by supporting unit testing and mocking out-of-the-box
> - seamlessly **integrates with all existing Java objects and libraries**
> - compiles straight to Java bytecode so you can use it anywhere you can use Java
>
> [Source: http://groovy.codehaus.org/] "

Groovy is an object-oriented programming language for the Java Platform [8] (*Figure 3: The Java Platform*), that appeared in 2003. James Strachan and Bob McWhirter were the first developers in this program. The first "1.0" version of Groovy was released on January 2, 2007. Guillaume Laforge is the current project manager [27].

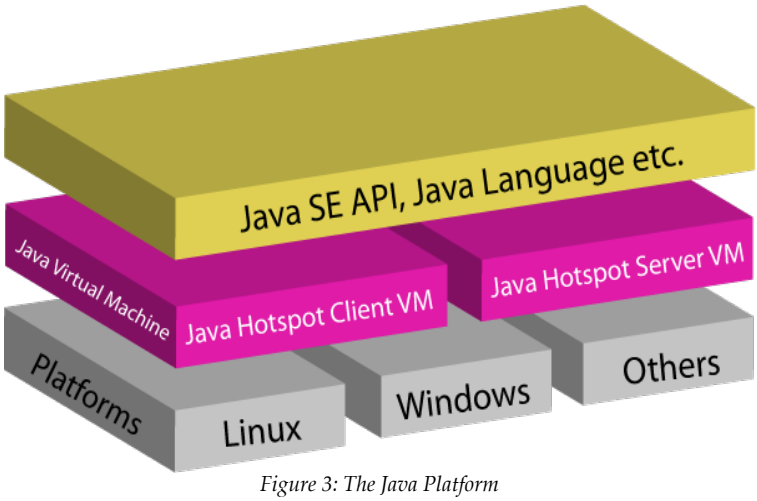

Figure 3: The Java Platform

Groovy uses a Java-like syntax which is dynamically compiled to Java byte code [7]. For this reason, Groovy can seamlessly work together with Java code and Java libraries (*Figure 4: Java Platform incl. Groovy*). This combination makes the use of Groovy really powerful; all Java (.java) files can be converted to Groovy files (.groovy) and the application should still compile and work [27]. Groovy has some unique, powerful features that can now be used in the code.

This also works the other way around; when an application needs functionality that can't be achieved with the existing Groovy libraries, the developer can still write Java code to achieve the goal.

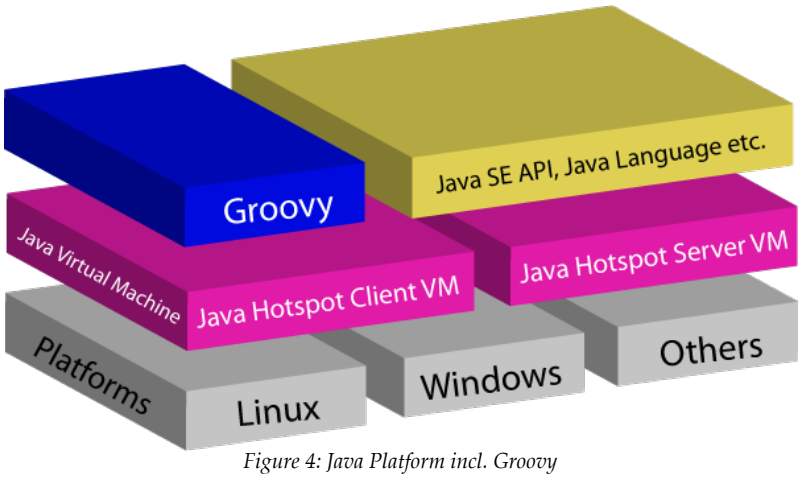

Figure 4: Java Platform incl. Groovy

> " Groovy is like a super version of Java. It can leverage Java's enterprise capabilities but also has cool productivity features like closures, builders and dynamic typing. If you are a developer, tester or script guru, you have to love Groovy.
>
> [Source: http://groovy.codehaus.org/] "

2.1 What are the characteristics of Groovy?

2.1.1 Dynamic Language

There are a few aspects of a programming language that highly infuence what is possible within the language. One of these is on what level the language functions. A language like C talks directly to the OS [30], and hereby is capable of things impossible in Java, like directly accessing the networking sockets or reading and manipulating the packages that are send over the network. This is something that can't be done in a language like, for example, Groovy.

Groovy, on the other end, runs on a VM [4] that does the memory allocation and garbage collection for you. These are things that are not readily available in C and are usually done manual or by adding middle-ware. Another aspect is whether the language is dynamic or static. C is a prime example of a static language [37]: it doesn't allow runtime manipulation of the code. Groovy, on the other hand, does allow this [38] , giving the programmer more flexibility.

But defining what makes a language like Groovy dynamic is more then just saying that it allows such runtime mutation of it's code, although this is one of the most important features [31]. Other features may involve;

- Dynamic typing [31]
- Late binding [31]
- Eval support [32]

Languages that support these features are not always dynamic though, many static languages are working to provide the developer with more and more dynamic features [39] , although they are are still static at their core. Defining a static language from a dynamic one gets more difficult this way [31] , but the most fundamental feature (allowing the manipulation of code while executing) is still limited to truly dynamic languages.

One of the best reasons to use a dynamic language over a static one is its flexibility [40] . Because the code is able to change at runtime, it can adapt itself when needed. Calls to non-existing function can be used to generate these function at runtime and actually return a useful result instead of a cryptic error. This same kind of functionality can be used to mock almost anything [41], saving the developer from having to type them all out and thus increasing their development speed. Developing in a dynamic language is usually faster as well. Because the syntax is shorter and more expressive, a function in a dynamic language is usually shorter then one in a static language, shaving of development time.

Dynamic programming languages have downsides as well: all of the compiling has to be done at runtime, and this takes it's toll on the speed of the software [30]. Normal algorithms to speed up this compiling process can't be applied either, since the code can be changed dynamically at almost anytime. This decreases the overall speed as well. The dynamic nature of a language also makes it more difficult for IDE's to provide decent support to the developer[31], making a good IDE harder to come by.

Since both of the most common language platforms today (Java and .NET) are moving towards becoming more dynamic, the downsides that most dynamic languages know today, will probably fade, making dynamic languages a more attractive choice. An increase of processing power in consumer grade computers and a demand for faster development will also contribute to a higher rate of acceptation within the software industry, creating a larger platform for dynamic languages themselves.

2.1.2 Groovy Development Kit

Groovy extends the JDK with extra methods and closure support. Basically, Groovy still uses the good old Java classes like java.util.ArrayList. These classes have more convenient methods.

For example: Java has a java.util.ArrayList class. If this arraylist contains strings which have to be joined, some sort of loop would have to be written and the result would have to be build up by hand. Groovy however, adds a join-method to the ArrayList, which can do this automatically.

2.1.3 Imports

Groovy automatically imports a series of packages and classes that are often used, so that they can be used immediately without the need to be implicitly imported [3] . These include:

- groovy.lang.*;
- groovy.util.*;
- java.lang.*;
- java.util.*;
- java.net.*;
- java.io.;
- java.Math.BigInteger;
- java.Math.BigDecimal;

Furthermore, Groovy makes a number of methods available in each Object, so that they can be called without the need to use their fully qualified name. The one most common used is '`System.out.print`', which can be called with '`print`' throughout the entire code.

2.1.4 Semicolons

The use of semicolons (`;`) in Groovy is completely optional [3]. The only time a semicolon has to be used, is when multiple function calls are placed on the same line.

```
print ("Hello")
print (" world!")
// Will print "Hello world!"

print ("Hello"); print (" world!");
// Will print "Hello world!"
```

2.1.5 Data type declaration

When creating an object, it's type doesn't have to be defined explicitly. By using the `def`-keyword, Groovy will automatically detect what object type has to be used.

```
/* Used in scripts */
word = "Hello World!"
print word.class
// class java.lang.String

/* Used in compiled classes */
def number = 10
print number.class
// class java.lang.Integer
```

Although optional, Groovy still enables object types to be declared explicitly. This might be useful in situations where only one data type is allowed.

In the following example, the method `addAmount` is called. The parameter it expects is of type int, since the method performs a calculation based on the input and returns the result.

```
def addAmount(int deposit)
{
      return 100 + deposit
}

print addAmount(20)
// 120
print addAmount("String")
// addAmount() is applicable for argument types:
// (java.lang.String) values: {"String"}
```

If `def deposit` would be used in this context, the last return value would be `100String` with the class `java.lang.String`. When an action is performed on a String and an Integer, the String is always used. Groovy doesn't look at the order the variables are given.

```
number = 10
print number.class
// class java.lang.Integer

string = "Hello World!"
print string.class
// class java.lang.String

print number + string
// 10Hello World!
print string + number
// Hello World!10
```

2.1.6 Groovy String

The Groovy String (Also called "GString") allows for any type of logic to be integrated in the String definition [5] . This can be done with the dollar symbol (**$**) and (optional) braces (**{** **}**). The difference between a String and a GString is automatically recognized by Groovy.

```
word = "Hello World!"
print word.class
// class java.lang.String

name = "World"
gstring = "Hello ${name}!"
print gstring.class
// class org.codehaus.groovy.runtime.GStringImpl
```

The dollar sign can be escaped with a backslash, to use it's String representation when printing a currency. In combination with Heredocs – further explained in chapter 2.1.8 – a GString can be very powerful and useful for programming.

```
honorific = "Mr. "
name = "Foo"
currentamount = 100.00
deposit = 20.00
gstring = """Hello ${honorific + name},
Amount of money you had:  € $currentamount
Your deposit is:          € $deposit
Your current amount is:   € ${currentamount + deposit}"""

print gstring

/* Above will output:
Hello Mr. Foo,
Amount of money you had:  € 100.00
Your deposit is:          € 20.00
Your current amount is:   € 120.00 */
```

As shown in the example, curly braces are only needed when an operation is performed as part of the result.

2.1.7 Embedded quotes

Groovy has a nice way of working with Strings. In Java, a single quote would represent the primitive type char. In Groovy, anything that is surrounded by either single or double quotes, is converted to a String [8].

This is very useful when working with Strings that contain quotes. As this example will show, Strings with doubles quotes in it can be surrounded with single quotes en vice versa. To escape a quote, a backslash is used.

```
embed_singlequote = "Hello 'World'"
print embed_singlequote
// Hello 'World'

embed_doublequote = 'Hello "World"'
print embed_doublequote
// Hello "World"

embed_escapequote = "Hello \"World\""
print embed_escapequote
// Hello "World"
```

Escaping a quote is not necessary when heredocs are used.

2.1.8 Heredocs

Heredocs present an easy way of declaring a multiline String in an easy way [3] . Furthermore, it can contain both single and double quotes, which don't need to be escaped.

A Heredoc is declared by surrounding a String-value with three double quotes on either side.

```
Heredoc = """<img src="image.jpg" alt="Let's get Groovy" />"""
print heredoc
// <img src="image.jpg" alt="Let's get Groovy" />
```

2.1.9 Collections

Groovy acknowledges three different types of collections [5]. The first two, Lists and Maps, are no different from the ones used in Java. Lists use a null-based index to retrieve items, whereas Maps use a unique key to find an item. Ranges however, are more or less unique to dynamic languages.

A simple example of a list in Groovy is:

```
def roman = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII']
```

The first entry being zero in Roman, which is the word 'nulla' and doesn't have a notation.

A map is created by assigning values to a corresponding key, like such:

```
def http = [
      100 : 'CONTINUE',
      200 : 'OK',
      400 : 'BAD REQUEST'
]
```

Although ranges don't appear in the standard Java libraries, most programmers have an intuitive idea of what a range is. Effectively, a range defines a start and an end point, with a notion of how to move from the start to the end point.

Groovy provides literals to support for ranges, along with other language features such as the for statement, which understands ranges.

Declaring a range is easy:

```
def x = 1..10
```

2.1.10 Declaring Classes

Classes are the cornerstone of object-oriented programming, because they define the blueprint from which objects are drawn.

The code below contains a simple Groovy class named Book, which has an instance variable title, a constructor that sets the title, and a getter method for the title.

By default, all methods are public and therefore access modifiers are excluded.

```
class Book {
    private String title

    Book (String theTitle) {
        title = theTitle
    }

    String getTitle(){
        return this.title
    }
}

def HarryPotter = new Book("Harry Potter 11")

print "The name of the book: " + HarryPotter.getTitle();
```

2.1.11 Return Statements

The last line of a method in Groovy is automatically the return statement [3]. For this reason, an explicit return statement can be left out.

```
def getFoobar()
{
      "Foobar"
}

print getFoobar()
// Foobar
```

To return a value that is not on the last line, the return statement has to be declared explicitly.

```
def getGreet(String name)
{
      if(name != null)
            return "Hello $name!"
      else
            return "Hello anonymous!"
}

print getGreet("World")
// Hello World!
print getGreet()
// Hello anonymous!
```

In the example above you'll see one of the usages of the GString, which was explained in chapter 2.1.6. Checking for the `null` value in Groovy can be much simpler as coded in the example, as you'll can see in the next chapter.

2.1.12 Checking NULL value

Null-checking is a tedious, yet necessary operation, especially when a long chain of get-methods is used to retrieve a variable deep inside of an object-hierarchy and each of them can be null.

Groovy simplifies this process by providing a syntax where null-checking can be done automatically when a property is referenced or a method call is made. It does this by using a question mark after each potential null-value, like such:

```
string = "Hello World!"
print string.size()
// 12

empty = null
print empty.size()
// java.lang.NullPointerException:
// Cannot invoke method size() on null object

print empty?.size()
// null
```

When encountering a question mark, Groovy will check if the preceding variable is null. If so, it will end the operation and skip to the next statement. If not, it will continue execution on the current operation, where it might encounter another question mark, after which the process will be repeated.

```
class Person
{
      def bankaccount = new Bankaccount()
}

class Bankaccount
{
      def getCurrentamount()
      {
              100.00
      }
}

person = new Person()
// Java: if (person != null && person.bankaccount != null
              // && person.bankaccount.getCurrentamount() != null)
println person?.bankaccount?.currentamount
```

2.1.13 Boolean

Every object in Groovy has a boolean representation, which value depends on it's content and type. A String for instance, will return true if populated and false if empty. This allows for quick "truth"-checking, and reduces the amount of code involved.

```
// Strings
if ("Hello")        // True:  String is not empty
if ("")             // False: String is empty

// Integers
if (1)              // True:  All non-0 (zero) values are true
if (-1)
if (0)              // False: 0 (zero) values are false

// null
if(!null)           // True:  All non null values are true
if(null)            // False: All null values are false

// Lists
list = ["value 1","value 2","value 3"]
if (list)          // True:  Array length is greater than 0 (zero)
list = []
if (list)          // False: Array length is less than 0 (zero)

// Maps
person = [firstname:"Foo",lastname:"Bar"]
if (person)         // True:  Map is populated
person = [:]
if (person)         // False: Map has no population
```

2.1.14 Operator overloading

Operator overloading allows the default use of operators to be overridden to enable a more intuitive approach for common methods [27]. The way Groovy implements this, can be seen in the following table [3].

```
a == b or a != b        -        a.equals(b)
a + b                   -        a.plus(b)
a - b                   -        a.minus(b)
a * b                   -        a.multiply(b)
a / b                   -        a.div(b)
a % b                   -        a.mod(b)
a++ or ++a              -        a.next()
a-- or --a              -        a.previous()
a & b                   -        a.and(b)
a | b                   -        a.or(b)
a[b]                    -        a.getAt(b)
a[b] = c                -        a.putAt(b, c)
a << b                  -        a.leftShift(b)
a >> b                  -        a.rightShift(b)
a < b or a > b
a <= b or a >= b        -        a.compareTo(b)
```

Groovy classes that implement any of these methods, automatically enable the use of the corresponding operator, without the need to implement an interface like in Java.

2.1.15 Parentheses

In Groovy's effort to make development simpler and quicker, it allows parentheses in method calls to be omitted, provided that the method expects arguments [3]. When a method call is made without both parentheses and arguments, Groovy interprets this as a reference to a property, and will throw an exception if this property is non-existent.

```
def present(name)
{
    println "Hi! My name is " + name + ", nice to meet you."
}

present 'G. R. Oovy'
// Will print "Hi! My name is G. R. Oovy, nice to meet you."

present 'G. R. Oovy'.toLowerCase()
// Will print "Hi! My name is g. r. oovy, nice to meet you."

present 'G. R. Oovy'.toLowerCase
// groovy.lang.MissingPropertyException: No such property: toLowerCase
```

2.1.16 Closures and blocks

One of the most interesting aspects of Groovy's dynamic features, is the closure [8]. A closure can be defined as a function wrapped in a variable, thus allowing it to be used as an argument in a method-call.

Closures can access variables that are defined in the same scope in which the closure is defined.

```
def name = "" //initialize variable
def printName = { println "The string in the name variable is " + name } //define method
name = "Youssef" //set string Youssef in variable name
printName() //result: The string in the name variable is Youssef
name = "Gaya" //set string Gaya in variable name
printName() //result: The string in the name variable is Gaya
```

In order for a closure to use variables that are outside it's scope, arguments can be passed to it. If only one argument is used, it is automatically and implicitly named '`it`' and can be referenced as such.

```
def name = "" //initialize variable
def printName = { println "The string in the name variable is " + it } //define method
name = "Youssef" //set string Youssef in variable name
printName(name) //result: The string in the name variable is Youssef
name = "Gaya" //set string Gaya in variable name
printName(name) //result: The string in the name variable is Gaya
```

When more influence on the naming of the arguments is preferred, or multiple arguments are used, these must be added to the closure, preceding the closure's body and separated with '->'.

```
def printName = { nameToPrint -> println "The string in the name variable is " + nameToPrint }
```

Groovy's closures are ordinary POGO's, and can therefore be nested, as this example will show.

```
def startTimer() {
      def initialDate = new java.util.Date()
      return {
              println "${initialDate} - ${new java.util.Date()} : Elapsed time $
{System.currentTimeMillis() - initialDate.time}"
      }
}
def timer = startTimer()
timer()
```

Although a closure is very powerful as is, Groovy enriched it to allow "currying" [3]. With this feature, arguments can be bound to closure calls, to prevent redundant code when repeated calls are made. Because this is quite an abstract concept, an example will clarify it's goal and use.

This closure let's a person print a message to the console.

```
def saySomething = {name, message ->
    println "${name} says: '$message'"
}

saySomething "Simon", "Hello!"
//Simon says: 'Hello!'
```

Because a certain Simon wants to say multiple things, and doesn't want to repeat his name every time he does that, the closure is curried.

```
def simonSays = saySomething.curry("Simon")
simonSays "How are you?"
simonSays "Well, goodbye!"
//Simon says: 'How are you?'
//Simon says: 'Well, goodbye!'
```

Excess arguments

When a closure requires a variable amount of arguments, some sort of list would have to be used to be able to achieve this. Groovy enables the use of an argument of type Object[], which is a list that can literally contain every type of object imaginable. If this list is added as the last argument, Groovy allows the contents of the list to be added to the functioncall as if they were plain arguments and part of the closures signature.

```
def replace = {
          format, Object[] args ->
          args.eachWithIndex{obj, i ->
              format = format.replaceAll("(\\{${i}\\})", obj)
          }
          format
      }

println replace ("SELECT {0} FROM {1} WHERE {2} = '{3}' ORDER BY {0} ASC", "username", "Users",
"lastname", "Laforge");
//SELECT username FROM Users WHERE lastname = 'Laforge' ORDER BY username ASC
```

The closure in the example above accepts two arguments; the format of the String that has to be returned and the words that have to be replaced in the desired places. After interpreting the first argument, Groovy detects that the excess arguments are part of the list that is defined as the second and last argument. This list is then iterated over in the closures body, replacing the numbered placeholders with the desired text.

2.1.17 Loops

Because of Groovy's origin in Java, it natively supports both the for- and while-loop. Groovy's for-each loops though, have a slight difference in syntax when compared with there Java equivalents [4] .

A Java for-each loop looks like this:

```
for (variable : iterable) { body }
```

Groovy's counterpart however, uses the keyword 'in' instead of Java's colon.

```
for (variable in iterable) { body }
```

As can be seen in chapter 2.1.9, Groovy has some special capabilities for collections, among which the range. Groovy provides some extra looping techniques that can be used in collaboration with these.

The each-method can be used on collections, and executes a given closure for each of the items in the list [8].

```
('Z'..'A').each{print it}
//ZYXWVUTSRQPONMLKJIHGFEDCBA
```

The `eachWithIndex` does essentially the same, but keeps an numbered index that can be accessed from within the closure.

```
('A'..'Z').eachWithIndex{obj, it -> println "${it+1}. ${obj}"}
/**
 * 1. A
 * 2. B
 * …
 * 25. Y
 * 26. Z
 */
```

2.1.18 Exception handling

Groovy lets the programmer decide to catch the exception or not.

In the following example, the developer tries to open and read the contents of a file. He does not need to surround the method with a `try` and `catch` block, when he knows that the file exists. Groovy gives the programmer control over the exceptions, so he choose to throw one but he does not need to. Java would not compile the code because it expects an `try` and `catch` block that throws the exception `FileNotFoundException`.

```
// checked and unchecked exceptions
def readerWithoutException = new FileReader("/home/wepkey.txt" )println 'Wepkey is: ' +
readerWithoutException.getText()
/**
      result: Wepkey is: D97951A2AEF7A4F927652F573A
**/
try{
      readerInException = new FileReader("/home/hax.txt" )
      println 'Wepkey is: ' + readerInException.getText()
}
catch(FileNotFoundException e){
  println 'Could not find the file'
}
/**
      result: Could not find the file
**/
```

The example above shows two ways of opening a file: With and without a `try` and `catch` block. When the developers knows that a file exists – As shown in the first example above – he doesn't need to surround it with a `try` and `catch`. Like stated earlier, the developer can choose whether he wants to catch the exception or not – This happens in the second example above.

Sometimes, code needs to catch more than one exception. Groovy allows the developer to catch all exceptions in one `catch`, instead of writing a `catch` statement for each `try`.

```
try
{
      // openFile does not exist
      openFile("document.txt" )

      // URL is formatted wrong
      URL url = new URL("http:/www.google.com/");
      InputStream stream = url.openStream();
}
catch(ex)
{
      // Catch all kinds of exeptions from the try
      println "ERROR: " + ex
}
// openFile throws: groovy.lang.MissingMethodException
// URL throws: java.lang.IllegalArgumentException
```

Note that the code above only throws one exception: The function `openFile` does not exist so it directly throws an exception. If it existed, another exception would be thrown since the URL is formatted wrong. The `catch` would show the exception: This is the same `catch` as the one from `openFile`.

Method Missing

An exception with the type “MissingMethodException” was thrown in the previous example. As the name suggest, it means that Groovy can't find a method with that name. Since Groovy is a dynamic language, the developer does not get the error while writing the code, but only when running it.

In Groovy you can implement the `methodMissing()` method. This method allows the developer to dynamically define what to do when `MissingMethodException` is thrown.

The `missingMethod` method accepts 2 arguments. The first is the name of the called method (String). The second argument is an Object[], so the programmer can pass as many variables as he want.

2.1.19 Interfaces

In Groovy you can implement an interface just like in Java. However, this functionality comes with a problem: When the developer tries to use a Groovy class inside an interface.

The reason for this is that Java uses the `javac` to translate Java code to byte code. So when `javac` compiles the interface it will search for the classfiles of the class that is defined in the interface, in this case a class written in Groovy. If the `javac` can't find the class it will search for the source code and try to compile it to byte code.

In this case it can not find the source file of the class because it will look for the <classname>.java file. But the class is written in Groovy so it ends in .groovy. Java will throw an exception after this.

The solution is to compile the Groovy code first and then let `javac` do his work. Javac will succeed this time, because it can find the class files which it could not before.

Frame

When adding an eventhandler to a frame element, the developer can create an anonymous inner class. In the Java example below, the code implements ActionListener and all the methods that are defined in this interface. Programmers do not always need these methods.

Groovy gives them the freedom to only call the functions that they are going to use. Define an anonymous closure with what Groovy has to do when the event fires and add the `as <listener>` to the call.

```
// Java
button.addActionListener(new ActionListener()
{
      public void actionPerformed(ActionEvent ae)
      {
            JOptionPane.showMessageDialog(frame, "You clicked!" );
      }
});

// Groovy
button.addActionListener(
      { JOptionPane.showMessageDialog(frame, "You clicked!" ) } as ActionListener
)
```

2.1.20 Annotations

It is possible to use annotations in Groovy like in Java (Java 5+). There is a limitation in Groovy: It does not allow to create custom annotations [8]: This is not implemented in Groovy (yet). Since Groovy works well with Java, the developers can still create their own annotations in Java and use them in Groovy.

2.1.21 Testing

Testing in Groovy is, like testing in Java, very extensive. A common way to test a application is with JUnit testing. The name for JUnit in Groovy is GUnit. Codehaus and IBM have written articles about unit testing in Groovy. Documentation on their findings can be found on the following websites:

- Codehaus's testing guide: http://groovy.codehaus.org/Testing+Guide
- IBM's findings: http://www.ibm.com/developerworks/java/library/j-pg11094/

Because the subject GUnit doesn't support our main question, this will not be fully described. Basically, it is the same as JUnit testing.

2.1.22 Working with databases

In addition to its ability to execute SQL in a variety of ways, Groovy also provides the user with a wrapper around the standard Java classes, adding more functionality and ease of use when working with databases. It is important to understand that Groovy currently doesn't replace all of Java's original databases connectivity, as the JDBC is still in use.

The first few examples will show a couple of ways to use plain SQL embedded in Groovy. After that, a few examples of how datasets work in Groovy are shown. The first example will start with JDBC, since this a requirement to be able to do anything with a database.

```
//This import allows for SQL objects, that can be used to talk straight to the database
import groovy.sql.Sql
import java.util.logging.*


      /*
       * There two ways to connect to databases using Groovy. This method is preferred,
       * for it allows Groovy to automate more of the usual SQL work,
       * and allows for reuse of connections.'''
       */
      source = new org.hsqldb.jdbc.jdbcDataSource()
      source.database = 'jdbc:hsqldb:mem:test'
      source.user     = 'sa'
      source.password = ''
      db = new groovy.sql.Sql(source)
```

The code above does not contain a lot of Groovy specific code, and should be fairly easy to understand. The HSQLDB is used in memory with a database called “test”. If desired, any database with a JDBC connector could be used instead.

Once we have a JDBC connection, we are able to parse SQL very easy. There are easier ways to manipulate the data, as shown further below.

```
/*
 * Groovy is capable to directly talk SQL to the database
 * using the Sql object that Groovy provides.
 */
db.execute     ''' DROP INDEX classIdx IF EXISTS;
                               DROP TABLE Class IF EXISTS;
                               CREATE TABLE Class(
                                              classId INTEGER GENERATED BY DEFAULT AS
IDENTITY,
                                              classCode VARCHAR(5),
                                              teacher VARCHAR(64)
                               );
                               CREATE INDEX classIdx ON class (classId);'''
```

The code above is the way of using plain SQL. This form of programming is not very reusable. To achieve re-usability, a so called prepared statement can be used. A prepared statement is a string with question marks where values should be inserted. By using the execute function of a Groovy SQL object with the prepared statement and the values as parameters, the statement can be used.

```
/**
* A more interesting way (and usually faster as well) is to create a String with question marks
* on places where the values should go. This is called a "prepared statement"
* When db.execute is called with the prepared statement as argument, and a argument
* containing the values that sould go on the question marks, it will work as well.
*/
    def classInsert = '''INSERT INTO Class (classCode, teacher)
                VALUES (?, ?); '''

      db.execute classInsert, ['382', 'Einstein']
```

Using a simple prepared statement can be a little tricky, since the user has no clue on what information goes where. In such a situation, switching first and last name becomes a simple mistake. Using a GString can solve this problem, as in the example shown below. Here a Gstring is used to create something similar to a table.

The example also shows how to use a closure with a `db.execute` call that has a GString as its parameter. If there are difficulties understanding how closures or GStrings work in Groovy, please refer to the earlier parts of this document (2.1.6 Groovy String).

```
/*
 * The Groovy implementation of the java String is called a GString, and can be used combined
 * with another GString and a call to the Groovy SQL object to insert data into the database.
 */
      def classes = [
                     [classCode: '106', teacher: 'Plato'],
                     [classCode: '204', teacher: 'Cleopatra'],
                     [classCode: '304', teacher: 'Bush']
                     ]

      classes.each { currentClass ->
                        db.execute """INSERT INTO Class (classCode, teacher)
                        VALUES (${currentClass .classCode}, ${currentClass.teacher})"""
      }
```

All of the examples above use just one table in their data operations, but since databases usually contain a lot more tables, this isn't very realistic. The technique above can also be used in a slightly different way that involves more tables. This is shown below.

```
def students = [
                     [first: 'James', last: 'Morrison', birth: '1987-04-21', payed: TRUE',
classCode: '106'],
                     [first: 'Britney', last: 'Stears', birth: '1989-09-02', payed: 'TRUE',
classCode: '106'],
                     [first: 'Robbie', last: 'Williamed ', birth: '1981-03-24', payed: 'TRUE',
classCode: '204'],
                     [first: 'Chist', last: 'Martin', birth: '1986-09-03', payed: 'TRUE', classCode:
'204'],
                     [first: 'Nick', last: 'Cavein', birth: '1984-08-08', payed: 'TRUE', classCode:
'304'],
                     [first: 'Pete', last: 'Dorethy', birth: '1986-06-06', payed: 'FALSE',
classCode: '304'],
                     [first: 'Ben', last: 'van Halen', birth: '1965-11-09', payed: 'TRUE',
classCode: '304']
                          ]

/*
 * The GString can also contain a SELECT query itself.
 */
students.each { student ->
   db.execute """INSERT INTO Student (firstname, lastname, dateOfBirth, payed, class) SELECT $
{student.first}, ${student.last}, ${student.birth}, ${student.payed}, classId FROM Class WHERE
classCode=${student.classCode};"""
}
```

In all of the examples above, data is added to the database, but none is read from the database. The code belows shows how this can be done, using the `eachRow` call to the Groovy SQL object, combined with a bit of SQL. A closure is run for every item that is return by the call, printing the full name of each student.

```
/*
* With the eachRow fuction of an Groovy SQL object, a list of records is returned.
* In this example, the list is used into a closure, that prints data from the list.
*/
      db.eachRow('SELECT firstname, lastname FROM Student')
      { row -> println row[0] + ' ' + row[1]}
```

It is not always desired to directly process each row that is returned like in the example above. In this case, the `rows()` function call returns a list. This then can be used just like any other list.

```
/*
 * It is also posible to get a regular list using the rows function of the Groovy SQL object.
 */
      List studentList = db.rows('SELECT firstname, lastname FROM Student')
      println "There are ${studentList.size()} students."
```

All of the example above use one of the forms of SQL processing that Groovy supports, but Groovy also provides another way to access the database. This method, using so called datasets, does have its limits: It does not allow any update or delete operations, nor does it allow the creation of database schema's [4]. Its syntax is usually much shorter than most SQL statements, and there is no SQL required.

The following examples will show similar or extended functionality to those above, except they will all use the Groovy dataset class.

The first example will show how a dataset on a single table is created, and how a single record is added to this table. Take note that `Class` is the name of the table to be accessed.

```
      /*
       * One of Groovy's most powerful database function is the dataset.
       * It can be used for a multitude of functions, one of them added new records.
       */
      classSet = db.dataSet(Class)

      classSet.add(
                      classcode: '606',
                      teacher: 'Genghis')
```

Using a dataset to read data from a database is not more complicated then adding data to the database. In the next example a closure is ran for each record returned. Note how fields from the table can be accessed like they where actual variables of a class.

```
/*
 * Above, a dataset is used to insert data into the database. Here, we use it to print
 * data for a table.
 */
      studentSet = db.dataSet('Student')

      studentSet.each {
      println it.firstname + " " + it.lastname + " payed: " + it.payed.toString()}
```

It is also possible to access data from multiple tables at once using a view as the source for the dataset, instead of a single table. To create the view, the use of SQL is needed. Also note that a dataset made from a view is read-only. To alter the table, the use of normal SQL is needed.

```
/*
 * It is also possible to create Datasets from views. A databased based upon a view creates a
read only view
 * across multiple tables. with the findall function, this data can be further trimmed down to
only contain
 * the needed data.
 */
      db.execute       '''DROP VIEW StudentTeacher IF EXISTS;
                              CREATE VIEW StudentTeacher AS
                              SELECT * FROM Student INNER JOIN Class
                              ON class=classId;'''

      record = db.dataSet('StudentTeacher').findAll{ it.classCode=='304'}
      record.each{ println it.firstname + ' ' + it.lastname + ': ' + it.classCode }
```

Summary

Groovy has a few ways to talk to a database (*Figure 5: Groovy and databases*), some simpler then others. We believe, that all of the them are fairly easy compared to the standard code Java provides. Groovy however, not only eases the use of databases, but also adds functionality to the standard Java code, using the Groovy specific features such as closures and the GString.

Further more, there is no single best technique to access the database. In a CRUD application, the Groovy dataset will be almost enough for the build the entire application, and hereby decrease the use of plain SQL in the code. Other database operations can still be done by using (prepared) SQL statement.

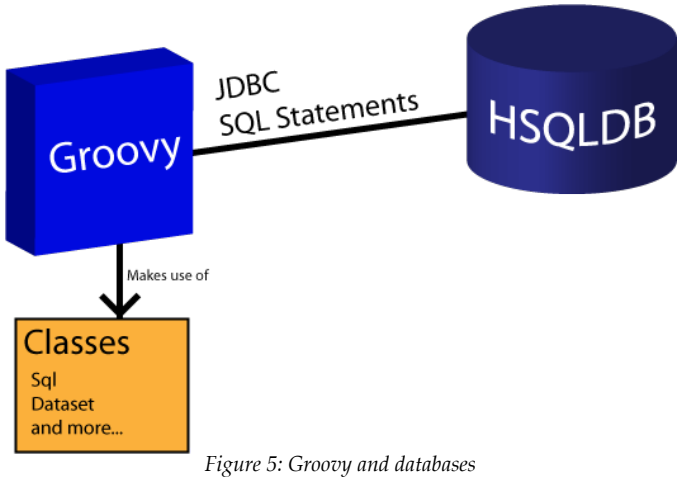

Figure 5: Groovy and databases

As such, we believe Groovy is a better alternative for Java in case of lot of database connectivity is required.

2.2 Meta Object Protocol

The Meta-Object Protocol (Figure 6: Working with the MOP) enables Groovy to dynamically change the behavior of classes and objects at runtime [4]. A MetaClass is provided for each class, whether it's POJO or a POGO. This MetaClass contains information like the available methods and properties.

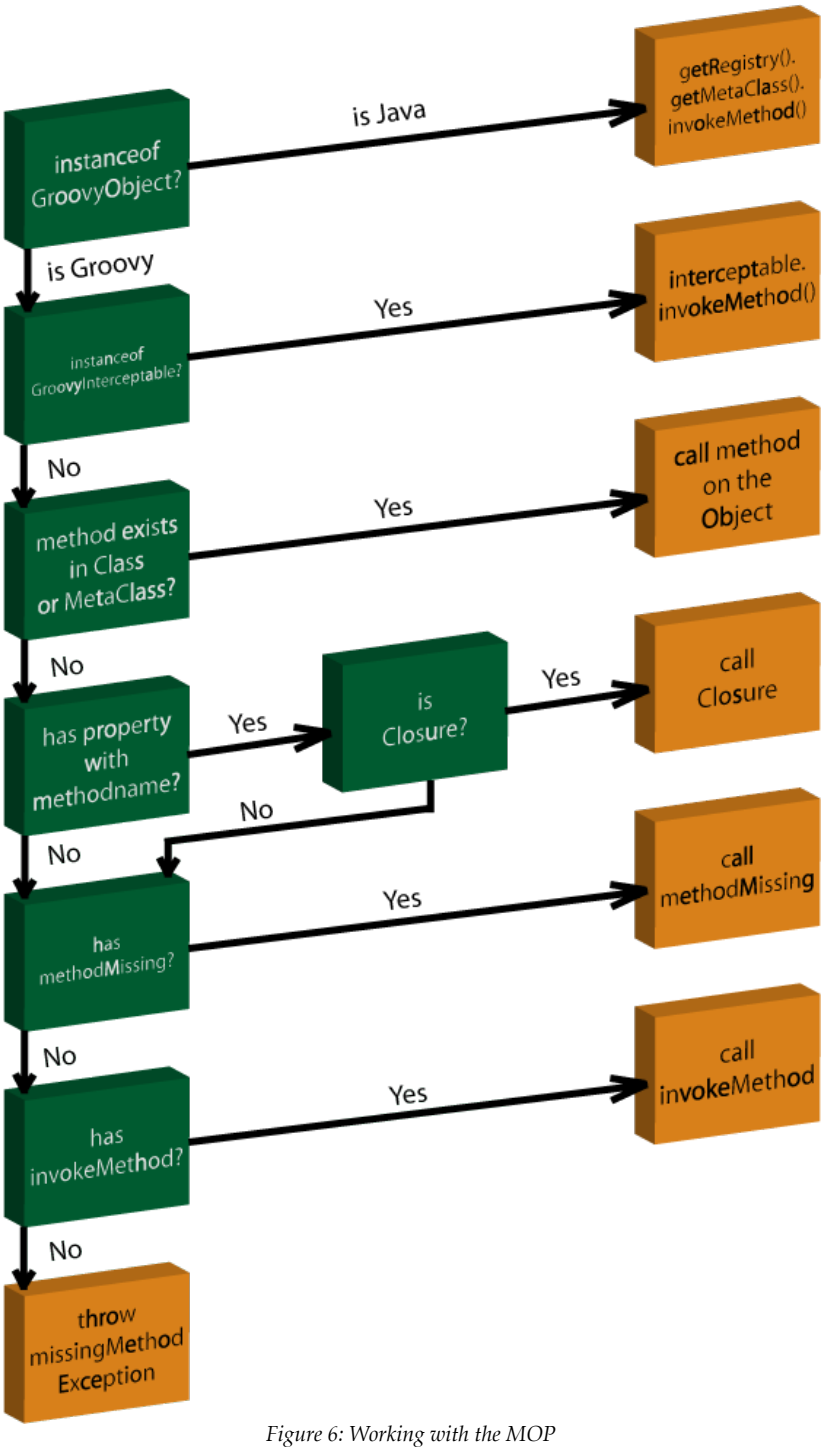

Figure 6: Working with the MOP

The MetaClass comes in action when a property of an object is being referenced or a method is being invoked. Instead of letting the objects handle the requests themselves, they are sometimes being routed to the right MetaClass. This process is influenced by a few factors. In the case of a method invocation, these are the following;

First of all, the type of the requesting object is inspected. If it is a POJO, the MetaClass is retrieved from the so called MetaClassRegistry, which aggregrates all MetaClasses that are available. Subsequently, the MetaClass' invokeMethod is called, with the name of the method and the arguments passed to it.

On the other hand, when the object is a POGO, a few other considerations are made. First, it is determined whether the object implements the GroovyInterceptable interface, which will be explained later on. If that is the case, the interceptables invokeMethod is called.

Otherwise, it is checked whether the requested method exists in the objects Class or MetaClass. If so, the method is called directly. If not, Groovy inspects the object and checks for properties with the name of the requested method. If one exists, it is determined whether this property is a Closure and thus executable. If so, Groovy calls the closure.

In all other cases, Groovy checks whether the object has an implementation of the methodMissing method. If so, it calls this, enabling the developer to intercept calls to inexistent methods. Finally, Groovy checks whether the objects itself has an implementation of invokeMethod, and if so, it calls this. If not, it throws an MissingMethodException.

2.2.1 Interceptable

As mentioned before, the GroovyInterceptable interface can be used to add supplementary logic to method calls, which can be useful for logging, security or exception handling.

```
import org.codehaus.groovy.runtime.InvokerHelper

public static class CustomLogger
{
	static void Log(String message)
	{
		println "Method called: " + message;
	}
}

public class Interceptable implements GroovyInterceptable
{
	Object invokeMethod(String name, Object args)
	{
		CustomLogger.Log(name);
		def metaClass = InvokerHelper.getMetaClass(this)
		def result = metaClass.invokeMethod(this, name, args)

		return result
	}

	def greet()
	{
		return 'Hello World!'
	}
}

def ic = new Interceptable();
println ic.greet();

//Method called: greet
//Hello World!
```

The above example illustrates how dynamic method invocation can be used to intercept method invocations. To achieve this, the class must implement the GroovyInterceptable interface. This allows for overriding the `invokeMethod` method, so the method call can be logged. However, because the default behavior is overridden, the actual method call must be rebuild. This is done by obtaining the MetaClass that is related to the Groovy class interceptable, and calling `invokeMethod` on it.

2.2.2 Categories

Furthermore the MOP provides Groovy with so called categories. A category is used when there is need to temporarily add functionality to a class that doesn't allow this.

The class Integer for example, is declared final and can thus not be extended. Say that for a given integer-value, you want to check whether it's a valid dutch bank account number. This can be done with a reasonably simple calculation which doesn't require further explanation to illustrate the workings of categories.

```
class PaymentValidation
{
	def static Boolean validateBankaccount(Integer bankaccount)
	{
		if(bankaccount.toString().length() == 9)
		{
			def i = 0
			def totaal = 0
			for(c in bankaccount.toString())
			{
				totaal += ((9-i) * c.toString().toInteger())
				i++
			}
			return (totaal % 11 == 0)
		}
		return false
	}
}

use(PaymentValidation)
{
	println 123456789.validateBankaccount()
	//prints true
	println 829531673.validateBankaccount()
	//prints false
}

println 123456789.validateBankaccount()
//throws MissingMethodException
```

The category-method has to be declared static in order for it to be accessible without an instance of the containing class. A call to the category has to be nested inside a use-block. This delimits the scope of the category as shown in the example.

2.2.3 Expando

Another useful aspect of the MOP as implemented by Groovy, is the Expando class. Other then the methods and properties it inherits from extending `groovy.lang.GroovyObject`, it is empty. At runtime, properties and methods can be added as required.

```
def ex = new Expando()
ex.length = 100
ex.width = 200

ex.getSurfaceArea = {
    return ex.length * ex.width
}
println ex.getSurfaceArea()
```

Internally, an Expando exists of a Map wherein al properties and methods are stored as key-value pairs, thus allowing the Expando to resize as needed.

3

Webservices in Groovy

" Trust me - we [Web Services] will move way beyond the suckline - there is too much money on the table. "
- Jeff Schneider

3.1 Creating, manipulating, validating and parsing XML

Due to Groovy's dynamic nature, working with XML is very easy [8]. This is demonstrated in the following examples, where a XML file containing contact data is used. For a given contact, the following information is included;

- Name;
- Birthday;
- Address, containing both city and street;
- One or more phonenumbers;
- One or more e-mail addresses;
- Occupation;

When represented in XML, this looks like this;

```
<contacts>
    <contact name="John Malmon" birthday="07-01-1960">
        <address>
            <street>Broadway</street>
            <city>New York</city>
        </address>
        <phonenumbers>
            <phonenumber>0191740284</phonenumber>
        </phonenumbers>
        <emailaddresses>
            <mailaddress>john.malmon@gmail.com</mailaddress>
            <mailaddress>j.malmon@microsoft.com</mailaddress>
        </emailaddresses>
        <job type="Microsoft Programmer" />
    </contact>
</contacts>
```

Due to Groovy's dynamic nature, processing an XML-document is quite easy. At runtime, classes and properties are created that map to the corresponding nodes in the XML-tree.

Groovy provides three classes for processing XML files. The XmlParser and XmlSlurper-classes have a similar API, but the XmlSlurper has less overhead, whereas the XmlParser has the ability to manipulate the XML file. The DOMCategory-class provides similar functionality, but also enables more low-level operations.

When using the XmlParser to print the contact-information for all users in the addressbook, the code will look something like this;

```
import nl.han.grandeur.xmlprocessing.documents.*;

def records = new XmlParser().parseText(XMLdocument.CONTACTS)
records.contact.each{
      println 'Name: \t\t\t' + it.'@name'
      println 'Birthday: \t\t' + it.'@birthday'
      println 'Street: \t\t' + it.address.street.text()
      println 'City: \t\t\t' + it.address.city.text()
      println 'Phonenumbers: \t\t' + it.phonenumbers.phonenumber
      println 'E-mailaddresses:\t' + it.emailaddresses.mailaddress
      println 'Job: \t\t\t' + it.job.'@type'.text()
}
```

- `it.'@name'` - Prints the node attribute (In this case "`name`")
- `.text()` - Prints the node value
- `phonenumbers.phonenumber` - Prints all "`phonenumber`" nodes found in the "`phonenumbers`" node.

First, the string representation of an XML-document is passed in to a new instance of the XmlParser-class. Then the information for the contacts is retrieved and printed to the console.

Constructing an XML-document like the one previously shown is, as shown below, really easy with Groovy's MarkUpBuilder and StreamingMarkUpBuilder.

```
def writer = new StringWriter()
def xml = new MarkupBuilder(writer)
xml.contacts()
{
      contact(name: 'John Malmon', birthday: '07-01-1960')
      {
               address()
               {
                        street('Broadway')
                        city('New York')
               }
               phonenumbers()
               {
                        phonenumber('0191740284')
               }
               emailaddresses()
               {
                        mailaddress('john.malmon@gmail.com')
                        mailaddress('j.malmon@microsoft.com')
               }
               job(type: 'Microsoft Programmer')
      }
}

println writer.toString()
```

This will print the exact same XML-document as the one shown before.

To manipulate an XML-document, both the XmlParser and the DOMCategory-class can be used. In this example, the DOMCategory class is used to change Johm Malmon's address and phonenumber.

```
import groovy.xml.DOMBuilder
import groovy.xml.dom.DOMCategory
import nl.han.grandeur.xmlprocessing.documents.*;

def reader  = new StringReader(XMLdocument.CONTACTS)
def doc     = DOMBuilder.parse(reader)
def records = doc.documentElement

use (DOMCategory)
{
    def john = records.contact.find{ it.'@name' == 'John Malmon' }
    john.address.street.each { street -> street.value = '40th St' }
    john.address.city.each { city -> city.value = 'Redmond, Washington' }

    john.phonenumbers.each { it.appendNode('phonenumber', "0191836953") }
}
```

Finally, Groovy provides the ability to validate XML to a certain schema [8]. The schema that is used can be found in appendix 9.1.

```
import nl.han.grandeur.xmlprocessing.documents.*
import javax.xml.XMLConstants
import javax.xml.transform.stream.StreamSource
import javax.xml.validation.SchemaFactory

def factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI)
def schema = factory.newSchema(new StreamSource(new StringReader(XMLschema.SCHEMA)))
def validator = schema.newValidator()

try
{
      validator.validate(new StreamSource(new StringReader(XMLdocument.CONTACTS)))
      println 'The document complies to the provided XML-schema'
}
catch(Exception ex)
{
      println 'The provided document does not comply to the provided XML-schema'
}
```

3.2 SOAP-based webservices

3.2.1 GroovySOAP

Groovy provides a module called Groovy SOAP, that enables developers to both create SOAP-based webservices and clients. Groovy SOAP is based on XFire [8].

XFire, an open source Java SOAP framework created by The Codehaus, has been created for development and consumption of web services. XFire makes service oriented development approachable through its easy to use API and supports standards. The framework uses the StAX (Streaming API for XML) for processing XML documents [26].

Features and Goals

> "
> - Support for important Web Service standards - SOAP, WSDL, WS-I Basic Profile, WS-Addressing, WS-Security, etc.
> - High performance SOAP Stack.
> - Pluggable bindings POJOs, XMLBeans, JAXB 1.1, JAXB 2.0, and Castor support.
> - JSR 181 API to configure services via Java 5 and 1.4 (Commons attributes JSR 181 syntax).
> - Support for many different transports - HTTP, JMS, XMPP, In-JVM, etc.
> - Embeddable and Intuitive API.
> - Spring, Pico, Plexus, and Loom support.
> - JBI Support.
> - Client and server stub generation.
> - JAX-WS early access support.
>
> [Source: http://xfire.codehaus.org/] "

> "
> - Create a flexible SOAP framework, where any processing mechanismcan be plugged in.
> - Be SOAP 1.2 and WS-I 1.1 compliant. Also to offer support for non-RPC/Encoded SOAP 1.1 services.
> - Intuitive, easy to use API.
> - Be Fast.
> - Allow many different binding methods (traditional java types, OGNL, Castor, JaxME, etc).
> - Create a processing model where your web service model and your java model can develop independently (see the Aegis Module).
> - Modules for WS-Security and WS-Adressing support.
>
> [Source: http://xfire.codehaus.org/FAQ] "

An important note about XFire

As stated on the XFire website [26]: **XFire is now Apache CXF**. When you're planning to work on a new project, you should use CXF. CXF is considered to be XFire 2.0, as CXF has many new features, bug-fixes and is JAX-WS compliant. XFire will be maintained through bug fixes, but no new features will be added, since most development will occur on Apache CXF.

To illustrate how Groovy SOAP can be used to create a SOAP-based webservice, the bank account-validation example from the previous chapter will be used. An endpoint with a single 'validateBankaccount'-method is created, that accepts an Integer and returns a Boolean to indicate whether the supplied value is a valid dutch bank account number.

```
import groovy.net.soap.SoapServer

public class PaymentValidation
{
      def boolean validateBankaccount(Integer bankaccount)
      {
               if(bankaccount.toString().length() == 9)
               {
                        def i = 0
                        def totaal = 0
                        for(c in bankaccount.toString())
                        {
                                 totaal += ((9-i) * c.toString().toInteger())
                                 i++
                        }
                        return (totaal % 11 == 0)
               }
               return false
      }
}

def server = new SoapServer("localhost", 7003)
server.setNode("PaymentValidation")
server.start()
```

The above code is quite simple, a PaymentValidation-class is declared, then a new instance of the SoapServer-class is instantiated, the PaymentValidation-class is set as the endpoint and the server is started.

XFire is responsible for everything that happens under the hood. It creates the WSDL and routes requests to the PaymentValidation-class.

XFire acknowledges that the types of both the inputparameter and the returnvalue are explicitly declared, and therefore maps these to corresponding XSD-types in the WSDL.

```
<xsd:element name="validateBankaccount">
      <xsd:complexType>
               <xsd:sequence>
                        <xsd:element maxOccurs="1" minOccurs="1" name="in0" nillable="true"
type="xsd:int"/>
               </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<xsd:element name="validateBankaccountResponse">
      <xsd:complexType>
               <xsd:sequence>
                        <xsd:element maxOccurs="1" minOccurs="1" name="out" nillable="true"
type="xsd:boolean"/>
               </xsd:sequence>
      </xsd:complexType>
</xsd:element>
```

If the Groovy `def` keyword would have been used, XFire would not be able to determine the type of the variable, and both the inputparameter and the returnvalue would have been translated into 'xsd:anyType'.

To consume this webservice in Groovy, a simple consumer can be made using the SoapClient-class like this;

```
import groovy.net.soap.SoapClient
def proxy = new SoapClient("http://localhost:7003/PaymentValidationInterface?wsdl")
println proxy.validateBankaccount(123456789)
//prints 'true'
println proxy.validateBankaccount(123456788)
//prints 'false'
```

Due to Groovy's dynamic nature, there is no need to generate stubs. This enables developers to consume a webservice in as many as two lines of code. Furthermore, it doesn't require any extensive configuration in XML-files.

The abstraction of technical details has it's downsides though. When the endpoint in the previous example is examined, no package-declaration can be found. If a package declaration would have been included, the reference to the endpoint in the SoapServer-instance would have to be changed to something like this in order for the PaymentValidation-class to be found;

```
server.setNode("nl.han.grandeur.PaymentValidation")
```

However, XFire creates an interface with the exact same name that points to the endpoint-class. In this case, this will result in an interface-declaration like this;

```
public interface nl.han.grandeur.PaymentValidation...
```

Because no periods are allowed in class- and interface names, this will lead to an exception.

Furthermore, Groovy SOAP can not handle Groovy's metaClass when mapping input parameters and return values to xsd-types. Therefore, the metaClass has to be explicitly excluded in this process. This creates a situation where, Expando's for example, cannot be used as the return value of an operation.

Another limitation concerns the client-side aspect of web services. Groovy SOAP does not support complextypes in requests; only datatypes that can be translated into the existing xsd-types like '`xsd:String`' and '`xsd:Integer`' can be used [33].

However, as stated above, the XFire project itself is no longer supported and is continued as Apache CXF. Groovy has adapted to this change with a new module, called GroovyWS which will be discussed in the next chapter.

3.2.2 GroovyWS

GroovyWS provides a module to create webservices as well as consume them. This module replaces the old Groovy SOAP [8]. GroovyWS requires Java 5, because it uses Apache CXF and CXF only works on Java 5+ platforms.

> “ Apache CXF is an open source services framework. CXF helps you build and develop services using frontend programming APIs, like JAX-WS. These services can speak a variety of protocols such as SOAP, XML/HTTP, RESTful HTTP, or CORBA and work over a variety of transports such as HTTP, JMS or JBI.
>
> [Source: http://cxf.apache.org/] ”

Furthermore, it is important to know that GroovyWS does not support contract first development of webservices yet [33].

Setting up a webservice

In this example, a client is written that sends an username to a webservice. The webservice will reverse the name and send it back.

First we will create a new file called Server.goovy. This will be our server.

```
import groovyx.net.ws.WSServer

server = new WSServer()
server.setNode("grandeur.flipservice.ServiceEndpoint", "http://localhost:9001/Flip")
server.start()
```

After creating an instance of WSServer, the setNode method is used. This method accepts two parameters. The first parameter is class used as endpoint. The second argument tells GroovyWS where the newly created service should be published.

The next step would be to create the ServiceEndpoint. This can be done either in the same file, where the server code is located (Server.groovy), or in a newly created file called ServiceEndpoint (show below).

Note how both the parameter and the returntype show below are statically typed. If Groovy specific dynamic typing would have been used, GroovyWS would not have been able to properly the required types, and would have responded with a xsd:any element.

The following code does the flip:

```
class ServiceEndpoint{
      def String flip(String username){
              def flippedUsername = username.reverse()
              return flippedUsername
      }
}
```

Running both of the code snippets above will create a webservice at the specified location, that can be accessed using the URL to the specified location when the "?wsdl" suffix is added. If desired, the service can be tested by using a tool like soapUI.

In the example above, the method only accepts one parameter of the type String. It is also possible to accept an POGO that was previously defined. If an object is to be accepted or returned, an additional XML configuration file is needed to make sure GroovyWS ignores the metadata class. An example of such a file is show below:

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns:sample="http://DefaultNamespace">
  <mapping name="sample:classname">
    <property name="metaClass" ignore="true"/>
  </mapping>
</mappings>
```

The above should be edited as needed and placed in an XML file called <classname>.aegis.xml [8].

To demonstrate how an RESTful service can be consumed using GroovyWS, the following example calls upon the Last.fm service to provide the last played track for the username send to the service.

To consume a webservice, the ServiceEndpoint needs to be modified. The following example shows the modified ServiceEndPoint, where an external webservice called upon (Last.fm in this case) with the given username. The response of the service now contains both the reversed name and the last played song in Last.fm.

```
def Profile flip(String username){
              def flippedUsername = username.reverse()

              URL url = new URL ("http://ws.audioscrobbler.com/2.0/?
method=user.getrecenttracks&user=${username}&api_key=42c902a953202e9b20d5b60af2fb77ea")
              InputStream inputStream = url.openStream()
              def lfm = new XmlParser().parse(inputStream)
              def lastTrack =  lfm.recenttracks[0].track[0].name[0].text()

              Profile profile = new Profile()
              profile.flippedUsername = flippedUsername
              profile.lastTrack = lastTrack

              return profile
      }
```

As seen above, consuming a webserver is as simple as providing an URL, calling upon the service and parsing the result to XML.

In this example, the only part of Last.fm's response used is the last played track. Using the XmlParser, only the last track is read from the response and stored in a object called profile. This definition of this object is show below:

```
class Profile{
      String flippedUsername
      String lastTrack
}
```

Note that before it is possible to use the Profile class as response for a service, an aegis XML file is needed. The content of such a file can be found in earlier examples.

3.3 RESTful webservices

This part is about making simple RESTful requests and transforming the resulting XML to Groovy objects.
For the following example, the following assumptions are made:

- There is a RESTful webservice available.
- The RESTful webservice returns XML.
- XML can be parsed to Groovy objects

Looking for a webservice

In this example, the Last.fm webservice is used. It has a clear API and has some interesting functions. Last.fm is also easy to use and has the potential to grow. This example uses the "User Get Recent Tracks" method. This returns information about a user, therefore, a valid last.fm username is required, as well as an api key. Both are easily obtainable from the last.fm website.

Checking the XML

Last.fm uses an request response method. The api provides an example of how the returned XML will look like. It is recommended to examine this before trying to use the XML in an application.

According to the api, the url should look something like this:

```
http://ws.audioscrobbler.com/2.0/?method=user.getrecenttracks&user=<username>&api_key=<apikey>
```

If desired, most modern webbrowsers can display the XML file using the URL above.

GroovyRestlet

The RESTlet framework has proven to be a nice framework for creating RESTful webservices. It is widely used in Java. When using RESTlet, one can create a RESTful webservices to their likings. RESTlet can also be used to develop client-side applications, all with the same API. RESTlet also provides URI as UI support, creating an easy way to point the request made to the right location inside your application.

GroovyRestlets enables the use of the RESTlet framework in Groovy. If one has created RESTlets in Java, this would be a gift from heaven. Why? Because Groovy is considered to be “an agile dynamic language for the Java Platform”. Wouldn't it be convenient to create RESTful services in a dynamic environment?

While GroovyRestlet can provide an RESTful environment in an application, Grails has almost the same support. The main difference between GroovyRestlets and Grails applications is that Grails is a framework with its own conventions. Grails also forces MVC implementations. Some programmers might not like the idea that Grails handles REST as CRUD, making it a file storage / reading service.

In RESTlet programmers can choose their own conventions and can choose not to approach REST as CRUD. However, this also lays a larger burden on the developer. If one likes MVC and approaching REST as CRUD, wouldn't it be smart to choose a framework already supports that?

RESTlet gives the freedom to create an application with RESTful support the way a developer might want it. Grails gives a set of widely used pattern solutions but at the same time enforcing them.

RESTlet is a Java framework and has it's own community, Grails is also a framework. But because Grails is Groovy related and RESTlet isn't only Groovy related, the working in Groovy is not highlighted. As said on the Groovy website; GroovyRestlet creates shortcuts to RESTlet related features.

Using the XML in Groovy

Groovy has a lot of ways to convert XML to understandable code. In this example XmlParser is used. XmlParser accepts file inputstreams, which is needed to load the file.

```
def xml = new XmlParser().parse(file);
```

The code above parses a filestream into a new object called xml.

In order to use XML from a external source, an filestream is needed. This show in the example below:

```
def username = "something";
def apikey = "something";
def address = "http://ws.audioscrobbler.com/2.0/?method=user.getrecenttracks&user=" + username
+ "&api_key=" + apikey;

println "Requesting " + address;

//create a new url, to put into a filestream
def u = new URL(address);
def file = u.openStream();
```

Using the code from the previous snippet allows Groovy to use the external file as an inputstream. All that's left now is to loop through the XML.

```
def xml = new XmlParser().parse(file);
//print the username returned by the XML (@user is the attribute 'user')
def output = xml.recenttracks.'@user'.text() + " listened to the following tracks:\n";

for (def i = 0; i < xml.recenttracks.track.size(); i++) {
      // create groovy objects from the XML
      def artistname = xml.recenttracks.track[i].artist.text();
      def trackname =  xml.recenttracks.track[i].name.text();
      def nowplaying = xml.recenttracks.track[i].'@nowplaying';
      //print the tracks: <artist> - <trackname>
      output += "\n";
      if (nowplaying == "true") {
              output += "(Now playing) ";
      }
      output += artistname + " - " + trackname;
}

println output;
```

3.4 Securing webservices

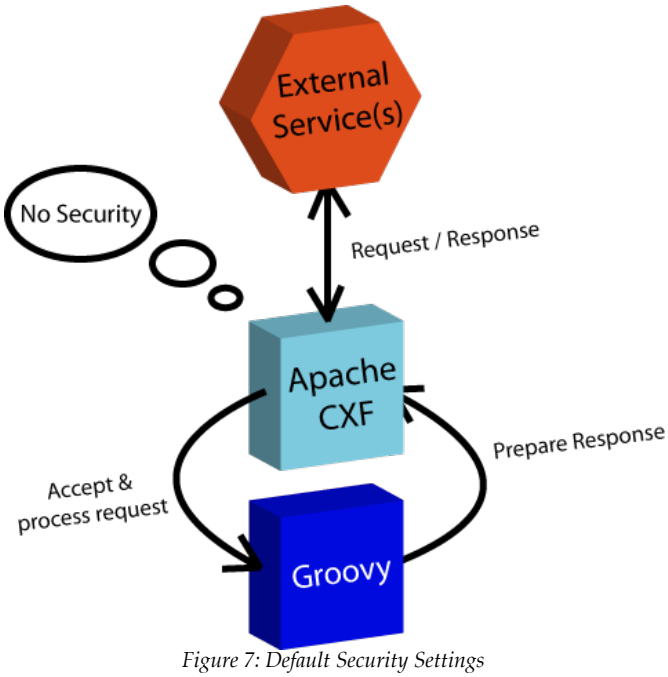

Figure 7: Default Security Settings

With the default settings, the ApacheCXF module that is used by Groovy doesn't implement any forms of security (*Figure 7: Default Security Settings*) [24]. Any kind of client can view the XML (WSDL) files and make use of the server. This is, for testing purposes, very convenient: there is no don't need to hassle with security settings. If the service is placed on a public location (For example: the Internet) and the service is not intended for public use, security is needed.

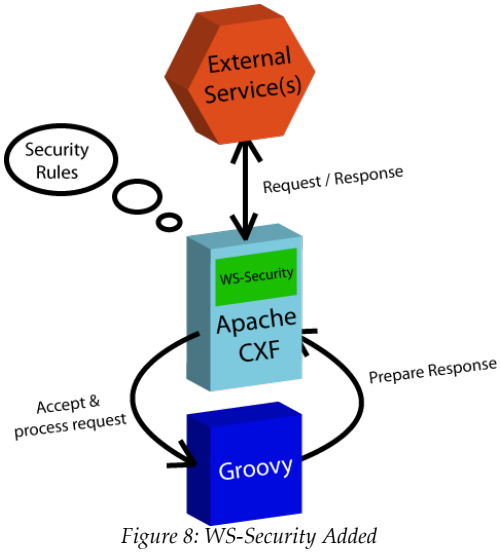

Figure 8: WS-Security Added

To add security, one of the WS-* standards comes in play: WS-Security. This standard enables security above and beyond transport level protocols, as well as allows encryption and signing of messages. ApacheCXF uses WSS4J to implement the WS-Security standards (*Figure 8: WS-Security Added*).

WSS4J has the following features [34] :

- XML Security
  - XML Signature
  - XML Encryption
- Tokens
  - Username Tokens
  - Timestamps
  - SAML Tokens (Security Assertion Markup Language)
- Basic Authentication

By combining a few of the above features, a service can be made as secure as desired.

It is important to know that GroovyWS is flawed considering security. The only form of security that GroovyWS currently supports is Basic Authentication [8]: Asking for a username and password. All the other kinds of security (XML Security and Tokens) are not (yet) implemented by GroovyWS.

If these security feature are needed, it is possible to use WSS4J or Apache CXF instead of GroovyWS, since Groovy is fully compatible with Java. On it's own, Groovy is only suitable to create unsecured or weakly secured webservices and therefore not all that suitable.

4

Grails

" There are only two kinds of languages: the ones people complain about and the ones nobody uses. "

- Bjarne Stroustrup

To achieve human interaction with webservices in a SOA, there needs to be some form of a client. This client could be a Groovy application, but this would mean the user would have to download and install the application. However, when an application is distributed to broad audience, a webapplication is often more convenient.

Clients for a SOA are regularly build as a dashboard that can be accessed through a web browser [10]. The user doesn't need to download and install a application: It can access a website and perform the actions needed. Currently, Groovy doesn't (natively) support web server support. This is were the Grails framework comes in to play.

Grails is an open source web application framework which leverages on Groovy (*Figure 9: Grails is build on top of Groovy*) [9]. It was first developed in 2005 and the first "1.0" release was announced in 2008 [35].

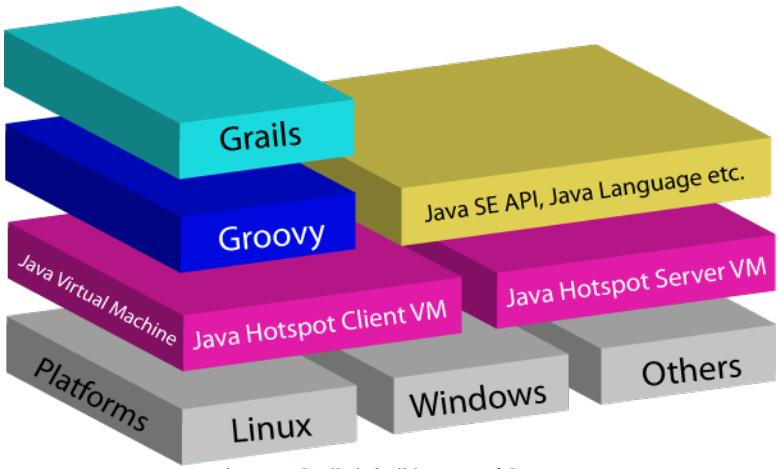

Figure 9: Grails is build on top of Groovy

The framework has been created following the "coding by convention" (also known as "Convention over configuration") and makes use of the Model-View-Controller pattern.

> " Grails has been developed with a number of goals in mind:
>
> - Provide a high-productivity web framework for the Java platform.
> - Re-use proven Java technologies such as Hibernate and Spring under a simple, consistent interface
> - Offer a consistent framework which reduces confusion and is easy to learn.
> - Offer documentation for those parts of the framework which matter for its users.
> - Provide what users expect in areas which are often complex and inconsistent:
>   - Powerful and consistent persistence framework.
>   - Powerful and easy to use view templates using GSP (Groovy Server Pages).
>   - Dynamic tag libraries to easily create web page components.
>   - Good Ajax support which is easy to extend and customize.
> - Provide sample applications which demonstrate the power of the framework.
> - Provide a complete development mode, including web server and automatic reload of resources.
>
> [Source: http://en.wikipedia.org/wiki/Grails_(framework)] "

4.1 Scaffolding

4.1.1 What is Scaffolding?

Scaffolding is a technique that's implemented by some MVC frameworks [36]. It allows the developer to generate the View and Controller, based on the information found in the Model. The code of the View and the Controller is turn-based on a pre-defined template.

The controller will automatically contain the CRUD actions. It allows for faster development, since the developer doesn't need to create the functions and views.

There are two types of scaffolding: *Static and Dynamic*.

Static Scaffolding (or Scaffold generation)

This form of scaffolding will create the actual code for the programmer. The developer needs to create his model and and execute the scaffolding command. The View and Controller will be generated based on the Model and will automatically implement CRUD actions. The way the code is generated is turn-based on a template.

The developer can now edit the Controller and the View to make any necessary changes (if there are any). This form of automatically creating code will speed up the programming process.

Dynamic Scaffolding

Dynamic Scaffolding allows the creation of the Controller and the View at runtime. This form of scaffolding only works in a dynamic language.

Because of this the developer would not be able to edit the Controller and View, since they do not exist yet while programming. Other than that, there is virtually no difference between the static and dynamic way of scaffolding.

4.1.2 Scaffolding in Grails

Grails supports both static and dynamic scaffolding. The First example will show how to work with the static scaffolding. And afterwards there will be a small example on how to use dynamic scaffolding.

The same domain and domainclasses as stated in the next chapter (*4.2 GORM*) will be used: Contact, Company, Employee and Address.

Static Scaffolding

To generate the scaffolds in Grails, the following command will need to be executed through the command-line inside the project directory:

```
grails generate-all [Domain Class]
```

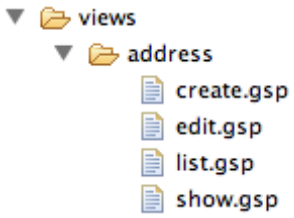

Figure 10: Generated views for the Address model with Static Scaffolding

In this example, the 'Address' domain class will be scaffolded. The Controller and Views (Figure 10: Generated views for the Address model with Static Scaffolding) will now be generated for the developer. The way the code is generated, is based on templates that can be changed.

Next to the CRUD actions, Grails adds the “list” action. The “edit” and “save” functions work together with “update” and “create”.

• Create	-	create.gsp	-	Creates a new address
• Read	-	show.gsp	-	Show one address (Based on ID)
• Update	-	edit.gsp	-	Retrieve one address (Based on ID) that needs to be changed
• Delete	-	*No view*	-	Delete one address (Based on ID)
• List	-	list.gsp	-	List all addresses
• Edit	-	*No view*	-	Save the updated address
• Save	-	*No view*	-	Saves the newly created address

The AddressController now looks like this. The functions are self-explanatory.

```
class AddressController {

    def index = { redirect(action:list,params:params) }

    // the delete, save and update actions only accept POST requests
    def allowedMethods = [delete:'POST', save:'POST', update:'POST']

    def list = { //... }

    def show = { //... }

    def delete = { //... }

    def edit = { //... }

    def update = { //... }

    def create = { //... }

    def save = { //... }
}
```

The entire AddressController can be found in appendix 9.2.

Dynamic Scaffolding

Dynamic Scaffolding in Grails is really easy. This can be achieved by adding the following line inside the controller that needs to be scaffolded.

```
def scaffold = true
```

The CRUD actions and views will be automatically generated at runtime for the previously defined model.

4.2 GORM

As noted before, the Grails framework heavily relies on predefined folder structures and other conventions in an attempt to reduce the amount of configuration that is needed for the different assets of the framework. As a part of this, domain classes are accommodated in a separate folder. This way, Grails can identify and treat them as such.

By default, Grails assumes that every domain class has to be persistent. It uses an technique called Grails Object Relational Mapping also shorted as GORM [3], to be able to automatically persist the data to a number of different data stores. GORM uses Hibernate under the hood to achieve this [9].

When creating an application, Grails automatically creates a Datasource-class where configurations for different environments can be defined. This allows for an easy switch between the development, test and production stage of a project. For instance, an in-memory database can be used for development, a local MySql-instance for testing, while the application, when in production, will use an existing remote database.

To illustrate how GORM works, the contact list-case will be reused, be it in a slightly extended form. The contact list consists of a number of contacts, each with a first- and last name, birthday, phone number, e-mail and address. Furthermore, a contact can have null or more jobs. As an employee, a contact has a job title, a phone number, an e-mail address and optionally a fax-number. Off course, an employee is also related to a certain company. A company has a name and an address.

When translated to Groovy classes, there might be an Contact-class that looks somewhat like the following;

```
import java.util.Date;

class Contact {
      String firstname
      String lastname
      Date birthday
      String phonenumber
      String emailaddress
      Address address
      //... some sort of list of jobs
}
```

In the above example, all of the properties except for the Address, will resolve in homonymous columns in a table called 'contact'. The address however, is accommodated in a separate class, and will therefore resolve in a table called 'address', which is referenced by a one-to-one relationship.

In order for the contact to be able to have multiple jobs, there has to be a one-to-many relationship between 'contact' and 'employee'. Grails enables this behavior by jet another convention.

```
static hasMany = [jobs:Employee]
```

When a static property called 'hasMany' is used, Grails recognizes that this class has a one-to-many relationship with one or more classes. This 'hasMany'-property is expected to be a map. This enables a reference to multiple classes, which are each named by their corresponding key.

An employee by itself is meaningless; it's only when references to both a contact and a company are made, that an employee can be identified by it's name and it's employer. Therefore, the relationship between both employee and contact and employee and company have to be bidirectional. This ensures that when either a company or a contact is deleted, it's accompanying employee's will be deleted too. In Grails, this is done through use of the 'belongsTo'-property

```
class Employee {
      Contact contact
      Company company
      String phonenumber
      String emailaddress
      String fax
      String jobtitle
      static belongsTo = Contact, Company
}
```

Besides these relations, some constraints have to be defined to be able to ensure the validity of the user input. There are numerous types of validation rules that can be applied, but in this case only the format of the zip-code has to be validated, which in this case means 4 numbers, an optional whitespace and two letters.

The code to implement this is pretty straightforward;

```
class Address {
      static constraints = {
               zipcode(matches:"\\d{4}\\s?[a-zA-Z]{2}")
      }

      String city
      String address
      String zipcode
      String country
}
```

The static property constraints expects an closure wherein the constraints are declared. In this case, the property zip-code is validated by checking it's value against a regular expression.

By default, Grails expects all of the properties to be mandatory. To override the default behavior, a list of optionals can be defined, in this case the 'fax'-property of the 'employee'-class.

```
static optionals = ["fax"]
```

4.2.1 Using GORM with legacy database schemas

Usually, the best way to use GORM is to let it automatically manage the database, and the tables it uses to store the data. Sadly, this way isn't always possible. In these cases, a custom mapping is needed to use to be able tot use GORM at all.

Creating custom mapping is done by explicitly specifying the table and the column used to store data on a request to save the object. This mapping looks like this:

```
class Member {
      String  name
      String  surname
      String  street
      Integer nr

      static mapping = {
               table 'person'
               name column:'name_first'
               surname column:'name_last'
               street column:'address'
               nr column:'house_nr'
               }
}
```

It is possible to only specify the table to use. In this case, Grails will try to use this table as it sees fit, possibly alerting the table if permitted. The mapping per variable prevents this, and makes sure all data is written to the right columns.

It is also possible to create one-to-one,one-to-many and many-to-many mappings using this way of mapping, as in the examples shown below:

One-to-One

```
class Member {

	String  name
	String  surname
	String  street
	Integer nr
	Occupation occupation

	static mapping = {
		table 'person'
		name column:'name_first'
		surname column:'name_last'
		street column:'address'
		nr column:'house_nr'
		occupation column:'jobID'
		}
	String toString()
	{
		name + " " + surname
	}
}
```

One if the simplest relations in the one-to-one relation. Specifying the table where the object is to be saved gives enough information to Grails to save all objects and fill in the foreign keys. In fact, the technique it's very similar to saving the data of an ordinary field.

One-to-Many relations are slightly more complex. On the “one” side of the relation, the configuration is exactly the same as the one found in a one-to-one relationship. The “many” side of the relation needs both the “`hasMany`” declaration (as seen below) and a mapping to save the relationship properly. The example below shows the “many” side of the One-to-Many and a side of a Many-to-Many relationship:

One-to-Many

```
class Organization {

	String name
	Integer companyCode

	static belongsTo = Land
	static hasMany = [employees:Occupation, lands:Land]

	static mapping = {
		table 'company'
		name column:'name'
		companyCode column:'number'
		employees column:'companyID'
		lands column:'Organization_Id',joinTable:'Land_Organization_Link'
		}

	String toString()
	{
		name
	}
}
```

To properly be able to save the many-to-many relationship, one of the sides must be the controller, and only a call to the `save()` function of the controller will save all the objects and the relationship to the database. Specifying the controller is done by adding the "`belongsTo`" keyword to the non-controlling side of the relationship, as can be seen in the example shown above. To properly create a many-to-many relationship a join-table is needed. If this table is not explicitly specified, Grails will create one on it's own, using it's own naming schema. To make sure Grails uses the existing table to save the relationship, a mapping specifying the join-table is used. This can be seen in the examples above and below.

Many-to-Many

```
class Land {

      String country
      String code

      static hasMany = [organizations:Organization]

      static mapping = {
               table 'country'
               country column:'name'
               code column:'landcode'
               organizations column:'Land_Id',joinTable:'Land_Organization_Link'
               }
}
```

As seen in the examples above, it is relatively simple to create a custom ORM mapping when this is needed. It is both easier and faster to let Grails take care of the ORM then specifying the structure manually. It is recommended not to use the method above unless the standard Grails approach can not be used [6].

4.3 REST in Grails

When people think about enterprise applications and solutions, the term REST (Represintational State Transfer) might be said once or twice. The term is often used in a looser sense to describe any simple interface which transmits domain-specific data over HTTP without an additional messaging layer such as SOAP or session tracking over HTTP cookies.

While the REST is not only for use on the web, the world wide web is a good example of a REST design. This leans on four types of requests that can be send to an external server: POST, GET, PUT and DELETE.

The popularity on webservices is growing, proving to be useful in situations where data has to be exchanged between different applications. Often webservices work with a SOAP protocol, but REST doesn't do that and gives whatever information is needed (or requested) by the consumer.

4.3.1 How to create a REST environment in Grails

Since Groovy has got a nice webdriven brother called Grails the question risen on what it can do to realize a REST environment. Luckily for those people there is Grails, because it *is* RESTful baby!

If one has looked into developing in Grails, they should have already seen the convenience of having the controller do all the navigation through your website. This can come in handy when developing a RESTful service.

Let's say there is a user database in a Grails environment and the model and controller have already been generated. The URI's would look something like:

- http://www.example.com/grailsproject/user/index
- http://www.example.com/grailsproject/user/1/view

But to make it work according to the REST principles the action doesn't have to be defined in the URI, but rather given to the server through the HTTP headers.

Then URI's will start to look like:

- http://www.example.com/grailsproject/users/
- http://www.example.com/grailsproject/users/1/

Note the use of a plural, since we are not directly talking to the controller in Grails anymore.

On the last URI's the standard HTTP actions can be preformed. A simple table describes the actions in different situations:

Resource	GET	PUT	POST	DELETE
Users root http://example.com/grailsproject/users/	**List** the members of the collection. For example list all the cars for sale.	Not generally used. Meaning defined as replace the entire collection with another entire collection.	**Create** a new entry in the collection where the ID is assigned automatically by the collection. The ID created is typically returned by this operation.	Not generally Used. Meaning defined as delete the entire collection.
Given user with 1 http://example.com/grailsproject/users/1/	**Retrieve** the addressed member of the collection	**Update** the addressed member of the collection or create it with a defined ID.	Not generally used. Use the root for this.	**Delete** the addressed member of the collection.

4.3.2 Getting it to work

The code example below shows a dull domain class that is created in Grails:

```
class User {
      String firstName
      String surName
}
```

And the following controller:

```
class UserController {

      def index = {
              //nothing here
      }
}
```

First thing needed for a RESTful service is a way to control the different HTTP actions given to the server and a root URI where a client can call to.

This is done in the UrlMapping file found in the groovy/conf folder.

```
class UrlMappings {
      static mappings = {
               "/users/$id?" {
                controller = "user"
                action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
               }
      }
}
```

This maps all the requests to users and a given id. With the controller property the required controller is defined. The action defines which actions to perform in the controller. In order to make this work the controller has to be adjusted.

The Grails application has to be restarted in order to make the mapping work.

The following actions are added to the controller:

```
def show = {}
def update = {}
def delete = {}
def save = {}
```

The code and what to do with the requests can be defined in the controller now.

There is also a minor thing changed in the controller. The domain-class User has to be included, which can be done with the following code:

```
def user = new User()
```

To see it work an easy GSP page can be used. The user controller can do this.

The grails-app/views/user/ folder contains the views used by the UserController. An index file is created (index.gsp).

In appendix 9.3 RESTful Webpage there is an example page on how to perform a simple HTTP actions on a Grails server and is created for a Mozilla (Firefox) browser.

With this easy HTML page REST requests are send to the server and JavaScript alerts the output to your browser.

All there is left to do is adjust the controller to it handles the requests.

```
def show = {
            render 'Showing ' + this.user.firstName + ' ' + this.user.surName
      }

      def update = {
            render 'Going to update ' + params['firstName'] + " " + params['surName']
      }

      def delete = {
            render 'Going to delete ' + this.user.firstName + ' ' + this.user.surName
      }

      def save = {
            render 'Going to save ' + params['firstName'] + " " + params['surName']
      }
```

Now requests are handled by the controller, working with the parameters given by the HTTP request.

The basics of the RESTful service (and the test page) are now ready.

5

Conclusion

" A conclusion is the place where you got tired of thinking. "
- Harold Fricklestein

Groovy and Grails contain a lot of good things, but also some bad features. This chapter will our point of view on various aspects of Groovy and Grails. This document has been based on the following question:

'What are the characteristics of Groovy (and Grails) and what impact do they have for an implementation in an SOA within enterprise applications?'

5.1 GDK

Groovy has been around for a while, and we think it is a good tool to use along Java. Its syntax is relatively simple yet powerful, it is capable to incorporate Java, and has many interesting added features. The details of these features can be found below:

Simple

Although the Groovy syntax is fairly simple and much more expressive than Java, the language itself isn't any less powerful. Groovy's developers believe that simple tasks should be simple, and more complex tasks should be possible. We think that that has been achieved, as our experience did indeed prove most tasks where achievable with less code than would be needed in Java, and more often if not, simpler than Java.

Capable of using existing Java code

One of Groovy's major strengths is its capability to incorporate Java into its code. This allows for a very smooth transition for a Java programmer to become a Groovy programmer, since the switch can made gradually and Java code works flawlessly in groovy. This feature also allows Groovy to use any excising Java framework. We believe this feature gives Groovy a head start over other new and emerging languages. A team working in a different language will be writing all the required code when no framework is available, Groovy can benefit from the numerous Java frameworks already created and thus save a development team quite a bit of work.

All of this is only possible because Groovy compiles to Java bytecode, and then runs this code on the same virtual machine as ordinary Java code.

Added functionality

Groovy has a few characteristics that may not be unique for it as a language, but do show their usefulness in any Java development environment, and not just a specific SOA development. The details about these characteristics can be found in earlier chapters and include feats like closures and the the Gstring. We believe that these tools, along with the Groovy syntax, will create an initial hit on the productivity while the development team is learning and gaining experience. After the tools are mastered enough, we believe the productivity of the team will return and then rise above its previous level. Since we believe that any development team is interested in increasing its output, we think these added feats will increase the attractiveness of Groovy.

Ready or not

Groovy is still a young language, and this is clear by the IDE support. Code completion is therefore not always available, especially when the dynamic features of Groovy are used. This makes developing more of a choir than it needs to be. But his also happens when other Dynamic Languages are used. Though it might not be a Groovy problem, it can become a reason for not choosing for Groovy as the language to use. We are sure that in the future more features will become available for IDEs.

Also, not all functionalities that might be needed are available as a specific Groovy framework or extension. This can be circumvented by using Java frameworks, but we believe this does decrease the overall benefit of using Groovy.

5.2 Groovy in a SOA

Contract first

In a SOA environment it's important to use the contract first approach. One of the reasons is that you want to make your webservice compatible with other platforms. The other reason is that if you redeploy the webservice there might occur some changes in the contract.

If you change your contract you have to notify all subscribers, and maybe they have to change their code too. This is something you don't want to have in your enterprise application. What you want is a contract that is consistent and used as long as possible.

API

While researching the GroovyWS module we couldn't find an API. And the examples on the Groovy homepage didn't work. We had to download the source code and and generate the Javadoc with the help of some eclipse plugins. GroovyWS is still in development, the version we used was version 0.4. This is another prove that GroovyWS isn't mature enough. The problem with this is that some parts of Groovy are just not ready yet. It might be fixed in the future, but you can never be sure.

Security

GroovyWS doesn't support any security except for one, and that is the Basic Authentication. So if you want to apply security in your webservice in Groovy, which is often the case is, you can't do it because it isn't supported (yet).

You can only use Basic Authentication, and this security method isn't a WS-* standard.

Groovy and SOA

We researched the Groovy language, and its role in SOA-environments. We noticed that it's very easy to set up a webservice and to consume one. Despite that its easy to set up a webservice and consume a webservice we concluded that GroovyWS is not mature enough to use in a SOA environment.

5.3 Grails

REST

The thing about REST in Groovy is that it has two kinds of solving the different kinds of implementation methods. Do you want to implement it in Groovy itself with the use of RESTlets or use Grails to support you in the development process?

REST implementation through Grails is widely used, and its support is good. Some people might say RESTlets work better, which we can't confirm. RESTlets is the Java solution to creating a RESTful service, and has been modified to work in Groovy. The choice between RESTlet and Grails depends on what a developer needs for his application and what environment he wishes to use. If the developer likes using the MVC pattern and handeling REST as CRUD, Grails is recommended. Some developers might not like these enforcements, those had better stick with RESTlets.

Grails supports two Java frameworks, which will come in handy when creating dynamic websites; namely Hibernate and Spring.

Grails provides UrlMapping and the binding of different actions to various HTTP request methods. This convenience combined with the other functionality Grails provides makes this a powerful framework to create a RESTful service on.

In a short amount of time we got a RESTful service running. Reachable by any kind of client / browser. Combining the power of Groovy and Grails to generate XML responses will lead to awesome results.

We were amazed by the features we could use to create services, everything just worked that way it should. That's why we recommend using Grails for your next RESTful service instead of Java RESTlets.

Work fast, create more productivity

Compared to a lot of developers using traditional Java web frameworks, Grails developers have a higher productivity. This is caused by various factors;

The fact that XML configuration is not needed is a large benefit. XML is needed in order to make a lot of other frameworks work (eg. Mule in Java). Grails will automatically configure a lot of settings. That saves the developer a lot of time.

When starting a Grails project, Grails will build an entire environment for you to start working in. This environment contains support for the MVC and all the source folder are separated in a great way. IDE's like Eclipse will understand the structure and you'll be able to integrate it easily into a project layout.

Model Persistence

If one creates a domain class, it is put in the model part of your application. Using Hibernate or just the dynamic methods that you can perform on domainclass objects, the persistence with the database will be kept. This feature provides us the possibility to easily maintain our data. No more writing lots of SQL!

An even easier way to do this is using scaffolding. Scaffolding enables you to create management pages for your data in just a few steps. It sure works, and it's awesome to not create the pages yourself anymore. With a simple command in your terminal the GSP pages can be generated and adjusted by the developers with ease.

This part makes Grails that much faster and convenient than a lot of other web frameworks. The generation of code lifts a heavy load off our shoulders, it is the future.

Overall conclusion about Grails

Would we use Grails on our next web project? Depends on the size of the project and the time available we'd have to put in the project. Because Grails is quite new to us and we had a short timespan, we didn't have time to create full blown projects in Grails.

However, if given a little extra time, we're sure that Grails will improve the development process of the following projects. Which is the effect of most of the web frameworks. There is always a learning curve when trying out / using a new framework. But given the many possibilities Grails has to offer; we'd go for it! It will surely safe you time.

6

Recommendations

> *" Question: How does a large software project get to be one year late? Answer: One day at a time! "*
>
> *- Fred Brooks*

After our time spend researching Groovy and it's possibilities, we drew a few conclusions about what Groovy can and can not do (yet). That's why we think that, on itself, Groovy is maturing quickly and can be used in active development. However, if Groovy is used in a SOA, some features and functions are needed that Groovy alone cannot provide. For any other language, this may prove to be a serious problem. However, since Groovy is able to use frameworks created for Java, or Java itself if desired, these can be used to provide functionality that is lacking in Groovy. This makes Groovy just as a valid choice as Java, and maybe even a better one, since Groovy adds lots of features that may speed up development.

But just because Groovy is usable and, as we believe, ready to be used in development, doesn't mean it will be as well. Especially since humans cling very strongly to their habits, and are often not very interested in change. We believe however, that Groovy has enough benefits to make it worth the while to switch due to it unique nature and should be recommended as a second language for many Java developers looking for a change.

7

Bibliography

" You will find it a very good practice always to verify your references sir. "
- Martin Routh

1. Van Buuren, Hans Hummel, Hans [and others] Wolters Noordhoff (2003) *Onderzoek de basis*. ISBN 90 01 18259 3
2. LWSVO, Vereniging van onderwijsmediathecarissen (2005) *Richtlijnen Bronvermelding*. Consulted on 17 September 2008, http://lwsvo.nl/
3. Davis, Scott The Pragmatic Programmers (2008) *Groovy Recipes – Greasing the Wheels of Java.* ISBN 0 9787392 9 9
4. König, Dierk [and others] Manning Publications Co. (2007) *Groovy in Action.* ISBN 1 932394 84 2
5. Subramaniam, Venkat The Pragmatic Progammers (2008) *Programming Groovy.* ISBN 1 934356 09 3
6. Rocher, Keith G. Apress (2006) *The Definitive Guide to Grails*. ISBN 1 59059 758 3
7. Judd, Christopher M. Nusairait, Faisal J. Apress (2008) *Beginning Groovy and Grails*. ISBN 978 1 4302 1045 0
8. General Groovy information (Developers page). Consulted on 17 september 2008, http://groovy.codehaus.org/
9. General Grails information (Developers page). Consulted on 17 september 2008, http://grails.org/
10. Erl, T Prentice Hall PTR (2005) *Service-Oriented Architecture: Concepts, Technology, and Design*. ISBN 0 13 185858 0
11. Bokor, L. (Java Magazine - Year of publication 7, September 2008, Number 3, Array Publications) Pages 59 till 62 *Even wennen, maar dan wordt het: 'Groovy, baby!'*
12. Overdijk, M. (Java Magazine - Year of publication 7, October 2008, Number 4, Array Publications) Pages 44 till 48 *Grails & Groovy: De zoektocht is voorbij*
13. ZhenChun Huang, C. H. (2005) *Groovy Service: On-Demand Web Service by Script Language*.
14. Kimsal, M. Klein, D. [and others] *GroovyMag for Groovy and Grails Developers* Volume one, issue one, November 2008
15. Kimsal, M. Klein, D. [and others] *GroovyMag for Groovy and Grails Developers* Volume one, issue two, December 2008
16. Glover, A. (2005) *Practically Groovy: Smooth operators.* Consulted on 21 October 2008, http://www.ibm.com/developerworks/java/library/j-pg10255.html
17. Krzywda, A. (17 December 2006) *Building a GUI with Groovy*. Consulted on 26 September 2008, http://tinyurl.com/5hz6e3
18. Leach, J. (Syger 2007) *Grails WebAlbum*. Consulted on 5 December 2008, http://www.syger.it/Tutorials/GrailsWebAlbum.html
19. Rudolph, J. (2007) InfoQ *Getting Started with Grails.* ISBN 978-1-4303-0782-2
20. Almiray, A. (30 January 2008) *MetaProgramming with Groovy.* Consulted on 22 October 2008, http://groovy.dzone.com/articles/metaprogramming-groovy-i
21. Fremantle, P. (4 December 2007) *Paul Fremantle on Making SOA Groovy -- A TSS Video.* Consulted on 5 November 2008, http://tinyurl.com/634o7j
22. Glover, A. (2005) *Practically Groovy: Of MOPs and mini-languages.* Consulted on 21 October 2008, http://www.ibm.com/developerworks/java/library/j-pg09205/index.html
23. Apache CXF (Date unknown) *Supplying a Configuration file to CXF*. Consulted on 3 December 2008, http://cwiki.apache.org/CXF20DOC/configuration.html
24. Gardner, D. (13 June 2008) *Apache CXF: All Grown Up.* Consulted on 3 December 2008, http://www.linuxinsider.com/story/63388.html.ONJava.com
25. Wielenga, G. (3 November 2007) *Groovy Web Service*. Consulted on 28 November 2008, http://blogs.sun.com/geertjan/entry/groovy_web_service
26. General XFire information (Developers page). Consulted on 28 September 2008, http://xfire.codehaus.org/

27. Groovy (Programming Language) on Wikipedia. Consulted on 12 September 2008, http://en.wikipedia.org/wiki/Groovy_(programming_language)
28. Strachan, G (29 August 2003) *Groovy - the birth of a new dynamic language for the Java platform*. Consulted on 12 September 2008, http://radio.weblogs.com/0112098/2003/08/29.html
29. Hammant, P [and others] (12 May 2007) *Simple Java and .NET SOA interoperability*. Consulted on 5 October 2008, http://www.infoq.com/articles/REST-INTEROP
30. Microsoft Corporation (7 November 2008) *Windows API*. Consulted on 19 December 2008. http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx
31. Yegge, S. (11 May 2008) *Dynamic Languages Strike Back*. Consulted on 6 December 2008. http://tinyurl.com/6a6den
32. Rado, D. (28 April 2007) *Early vs. Late Binding*. Consulted on 6 December 2008. http://word.mvps.org/fAQs/InterDev/EarlyvsLateBinding.htm
33. Galleon (3 March 2008) *Groovy SOAP*. Consulted on 12 November 2008. http://docs.codehaus.org/display/GROOVY/Groovy+SOAP
34. Apache WSS4J (6 February 2008) *Apache WSS4J*. Consulted on 24 November 2008. http://ws.apache.org/wss4j/
35. Grails (Framework) on Wikipedia. Consulted on 12 September 2008, http://en.wikipedia.org/wiki/Grails_(Framework)
36. Scaffold (programming). Consulted on 25 September 2008, http://en.wikipedia.org/wiki/Scaffold_(programming)
37. Dynamic programming language - Wikipedia, the free encyclopedia. (n.d.). . Retrieved January 8, 2009, from http://en.wikipedia.org/wiki/Dynamic_programming_language
38. Groovy - ExpandoMetaClass. (n.d.). . Retrieved January 8, 2009, from http://groovy.codehaus.org/ExpandoMetaClass
39. Technorabble: Collection of Dynamic Features for Static Languages. (n.d.). . Retrieved January 8, 2009, from http://tech.norabble.com/2006/08/collection-of-dynamic-features-for.html
40. Bitwise Magazine:: Dynamic Languages - Who Needs Them? (n.d.). . Retrieved January 8, 2009, from http://www.bitwisemag.com/2/Dynamic-Languages-Who-Needs-Them
41. Groovy - Using MockFor and StubFor. (n.d.). . Retrieved January 8, 2009, from http://groovy.codehaus.org/Using+MockFor+and+StubFor.

Glossary

" Why is 'abbreviation' such a long word? "
- Author Unknown

- **Annotations**

  An annotation, in the Java computer programming language, is a special form of syntactic metadata that can be added to Java source code. Classes, methods, variables, parameters and packages may be annotated. Unlike Javadoc tags, Java annotations are reflective in that they are embedded in class files generated by the compiler and may be retained by the Java VM to be made retrievable at run-time.

  Source: http://en.wikipedia.org/wiki/Java_annotation (19 December 2008)

- ***API*** **Application programming interface**

  An application programming interface (API) is a set of functions, procedures, methods, classes or protocols that an operating system, library or service provides to support requests made by computer programs.

  Source: http://en.wikipedia.org/wiki/API (18 December 2008)

- **Closure**

  A Groovy closure is like a "code block" or a method pointer. It is a piece of code that is defined and then executed at a later point.

  Source: http://groovy.codehaus.org/Closures (19 December 2008)

- ***CRUD*** **Create, read, update and delete**

  Create, read, update and delete (CRUD) are the four basic functions of persistent storage, a major part of nearly all computer software. Sometimes CRUD is expanded with the words retrieve instead of read or destroy instead of delete. It is also sometimes used to describe user interface conventions that facilitate viewing, searching, and changing information; often using computer-based forms and reports.

  Source: http://en.wikipedia.org/wiki/Create,_read,_update_and_delete (18 December 2008)

- ***CXF*** **Apache CXF**

  Apache CXF is an open-source fully featured easy to use Web Services framework. It is the combination of two projects: Celtix developed by IONA and XFire developed by Codehaus working together at the Apache Software Foundation.

  Source: http://en.wikipedia.org/wiki/CXF (18 December 2008)

- ***DOM*** **Document Object Model**

  The Document Object Model (DOM) is a platform- and language-independent standard object model for representing HTML or XML and related formats.

  Source: http://en.wikipedia.org/wiki/DOM (18 December 2008)

- ***DSL*** **Domain-specific language**

  The term domain-specific language (DSL) has become popular in recent years in software development to indicate a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. The concept isn't new—special-purpose programming languages and all kinds of modeling/specification languages have always existed, but the term has become more popular due to the rise of domain-specific modeling. Domain-specific languages are 4GL programming languages.

  Source: http://en.wikipedia.org/wiki/Domain-specific_language (18 December 2008)

- **Dynamic Language**

  Dynamic programming language is a term used broadly in computer science to describe a class of high-level programming languages that execute at runtime many common behaviors that other languages might perform during compilation, if at all. These behaviors could include extension of the program, by adding new code, by extending objects and definitions, or by modifying the type system, all during program execution. These behaviors can be emulated in nearly any language of sufficient complexity, but dynamic languages provide direct tools to make use of them.

  Source: http://en.wikipedia.org/wiki/Dynamic_programming_language (19 December 2008)

- **Endpoint**

  In service-oriented architecture, an endpoint is the entry point to a service, a process, or a queue or topic destination

  Source: http://en.wikipedia.org/wiki/Endpoint (19 December 2008)

- ***ESB*** **Enterprise Service Bus**
  In computing, an enterprise service bus (ESB) refers to a software architecture construct. This construct is typically implemented by technologies found in a category of middleware infrastructure products, usually based on recognized standards, which provide fundamental services for complex architectures via an event-driven and standards-based messaging engine (the bus).
  Source: http://en.wikipedia.org/wiki/Enterprise_service_bus (18 December 2008)

- **Framework**
  A software framework is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality.
  Source: http://en.wikipedia.org/wiki/Software_framework (19 December 2008)

- ***GDK*** **Groovy Development Kit**
  The GDK is the Groovy version of the JDK (Java Development Kit).

- ***GORM*** **Grails Object Relational Mapping**
  GORM is Grails' object relational mapping (ORM) implementation. Under the hood it uses Hibernate 3 (an extremely popular and flexible open source ORM solution) but because of the dynamic nature of Groovy, the fact that it supports both static and dynamic typing, and the convention of Grails there is less configuration involved in creating Grails domain classes.
  Source: http://grails.org/GORM (18 December 2008)

- **Grails**
  Grails is an open source web application framework which leverages the Groovy programming language (which is in turn based on the Java platform). It is intended to be a high-productivity framework by following the "coding by convention" paradigm, providing a stand-alone development environment and hiding much of the configuration detail from the developer.
  Source: http://en.wikipedia.org/wiki/Grails_(framework) (19 December 2008)

- **Groovy**
  Groovy is an object-oriented programming language for the Java Platform as an alternative to the Java programming language. It is a dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk. It can be used as a scripting language for the Java Platform.
  Source: http://en.wikipedia.org/wiki/Groovy_(programming_language) (19 December 2008)

- **GroovySOAP**
  SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. Groovy has a SOAP implementation based on XFire which allows you to create a SOAP server and/or make calls to remote SOAP servers using Groovy.
  Source: http://groovy.codehaus.org/Groovy+SOAP (19 December 2008)

- **GroovyWS**
  GroovyWS is a library that allows rapid transparent integration with SOAP-based web services from Groovy scripts and classes. It provides an auto-generated service proxy based on the WSDL of the remote service and automatically generates class bindings for any complex types that are required. GroovyWS is based on Apache CXF and runs on Java 5 or later.
  Source: http://www.ohloh.net/p/10588 (19 December 2008)

- ***GUI*** **Graphical user interface**
  A graphical user interface (GUI) is a type of user interface which allows people to interact with electronic devices such as computers, hand-held devices (MP3 Players, Portable Media Players, Gaming devices), household appliances and office equipment. A GUI offers graphical icons, and visual indicators as opposed to text-based interfaces, typed command labels or text navigation to fully represent the information and actions available to a user. The actions are usually performed through direct manipulation of the graphical elements.
  Source: http://en.wikipedia.org/wiki/GUI (18 December 2008)

- **GString**
  Strings that are declared inside double-quotes (i.e. either single double-quotes or triple double-quotes for multi-line strings) can contain arbitrary expressions inside them as shown above using the ${expression} syntax in a similar way to JSP EL, Velocity and Jexl. Any valid Groovy expression can be enclosed in the ${...} including method calls etc. GStrings are defined the same way as normal Strings would be created in Java.
  Source: http://groovy.codehaus.org/Strings+and+GString (19 December 2008)

- ***HSQLDB*** **Hyperthreaded Structured Query Language Database**
  HSQLDB (Hyperthreaded Structured Query Language Database) is a relational database management system written in Java. It is based on Thomas Mueller's discontinued Hypersonic SQL Project.
  Source: http://en.wikipedia.org/wiki/HSQLDB (18 December 2008)

- ***HTTP(S)*** **Hypertext Transfer Protocol (over Secure Socket Layer)**
  Hypertext Transfer Protocol (HTTP) is a communications protocol. Its use for retrieving inter-linked text documents (hypertext) led to the establishment of the World Wide Web.
  Source: http://en.wikipedia.org/wiki/HTTP (18 December 2008)

- ***IDE*** **Integrated development environment**
  An integrated development environment (IDE) also known as integrated design environment or integrated debugging environment is a software application that provides comprehensive facilities to computer programmers for software development.
  Source: http://en.wikipedia.org/wiki/Integrated_development_environment (18 December 2008)

- ***JDBC*** **Java Database Connectivity**
  Java Database Connectivity (JDBC) is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases.
  Source: http://en.wikipedia.org/wiki/JDBC (18 December 2008)

- ***JDK*** **Java Development Kit**
  The Java Development Kit (JDK) is a Sun Microsystems product aimed at Java developers. Since the introduction of Java, it has been by far the most widely used Java SDK.
  Source: http://en.wikipedia.org/wiki/JDK (18 December 2008)

- ***JSP*** **JavaServer Pages**
  JavaServer Pages (JSP) is a Java technology that allows software developers to dynamically generate HTML, XML or other types of documents in response to a Web client request. The technology allows Java code and certain pre-defined actions to be embedded into static content.
  Source: http://en.wikipedia.org/wiki/JavaServer_Pages (18 December 2008)

- ***JVM*** **Java Virtual Machine**
  A Java Virtual Machine (JVM) is a set of computer software programs and data structures which use a virtual machine model for the execution of other computer programs and scripts. The model used by a JVM accepts a form of computer intermediate language commonly referred to as Java bytecode. This language conceptually represents the instruction set of a stack-oriented, capability architecture.
  Source: http://en.wikipedia.org/wiki/JVM (18 December 2008)

- **MetaClass**
  In object-oriented programming, a metaclass is a class whose instances are classes. Just as an ordinary class defines the behavior of certain objects, a metaclass defines the behavior of certain classes and their instances.
  Source: http://en.wikipedia.org/wiki/Metaclass (19 December 2008)

- ***MOP*** **Meta Object Protocol**
  A metaobject protocol (MOP) is an interpreter of the semantics of a program that is open and extensible. Therefore, a MOP determines what a program means and what its behavior is, and it is extensible in that a programmer (or metaprogrammer) can alter program behavior by extending parts of the MOP. The MOP exposes some or all internal structure of the interpreter to the programmer. The MOP may manifest as a set of classes and methods that allow a program to inspect the state of the supporting system and alter its behaviour. MOPs are implemented as object-oriented programs where all objects are metaobjects.
  Source: http://en.wikipedia.org/wiki/Metaobject_Protocol (18 December 2008)

- ***MVC*** **Model-View-Controller**
  Model-view-controller (MVC) is an architectural pattern used in software engineering. Successful use of the pattern isolates business logic from user interface considerations, resulting in an application where it is easier to modify either the visual appearance of the application or the underlying business rules without affecting the other. In MVC, the model represents the information (the data) of the application; the view corresponds to elements of the user interface such as text, checkbox items, and so forth; and the controller manages the communication of data and the business rules used to manipulate the data to and from the model.
  Source: http://en.wikipedia.org/wiki/Model-view-controller (18 December 2008)

- ***OO(P)*** **Object Oriented Programming**
  Object-oriented programming (OOP) is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. Programming techniques may include features such as encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application development until the early 1990s.
  Source: http://en.wikipedia.org/wiki/Object-oriented_programming (18 December 2008)

- **Operator overloading**
  Operator overloading (less commonly known as operator ad-hoc polymorphism) is a specific case of polymorphism in which some or all of operators like +, =, or == have different implementations depending on the types of their arguments. Sometimes the overloadings are defined by the language; sometimes the programmer can implement support for new types.
  Source: http://en.wikipedia.org/wiki/Operator_overloading (19 December 2008)

- ***POGO*** **Plain Old Groovy Object**
  Groovy version of a POJO (Plain Old Java Object).

- ***POJO*** **Plain Old Java Object**
  POJO is an acronym for Plain Old Java Object. The name is used to emphasize that the object in question is an ordinary Java Object, not a special object, and in particular not an Enterprise JavaBean (especially before EJB 3).
  Source: http://en.wikipedia.org/wiki/POJO (18 December 2008)

- ***REST*** **Representational State Transfer**
  Representational state transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. As such, it is not strictly a method for building what are sometimes called "web services." The terms “representational state transfer” and “REST” were introduced in 2000 in the doctoral dissertation of Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification. The terms have since come into widespread use in the networking community.
  Source: http://en.wikipedia.org/wiki/REST (18 December 2008)

- **Scaffolding**

  Scaffolding is a meta-programming method of building database-backed software applications. It is a technique supported by some model-view-controller frameworks, in which the programmer may write a specification that describes how the application database may be used. The compiler uses this specification to generate code that the application can use to create, read, update and delete database entries, effectively treating the template as a "scaffold" on which to build a more powerful application.

  Source: http://en.wikipedia.org/wiki/Scaffold_(programming) (19 December 2008)

- ***SDK*** **Software development kit**

  A software development kit (SDK or "devkit") is typically a set of development tools that allows a software engineer to create applications for a certain software package, software framework, hardware platform, computer system, video game console, operating system, or similar platform.

  Source: http://en.wikipedia.org/wiki/SDK (18 December 2008)

- ***SOA*** **Service-Oriented Architecture**

  Service-oriented architecture (SOA) provides methods for systems development and integration where systems group functionality around business processes and package these as interoperable services. SOA also describes IT infrastructure which allows different applications to exchange data with one another as they participate in business processes. Service-orientation aims at a loose coupling of services with operating systems, programming languages and other technologies which underlie applications. SOA separates functions into distinct units, or services, which developers make accessible over a network in order that users can combine and reuse them in the production of business applications. These services communicate with each other by passing data from one service to another, or by coordinating an activity between two or more services.

  Source: http://en.wikipedia.org/wiki/Service-Oriented_Architecture (18 December 2008)

- ***SOAP*** **Simple Object Access Protocol**

  SOAP, originally defined as Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on Extensible Markup Language (XML) as its message format and usually relies on other Application Layer protocols, most notably Remote Procedure Call (RPC) and HTTP for message negotiation and transmission. SOAP forms the foundation layer of the web services protocol stack providing a basic messaging framework upon which abstract layers can be built.

  Source: http://en.wikipedia.org/wiki/SOAP_(protocol) (18 December 2008)

- ***SQL*** **Structured Query Language**

  SQL (Structured Query Language) is a database computer language designed for the retrieval and management of data in relational database management systems (RDBMS), database schema creation and modification, and database object access control management.

  Source: http://en.wikipedia.org/wiki/SQL (18 December 2008)

- ***StAX*** **Streaming API for XML**

  Streaming API for XML (StAX) is an application programming interface (API) to read and write XML documents in the Java programming language.

  Source: http://en.wikipedia.org/wiki/StAX (18 December 2008)

- **Tokens**

  A security token (or sometimes a hardware token, hard token, authentication token, cryptographic token, or key fob) may be a physical device that an authorized user of computer services is given to ease authentication. The term may also refer to software tokens.

  Source: http://en.wikipedia.org/wiki/Security_token (19 December 2008)

- ***URL*** **Uniform Resource Locator**

  Uniform Resource Locator (URL) is a type of Uniform Resource Identifier (URI) that specifies where an identified resource is available and the mechanism for retrieving it. In popular usage and in many technical documents and verbal discussions it is often, imprecisely and confusingly, used as a synonym for uniform resource identifier. The confusion in usage stems from historically different interpretations of the semantics of the terms involved. In popular language a URL is also referred to as a Web address.

  Source: http://en.wikipedia.org/wiki/URL (18 December 2008)

- **Web Service**

  A 'Web service' (also Web Service) is defined by the W3C as "a software system designed to support interoperable machine-to-machine interaction over a network". Web services are frequently just Web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services.

  Source: http://en.wikipedia.org/wiki/Webservice (19 December 2008)

- **WS-*** **Web Service Specifications**

  There are a variety of specifications associated with web services. These specifications are in varying degrees of maturity and are maintained or supported by various standards bodies and entities. Specifications may complement, overlap, and compete with each other. Web service specifications are occasionally referred to collectively as "WS-*", though there is not a single managed set of specifications that this consistently refers to, nor a recognized owning body across them all. The reference term "WS-*" is more of a general nod to the fact that many specifications are named with "WS-" as their prefix.

  Source: http://en.wikipedia.org/wiki/WS- (18 December 2008)*

- ***WSDL*** **Web Services Description Language**

  The Web Services Description Language is an XML-based language that provides a model for describing Web services.

  Source: http://en.wikipedia.org/wiki/Web_Services_Description_Language (18 December 2008)

- **XFire**

  Codehaus XFire is a Java framework for development and consumption of web services. Unlike similar products, such as Apache Axis 1.x, XFire uses StAX for XML processing, resulting in better performance. Apache Axis2 also uses StAX. Codehaus XFire is a java SOAP framework.

  Source: http://en.wikipedia.org/wiki/Codehaus_XFire (19 December 2008)

- ***XML*** **Extensible Markup Language**

  The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages. It is classified as an extensible language, because it allows the user to define the mark-up elements. XML's purpose is to aid information systems in sharing structured data, especially via the Internet, to encode documents, and to serialize data; in the last context, it compares with text-based serialization languages such as JSON and YAML.

  Source: http://en.wikipedia.org/wiki/XML (18 December 2008)

- **XSD** **XML Schema**

  XML Schema, published as a W3C recommendation in May 2001, is one of several XML schema languages. It was the first separate schema language for XML to achieve Recommendation status by the W3C. Like all XML schema languages, XML Schema can be used to express a schema: a set of rules to which an XML document must conform in order to be considered 'valid' according to that schema. However, unlike most other schema languages, XML Schema was also designed with the intent that determination of a document's validity would produce a collection of information adhering to specific data types. Such a post-validation infoset can be useful in the development of XML document processing software, but the schema language's dependence on specific data types has provoked criticism.

  Source: http://en.wikipedia.org/wiki/XSD (19 December 2008)

9

Appendices

9.1 Contacts Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="contacts">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="contact"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="contact">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="address" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="phonenumbers" minOccurs="0" maxOccurs="1"/>
         <xs:element ref="emailaddresses" minOccurs="0" maxOccurs="1" />
        <xs:element ref="job" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:string" />
      <xs:attribute name="birthday" use="required" type="xs:date"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="address">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="street" type="xs:string"></xs:element>
        <xs:element name="city" type="xs:string"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="emailaddresses">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="mailaddress" type="xs:string" minOccurs="1"
maxOccurs="unbounded"></xs:element>
        </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="phonenumbers">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="phonenumber" type="xs:string" minOccurs="1"
maxOccurs="unbounded"></xs:element>
        </xs:sequence>
      </xs:complexType>
  </xs:element>

  <xs:element name="job">
      <xs:complexType>
        <xs:attribute name="type" type="xs:string" use="required"></xs:attribute>
      </xs:complexType>
  </xs:element>
</xs:schema>
```

9.2 Address Controller

```
class AddressController {

    def index = { redirect(action:list,params:params) }

    // the delete, save and update actions only accept POST requests
    def allowedMethods = [delete:'POST', save:'POST', update:'POST']

    def list = {
        if(!params.max) params.max = 10
        [ addressInstanceList: Address.list( params ) ]
    }
```

```
def show = {
        def addressInstance = Address.get( params.id )

        if(!addressInstance) {
            flash.message = "Address not found with id ${params.id}"
            redirect(action:list)
        }
        else { return [ addressInstance : addressInstance ] }
    }

    def delete = {
        def addressInstance = Address.get( params.id )
        if(addressInstance) {
            addressInstance.delete()
            flash.message = "Address ${params.id} deleted"
            redirect(action:list)
        }
        else {
            flash.message = "Address not found with id ${params.id}"
            redirect(action:list)
        }
    }

    def edit = {
        def addressInstance = Address.get( params.id )

        if(!addressInstance) {
            flash.message = "Address not found with id ${params.id}"
            redirect(action:list)
        }
        else {
            return [ addressInstance : addressInstance ]
        }
    }

    def update = {
        def addressInstance = Address.get( params.id )
        if(addressInstance) {
            addressInstance.properties = params
            if(!addressInstance.hasErrors() && addressInstance.save()) {
                flash.message = "Address ${params.id} updated"
                redirect(action:show,id:addressInstance.id)
            }
            else {
                render(view:'edit',model:[addressInstance:addressInstance])
            }
        }
        else {
            flash.message = "Address not found with id ${params.id}"
            redirect(action:edit,id:params.id)
        }
    }

    def create = {
        def addressInstance = new Address()
        addressInstance.properties = params
        return ['addressInstance':addressInstance]
    }

    def save = {
        def addressInstance = new Address(params)
        if(!addressInstance.hasErrors() && addressInstance.save()) {
            flash.message = "Address ${addressInstance.id} created"
            redirect(action:show,id:addressInstance.id)
        }
        else {
            render(view:'create',model:[addressInstance:addressInstance])
        }
    }
}
```

9.3 RESTful Webpage

```
<html>
	<head>
		<title>Test RESTful service</title>

		<style type='text/css'>
			label {
				width: 100px;
				float: left;
				display: block;
			}

			br {
				clear: left;
			}
		</style>
		<script type='text/javascript'>
			function execute(type) {
				var firstname = document.getElementById('firstname').value;
				var surname = document.getElementById('surname').value;

				var obj = document.getElementById('chooseMethod');
				var method = obj.options[obj.selectedIndex].value;

				xmlhttp = new XMLHttpRequest();
				xmlhttp.onreadystatechange = function() {
					if (this.readyState == 4) {
						alert(xmlhttp.responseText);
					}
				}

				if (type == "root") {
					var uri = """../user/";
				} else if (type == "id") {
					var uri = """../user/1/";
				}

				xmlhttp.open(method, uri, true);
				xmlhttp.setRequestHeader('Content-Type', 'application/x-www-
form-urlencoded');
				xmlhttp.send("firstName=" + firstname + "&surName=" + surname);

				return false;
			}
		</script>
	</head>
	<body>
		<h1>REST tester</h1>
		This is a simple REST test, so we can do stuff.<br />
		Choose method (GET default):
		<select id='chooseMethod'>
			<option value='GET' selected='selected'>GET</option>
			<option value='PUT'>PUT</option>
			<option value='DELETE'>DELETE</option>
			<option value='POST'>POST</option>
		</select>
		<br /><br /><br />

		<label>Firstname:</label>
		<input type='text' name='firstName' id='firstname' value='${this.user.firstName}'
/><br style='clear: left' />

		<label>Surname:</label>
		<input type='text' name='surName' id='surname' value='${this.user.surName}' /><br
style='clear: left' /><br />

		<label> </label>
		<input type='button' name='submitButton' onclick='execute("root")' value='send
request to root' />
		<input type='button' name='submitButton' onclick='execute("id")' value='send
request to member 1' />

	</body>
</html>
```