Introduction to Groovy and Grails

Mohamed Seifeddine

November 6, 2009

Contents

1 Foreword

This document is written as my Masters' thesis and is based on a long individual learning period of Groovy and Grails through the developement of what later became `SoukLubnan.com`. A Website in its younger days that allows people in Lebanon to Buy & Sell products online.

The decision to do this on the relatively new Grails framework was not an easy one to make. But Groovy had been brought to my attention while studying abroad in North Carolina and the choice between going with Grails or one of the many other application frameworks soon became clear. It's a choice I'm happy I made and one I do not regret today. Both Groovy and Grails are here to stay for a long period of time and that's because they are both excellent on what they're intended for. I'm not the only one to share this belief either. In November 2008, G2One, the company behind Groovy and Grails was acquired by SpringSource and in August 2009, SpringSource was acquired by VMware for the sum of 420 million USD [19] and the success of Groovy and Grails are likely to be a continued priority for the new parties involved.

With little to no previous experience in Web development this would however prove to be a tough journey to make on your own. My toughest battle was against time. I had to finish the development of the Website to later write this document about Groovy and Grails. Of course, the optimistic time frame didn't go as planned, but I have learned and gained lots of experience during this period as well.

That's why this document is written with the *novice* reader in mind. A reader that like me when I started, knew little to nothing about Web development. I've pretty much attempted to write a getting started with Groovy and Grails document and my hope is that it'll prove to be a useful resource for people in my then situation.

It's not that there aren't plenty of books and material on Groovy and Grails available out there, but with most books being between 500-700 pages, it's likely for some potential readers to feel discouraged on start digging through all those pages. The books are great for providing an in depth coverage of these topics, but for those want a brief presentation on how to get started, a shorter compact version is likely to be a desirable resource as well.

The keyword for this document is on introduction. To get the reader familiar with key topics of Groovy and Grails. To provide a level of understanding of what Groovy and Grails is, what they offer, how they interact, what purpose they fill, what new they bring to the table and of course, how to use them.

The purpose of this document is not to be a complete coverage on Groovy and Grails although many of the described topics often are. The ones covered are primarly the ones that needs to be brought to the novice readers attention[1].

The document is divided into two chapters. One covering the programming language Groovy, the second covering the Web development framework Grails.

[1] Except for *some* of the topics described in Section 6

Sections often start of with a small presentation of the topic, followed by easy, often abstract examples that are normally commented on right after.

The experience required from the reader of this document is basic knowledge of the Java language, SQL and very basic HTML. This document do not attempt to teach the reader anything about HTML, XML, CSS or JavaScript. These are topics that you're hopefully already familiar with or topics you'll hopefully improve after reading this document and start working with Grails by developing your own Web application. Learning those topics then will surely come to you automatically.

Hopefully, this document will prove to be an easy step into the world of Groovy and Grails.

Many thanks for lots of the help must be awarded to the people on the Groovy and Grails mailing lists for answering questions and for enabling me to start getting hang of it all. Also thanks to the authors of all the Groovy and Grails books.

A special thanks to my instructor Roger Henriksson at Lund University for taking on my suggestion on a document about Groovy and Grails, for being patient, helpful and for trusting me on finally delivering a "complete" document.

2 Introduction to Groovy

Groovy is a relatively new dynamic language built to run natively on the powerful Java Virtual Machine. This is Groovy's best feature. This feature also enables Groovy to integrate seamlessly with Java and because Groovy was designed with the JVM in mind, there is little to no impedance mismatch [4].

Groovy compiles to native Java bytecode, with resulting *.class* files almost indistinguishable from what would have been produced by similar Java code. It's therefore subject to the same just-in-time byte code optimizations that are performed on Java byte code [2], making it comparatively fast, even for scripting tasks.

The Java platform has an undeniable market share, and corporate clients often run on a Java platform, such as Java Enterprise, allowing nothing but Java to be developed and deployed in production [15]. "The language itself has remained pretty much unchanged since the early days to help support backward compatibility" (Christopher M.Judd & al) [4]. Lots of baggage have piled up throughout all these years and Java has become complex and difficult.

Groovy was influenced and is often compared to other powerful scripting languages such as Python, Ruby and Perl as well as non scripting dynamic ones such as Smalltalk, Lisp or Dylan. Dierk König & al says, "the goal of Groovy is to provide language capabilities of comparable impact on the Java platform, while obeying the Java object model and keeping the perspective of a Java programmer" [15].

"Groovy and Java are like close cousins, and their syntax are very similar hence why Groovy is so easy to learn for Java developers" (Guillaume Laforge) [27]. The similiarities are so close that most Java programs are valid Groovy programs. Different from other dynamic languages that are ported to the JVM such as JRuby and Jython, Groovy was designed with Java in mind [4], "significantly reducing the learning curve so Java developers feel right at home with Groovy" (Christopher M.Judd & al). Neither JRuby or Jython show as much promise and potential as Groovy. Futhermore, using the Groovy language does not limit us to Groovy only. Parts can be written in Groovy and others in Java, not only by separation to different source files, but we can write and call Java code within a Groovy program because the integration between the two languages is complete. Groovy itself is actually written in a combination of Java and Groovy [20].

This seamless integration between the two languages allows us to harness "all the power of Java, including the massive set of available libraries" (Dierk König & al) [15]. Groovy does not only have access to the existing Java API, but provides its own Development Kit (GDK), which also extends the Java API by adding new methods to existing Java classes, making them *groovier*. Almost anything we can do with Java we can do with Groovy only most things are much easier and cleaner to do in Groovy.

Groovy is a dynamic language, often thought of as just a scripting language. But as pointed out already, it's more than a just scripting language. Describing the Groovy dynamic language, Venkat Subramaniam says that "dynamic lan-

guages have the ability to extend a program at runtime, including changing the structure of objects, types and behavior. Dynamic languages allows us to do things at runtime that static languages do at compile time" [1].

Java is often described as an object oriented language but as Bashar Abdul-Jawad points out "Java is not a purely object oriented language despite what some people might believe. It has primitive types, such as `int`, `long` and `double` that are not objects and have no reference semantics. Operators in Java, such as +, * and - can operate on primitive types only and not on objects (besides `+` on `String`)" [20].

In Groovy everything is an object and we can call methods on anything. Even on numbers. We can "pass blocks of code around, known as closures for immediate or later execution and we can easily augment existing libraries with our own specialized semantic" (Dierk König & al) [15] using the Meta-Object Protocol.

Bashar Abdul-Jawad continues, "Java has no language level support for collections, that is, no literal declaration for collections such as lists or maps, as it has for arrays" [20]. Many everyday things we want to do with Java, like recursively going through a directory might prove to be too much effort, even for an experienced Java professional. Groovy makes all this very simple.

The intention behind Groovy is not to replace the Java programming language. Instead, Groovy and Java should each contribute with its respective strengths to "smooth out some of the speed bumps that have historically slowed down the Java development process" (Scott Davis) [7]. The role of Groovy is to simplify tasks that are tedious in Java. "Where the Java programming language is exacting, Groovy is expedient. Where the Java programming language is extensive, Groovy is convenient" [8].

To summarize, Groovy is intended to coexist with Java and address the weak points of previous approaches. "Groovy brings the best of both worlds. A flexible, highly productive, agile and dynamic language that runs on the rich framework of the Java Platform" (Venkat Subramaniam) [1].

3 Getting started

In this section we'll try to cover how you can get started using the Groovy language.

3.1 Installing Groovy

Installing Groovy is easy.

3.1.1 JDK - A Prerequisite

A prequisite for using Groovy is having Java Development Kit (JDK) version 1.5 or higher installed. You can skip this part if you already have this installed.

1. Goto `http://java.sun.com/javase/downloads`. Look for *Java SE Development Kit (JDK)*. Download and install.
2. Create the environment variable `JAVA_HOME` and let it point to the JDK installation directory. For example
   c:/Program Files/Java/jdk1.6.0_13 on a Windows system.
3. On a Windows system: Add *%JAVA_HOME%/bin*
   On a Unix like system: Add *$JAVA_HOME/bin*
   to the `PATH` environment variable.
4. Open up a new command prompt or terminal and enter
   » java -version.
   If properly installed, you should see output similar to:

   ```
   java version "1.6.0_13"
   Java(TM) SE Runtime Environment (build 1.6.0_13-b03 )
   Java HotSpot(TM) Client VM (build 11.3-b02, mixed mode, ...)
   ```

3.1.2 Installing Groovy

This guide for installing Groovy is applicable for Windows and Unix like systems. There are installers available but this guide is based on the **binary release** version in **zip** format, which is platform independent and the recommended way to install Groovy.

1. Download the lastest stable **binary release** version in **zip** format of Groovy, currently 1.6.4 from `http://groovy.codehaus.org/Download`. We will *refer* to this version in the rest of this installation guide.
2. Extract the content from the downloaded zip file into a location of your choice. In *c:/* will do just fine on a Windows system. You should now have a folder *c:/groovy-1.6.4*.
3. Create the environment variable `GROOVY_HOME` and let it point to the extracted directory. For example *c:/groovy-1.6.4*.

4. On a Windows system: Add *%GROOVY_HOME%/bin*
   On a Unix like system: Add *$GROOVY_HOME/bin*
   to the `PATH` environment variable.
5. Open up a new command prompt or terminal and enter
   » groovy -version.
   If properly installed, you should see output similar to:

   ```
   Groovy Version: 1.6.4 JVM: 1.6.0_13
   ```

3.2 Updating Groovy

To update Groovy, download the latest stable binary release version in zip format of Groovy, extract and edit the `GROOVY_HOME` environment variable. For example to *c:/groovy-1.6.5*.

3.3 Editors for Groovy and Grails

There is Groovy support available for nearly every modern IDE and text editor. Eclipse, NetBeans, Intellij IDEA, jEdit, Oracle JDeveloper, XCode, TextPad, SubEthaEdit, Vim, Emacs and more.

Popular editors for working with *Groovy* and *Grails* are:

- Intellij IDEA (not free)
- NetBeans
- Eclipse
- TextMate (Mac only)

Visit `http://groovy.codehaus.org/IDE+Support` for more information on Groovy capable editors.
Visit `http://www.grails.org/IDE+Integration` for more information on Grails capable editors.

3.4 groovysh, groovyConsole & groovy

There are three commands we can use to execute Groovy scripts and Groovy code. They will be helpful and valuable for the learning experience and provides and easy way to test, analyze and understand Groovy better. It's recommended to use these tools, and especially the **groovyConsole** in an hands on approach to be able to fully understand the basics of the Groovy language.

» groovysh

This command will start the groovysh command line shell, which can be used to execute code interactivly 'on the fly'. Here is a simple example:

```
groovy:000> print 'Hello, World!'
hello, world===> null
groovy:000> System.out.print("Hello, World!");
hello, world===> null
```

Both these calls, first being the Groovy way, second the Java way will output the text to the console along with a return value. As a matter of convention, Groovy removes the need for you to explicitly type *return someObject* and always returns the results of what is placed at the end of methods and scripts. In this case, there is no result, so `null` is returned.

» groovyConsole

The Groovy console is a minimal graphical based editor written in Groovy that lets you load, create, write and run Groovy code instantly from within the editor. To run the code in the editor, press *Ctrl+R*. To execute only a small portion, highlight the code and press *Ctrl+Shift+R* or select *Script -> Run Selection* in the meny. To clear the output-window press *Ctrl+W*.

The following is a small Groovy sample written and run in the Groovy console:

```
0  class Person{
1      def firstName                          // Dynamic type
2      def lastName
3      def gender
4  }
5
6  def newP = new Person(firstName:'Eva')     // Declare and set
7  newP.setLastName('Svensson')               // Use of the setter
8  newP.gender = 'Female'                     // Without the setter
9
10 /* With and without the getter */
11 print "${newP.getFirstName()} $newP.lastName : $newP.gender"
```

Output:

```
Eva Svensson : Female
```

The setter (line 7) and getter (line 11) were automatically made available, similar to how a constructor is available in Java even when we don't write any. We can also use a shorter form without the setter (line 8) and getter (line 11).

Notice the use of what is called a `GString` (line 11). In Java we often have to use `+` to concatenate strings which can result in ugly, and less readable code.

Yet another thing to notice is the missing **semicolon** at the end of each line. In Groovy the semicolon is optional, but we must of course use a semicolon if there are several statements on the same line.

If we save the contents of the last script into a file *demo.groovy*, we can run

Groovy code in yet a way:

» **groovy** *demo.groovy*

Output:

```
Eva Svensson : Female
```

This is different from Java. In Groovy, we are executing the source code.

An ordinary Java class is generated for us in the background, and if we are running a script, Groovy will generate a class with a main method containing the script source [4], compile the script to finally execute it.

There is also a separate Groovy compiler that can be run like this:

» **groovyc** *demo.groovy*

It's similar to *javac* and should be used to avoid having to recompile the code each time the code is being run.

4 Boolean Evaluation, Elvis Operator & the Safe Navigation Operator

Groovy has many exciting little features that makes development less effort. Boolean evaluation, called *Groovy Truth* is different in Groovy than in Java. Java insist that you provide a boolean expression for the condition part in an `if` statement for example. Groovy is more dynamic and includes more expressive syntax. Depending on the context, Groovy will evaluate expressions such as `null`, empty string `""` and `0` to `false`.

Below we are looking at a couple of examples, while comparing them with Java to try to uncover some of the differences between them. Let's start with `String` evaluations:

```
String str = ...                                    // Unknown

/* Enter block if str has characters */
if( str ) {...}                                     // In Groovy
if( str != null && !str.isEmpty() ) {...}           // In Java
```

Here we'd like to safely operate on `str` inside the block, but only if `str` contains characters. Of course there are several ways to go about this[2] but even then, Groovy's approach is more readible and easy to understand.

We can see in Table 1 how *Groovy Truth* adds new flavors to boolean evaluation. For the last example, we see that if `str` is `null`, the expression will automatically evaluate to `false` and if not, evaluate to `true` only if its length is larger than zero.

Context of expression	*Condition for true*
Boolean	true
Collection	not empty
Character	value not 0
CharSequence	length > 0
Enumaration	has more elements
Iterator	hasNext
Number	double value not 0
Map	not empty
Matcher	at least one match
Object[]	length > 0
Any other type	not null

Table 1: Treatment of types for boolean evaluation [1]

[2] `But with no way to get around the null check. !("").equals(str) is true if str is null`

Example with numbers:

```
int number = ...            // Unknown

/* Enter block if number is not 0 */
if( number ) {...}          // In Groovy
if( number != 0 ) {...}     // In Java
```

Note that if the value of `number` had been `-1`, Groovy would still evaluate the condition to `true`. Only if the `double` value of a number equals `0.0`, will Groovy evaluate the condition to `false`.

Groovy Truth examples on collections:

```
List lst = []
Map mp = [:]
if(lst || mp)
    print 'Won't print'      // Because lst and mp are empty

lst += 'someValue'           // Add to lst
mp  += [a:'1', b:'2']        // Add to mp

if(lst && mp)
    print 'Will print'

if([1] && ['a'] && [0] && [0.0] && [false] && [null])
    print 'Will print'
```

The last condition contained several non-empty lists and therefore each and one of them evaluates to `true` no matter the content in them. Same rules applies for `Map`.

Groovy overrides Java's equality operator `==` on types such as `String`, `Integer`, `List` and `Map` to name a few, to match on content rather than object equality.

In fact, Groovy lets us override and overload any operator without implementing any specific interface by adding the corresponding method based operator to our class. There are a wide range of operators
`(+, -, ==, <=, <=>, !=, +=, -=, a[b], ...)` that can be overridden but isn't always straightforward to get right. For more information on this topic, please refer to [15].

Let's examine the following two examples, written to work in both Java and Groovy, to demonstrate the equality operator `==` :

```
/* Ex: 1 */
String str1 = new String("abc");
String str2 = new String("abc");
String str3 = str1;

if( str1 == str2 ) System.out.print("Will print in Groovy only");
if( str1 == str3 ) System.out.print("Will print in both");

/* Ex: 2 */
ArrayList<String> lst1 = new ArrayList<String>(); lst1.add("1");
ArrayList<String> lst2 = new ArrayList<String>(); lst2.add("1");
ArrayList<String> lst3 = lst1;

if( lst1 == lst2 ) System.out.print("Will print in Groovy only");
if( lst1 == lst3 ) System.out.print("Will print in both");
```

We can see that Groovy will evaluate all conditions to `true` while Java only those when the object reference are the same. This is becuase Groovy will look at the content when comparing both `String` and `ArrayList` as well as on several other objects.

To test on object equality, we can use Groovy's `is()` method instead:

```
str1.is(str2)
```

4.1 Elvis Operator

The Elvis Operator is a shortening of Java's ternary operator.

In the example below, we want to use a users `chatName` if it's set, otherwise set it to *Anonymous*:

```
String chatName =   user.chatName ?: 'Anonymous'      // In Groovy

String chatName =   user.chatName != null ?
                    user.chatName :  "Anonymous";     // In Java
```

4.2 Safe Navigation Operator

The Safe Navigation Operator (?.) lets us discretly check whether or not an object is `null` when accessing its methods and properties.

To be on the safe side, we often need to make sure that a `NullPointerException` is not thrown before performing the task we really want, which can result in less readable code. Take the following example:

```
if( user?.email?.isConfirmed() ) {...}          // In Groovy

if( user != null && user.email != null
   && user.email.isConfirmed() ) {...}          // In Java
```

By providing the Safe Navigation Operator we can call the objects internals without having to worry about the object being `null`. If the object is `null`, the same expression will return `null` which will evaluate to `false` when used in a condition. It's usage is not limited to conditions only, but can be used on a single line as well:

```
user?.doSomething()                             // In Groovy

if(user != null)
    user.doSomething()                          // In Java
```

This is an excellent and important feature, allowing us to call methods and use properties without the need of having clumsy `if` blocks everywhere.

However, this particular part of Groovy is a place we need to be careful with not to make simple mistakes. Take a look at these two conditions:

```
User user = null

if( !user?.isOnline() )         { print 'User is Offline' }

if( user?.isOnline() == false ) { print 'User is Offline' }
```

The two conditions does not evaluate to the same value. This typical rewrite, where a `false` condition is usually simplified can unfortantly cause our program to behave in a an unexpected way if we're using the Safe Navigation Operator and are not cautious.

Since the `?.` returns `null` if the object it's being used on is `null`, the first condition becomes `!null` which always evaluates to `true` and we enter the block. Our intention here was to enter only if there **was a user** that happened to be offline. The second condition will be interpreted as `null == false` and evaluate to `false`.

Scenarios like this can also occur when comparing two objects, using `?.` on one or both and where both happen to be `null`. It's a replacement for the majority of the `null` checks we'd normally do, but not for all.

As mentioned earlier, Groovy will evaluate expressions as boolean depending on the context they're being used in.

```
/* In Groovy */
boolean notEmpty(String str){
    str?.size()         // Just str is actually enough
}

/* In Java */
public boolean notEmpty(String str){
    return str != null && str.length() > 0 ;
}
```

In the Groovy code we are returning an `int` or `null` from `str?.size()` by placing the code at the bottom of the method, and notice, without having to explicity type `return` in front. Since the method return declaration is of type *boolean*, Groovy will automatically evaluate the `null` or `int` value according to Table 1 and return `true` or `false`.

Wondering over where `size()` came from? The inconsistency in Java by offering `length()` for objects such as `String`, `StringBuffer`, `CharBuffer`, and `length` without `()` for arrays, and `size()` for collections such as `ArrayList` can at times be unnecessary confusing. Since they all provide similar functionality, Groovy have made `size()` available to all these types, and to others as well.

5 String & GString

Groovy strings are instances of `java.lang.String` but are alot groovier. Groovy builds on Java `String` in many ways and recognizes that not all use of string literals are the same, so it offers a variety of options.

By supporting single and double quotes, triple single and double quotes, and forward slashing to declare strings we can reduce noice and avoid escaping.

Groovy also offers something called a `GString` (short for 'Groovy String' of course).

Let's take a look at a couple of examples to get an idea of what all this means:

```
/* Double quote */
String str = "Hello, World!"

/* Single quote */
'Hello, World!' == str                  // True

/* Triple single quotes */
'''Hello, World!''' == str              // True

/* Triple double quote */
"""Hello, World!""" == str              // True

/* Forward slash */
str == /Hello, World!/                  // True
```

These are all different ways of writing a `java.util.String` in Groovy. But each one has its own special usage area.

For example, if we intend to use a double quote (") in we text, we should use the single quotes as a wrapper and vice versa to avoid having to escape, i.e. `'Hello, "Moe"'` vs `"Hello, \"Moe\""` in Java.

Triple single and double quotes are normally used when a `String` spans over multiple lines:

```
String str1 = """
SELECT * FROM Ad
WHERE status >= 2
"""

String str2 = "\nSELECT * FROM Ad\nWHERE status >= 2\n"
str1 == str2   // True
```

The triple single quotes would have yielded the same `String`. Notice how the newlines where included in the `str1` as well. To omit the newlines, we can simply put a backslash (\) at the end of each line.

```
String str1 = """\
SELECT * FROM Ad \
WHERE status >= 2\
"""

String str2 = "SELECT * FROM Ad WHERE status >= 2"
str1 == str2   // True
```

A `GString` is of instance `groovy.lang.GString` and represents a string that contains embedded values or placeholders:

```
String str = "World!"
print "Hello, ${str}"     // Hello, World!
print 'Hello, ${str}'     // Hello, ${str}

/* Multiline GString */
int status = 2
String str = """
SELECT * FROM Ad
WHERE status >= ${status}
"""
```

Whenever a *non-single* quoted `String` contains an unescaped dollar sign `$`, it is interpreted as a `GString` instead of a plain `String`.

There is also the special purpose forwardslash (/) to declare strings, which can be of good use when dealing with paths and regular expressions. They eliminate the constant need for having to escape the backslash (\) in patterns and for special characters. Example:

```
/* Path */
"c:\\windows" == /c:\windows/           // True

/* Regular expression */
"\\d+-\\w+-\\d+" ==  /\d+-\w+-\d+/      // True
"2009-Jan-01"    ==~ /\d+-\w+-\d+/      // True (Match Operator)
```

Sometimes the forwardslash syntax interferes with other valid Groovy expressions such as line comments, numerical expressions with multiple slashes for division and some other less common areas and might not work as expected. So it's recommended to use the slashy syntax for their intended usage area, paths and regular expressions.

Groovy adds several convenience methods to the String class through the GDK such as `isInteger()`, `toInteger()`, `toDouble()`, `minus()`, `reverse()`, `each()`, `eachLine()`, `center()`, `execute()`[3] to name a few. A complete list on these methods can be found on `http://groovy.codehaus.org/groovy-jdk/`.

[3]Enables us to run platform dependent shell commands on a String.
Try 'ls -l'.execute().text on a Unix system to get the listing of the current directory

6 Classes, Dynamic type, Methods, Closures the & Meta-Object Protocol

Groovy is a fully fledged object oriented language with all of the OO programming concepts that are familiar to Java developers such as classes, objects, interfaces, inheritance, and etc. It's a *pure* OO language unlike Java which mixes both primitive and reference types.

In Groovy, when a primitive type gets passed into the Groovy world it's automatically *boxed* into its object equivalent and vice versa (see Figure 1).

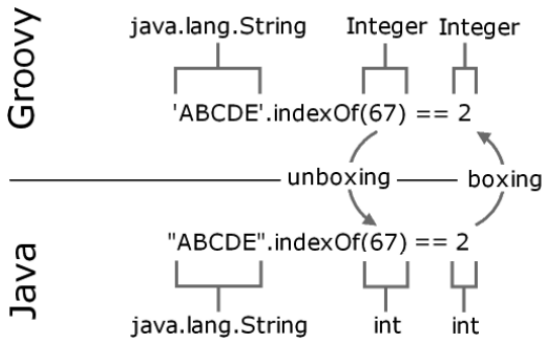

Figure 1: Autoboxing in example [15]

This allows Groovy to support some interesting concepts, such as being able to call methods on numbers. The following call `2.upto(4){print it}` will for example print `234`.

Class definition in Groovy is almost identical to Java. Classes are declared using the `class` keyword and may contain properties, constructors, methods and etc.

In Java, we can declare a property that by default is made *package-private*[4] unless specified otherwise. When it comes to classes and methods, a modifier needs however to be specified.

Groovy makes no such distinction. `public`, `private` and `protected` are all available for use, but Groovy will by default use `public` if not explicitly specified otherwise, for methods and classes as well. Getters and setters are automatically available for declared properties, unless a modifier have been specified for that property (including `public`).

Groovy allows us to use scripts as already seen in Section 3. "Any code that's not inside a class is called a script" (Bashar Abdul-Jawad) [20]. A Groovy file can have one or more classes which do not need to match the name of the containing file. The file can in addition to classes contain scripting code.

Common classes and packages are already imported. For example, the `Calendar` class already refers to `java.util.Calendar`. Other Java packages and classes

[4] Accessible to other classes in the same package

that are imported by default are `java.lang`, `java.util`, `java.io`, `java.net`, `java.math.BigDecimal` and `java.math.BigInteger`. In addition, the Groovy packages `groovy.lang` and `groovy.util` are imported.

The `as` keyword can in Groovy be used to cast objects from one type to another, aswell as type aliasing when used in an `import` statement. Type aliasing allows us to refer to a class by a name of our choosing. This feature can for example be used to resolve class naming conflicts.

Groovy also supports dynamic typing, closures, expandos[5] and the Meta-Object protocol which we'll try to introduce in this section.

On the next page follows a Groovy file that uses many of the above mentioned features.

Read it line by line, starting at the top. Pay attention to the comments and the output from the script:

[5]Expandos are not covered in this document

Groovy File - DemoClasses.groovy

```
import java.util.Date
import java.sql.Date as SQLDate      // Type alias

/* First class */
class DemoOne {                      // public not necessary
    String firstName                 // Has a setter and getter
    public String lastName           // No setter and getter

    /* Constructor. Type for methods not a requirement */
    DemoOne(firstName, lastName){
        this.firstName = firstName
        this.lastName = lastName
    }

    String getFullName(){            // public not necessary
         /* Returns what's on the last line */
         lastName + ", " + firstName
    }
}


/* Second class */
class DemoTwo {
    int number

    static String demoTypeAlias(){
        /* Using the type alias SQLDate */
        SQLDate sqlDate = new SQLDate(0)
        Date utilDate = new Date()

        "SQLDate:\t\t$sqlDate\nutilDate:\t\t$utilDate"
    }

    /* Calling methods on numbers */
    void demoNumber(){
            print '2.upto(4):\t\t'
        2.upto(4) { print it }           // Use of closure
            print '\n4.downto(2):\t'
        4.downto(2) { print it }
            print '\n2.step(8,2):\t'
        2.step(8,2) { print it }
            print '\n3.times:\t\t'
        3.times { print "Hello_$it! " }
    }
}
```

```
/* Script Starts */
println '---DemoOne---'

DemoOne demoOne = new DemoOne('Mikkey', 'Mouse') // Constructor
demoOne.setFirstName('Mickey')            // Setter

println demoOne.getFirstName()            // Getter
try {
    println demoOne.getLastName()
    println demoOne.setLastName('Will not work')
}catch(e){ println 'get & set not avail. Declared public' }
println demoOne.lastName                  // Acessing property
println demoOne.getFullName()             // Method call

println '---DemoTwo---'

DemoTwo demoTwo1 = new DemoTwo(number:1)  // Maps by map
DemoTwo demoTwo2 = new DemoTwo()
demoTwo2.number  = 2                      // Setting property

println demoTwo1.getNumber().class        // Integer
println demoTwo2.number                   // Accessing property
println demoTwo2.number as double         // as keyword casting
println ( (double) demoTwo2.number )      // Java style casting

println DemoTwo.demoTypeAlias()           // Call static method
demoTwo1.demoNumber()                     // Call number method
```

Output:

```
---DemoOne---
Mickey
Mouse
Mouse, Mickey
get & set not avail. Declared public
---DemoTwo---
class java.lang.Integer
2
2.0
2.0
SQLDate:        1970-01-01
utilDate:       Thu Jul 16 11:43:18 CEST 2009
2.upto(4):      234
4.downto(2):    432
2.step(8,2):    246
3.times:        Hello_0! Hello_1! Hello_2!
```

6.1 Dynamic type

Groovy offers, just like Java a way of assigning a type to a variable when declaring it and we've seen many examples of this already. The type assigned will be used for the variable during its lifetime and can't be changed. This is better known as static typing.

With Groovy we are not forced to define the type of a variable if we don't want to. A variable can be typed or untyped. The `def` keyword is used to express that the variable is untyped and that no particular type is demanded. Under the covers, it's "given the static type `Object` and a dynamic type depending on the value assigned to it" (Bashar Abdul-Jawad) [20].

Dierk König & al emphasise that "it's important to understand that regardless of whether a variable's type is explicitly declared, the system is type safe. Unlike untyped languages, Groovy doesn't allow you to treat an object of one type as an instance of a different without a well defined conversion being made" [15]. For example, `def str = '10'` can't be treated as an `Integer` such as being passed to a method that expects an `Integer`. A type is always assigned, it's just that we let Groovy determine it for us. In this case `str` is of type `String`[6].

Objects in dynamically typed languages don't have to satisfy the contract on declaration [1]. We may change the class type on a variable several times. They simply have to respond correctly to property and method calls at runtime [1].

Dynamic typing can be useful when we're not interested in finding out what the actual type a method returns is. We just know that we need the returned result. To pass it along to some other method, add it to a list or whatever. "We are spared the work of looking them up, declaring the type, and importing the package" [15].

Dynamic typing is not only convient for the 'lazy' programmer but also useful for **duck typing**. The naming comes from *If it walks like a duck, talks like a duck, then it's probably a duck.*

Duck typing is a style of dynamic typing in which an object's current set of properties and methods determines the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface [13].

Take a look at the following example, where duck typing is used instead of an interface:

[6] Try in groovyConsole: def str = '10'; str.class == String

```
class Duck {

    def Iam(){
        'Duck'
    }

    def goTo(def from, def to) {
        "I will walk, swim and fly from $from to $to"
    }
}
class Frog {

    String Iam(){
        'Frog'
    }

    def goTo(from, to) {
        "I will jump from $from to $to"
    }
}

def animals = [new Duck(), new Frog()]      // A list of animals
for (a in animals)                          // Iteration in a for
     doToday(a, 'Göteborg', 'Lund')

void doToday(def a, def from, def to){
    println "${ a.Iam() }: ${ a.goTo(from, to) }"
}
```

Output:

```
Duck: I will walk, swim and fly from Göteborg to Lund
Frog: I will jump from Göteborg to Lund
```

Notice that we can define parameter and return types as `def` for methods. We can also omit `def` when declaring incoming parameters and just declare the variable name on methods (seen on `Frog`'s `goTo()` method). The `doToday()` method can be used by **any** class that implements the methods `Iam()` and `goTo()`.

Interfaces have their own share of problems. Changes to an interface are breaking changes for all classes implementing it. Abstract classes and methods are a way around this but when using Groovy, we are not obligated to stick with any style.

As Dierk König & al summarizes it, "the *choice* between static and dynamic typing is one of the key benefits of Groovy" [15].

Dynamic duck typing allows us to add and remove methods, but if we use a method on a `def` declared `Duck` that's only available in `Frog`, we would not get a compilation error but a runtime one.

So this does put some responsibility on the developer. For small projects this

should rarely be a problem, and for larger ones, the program should be backed up by proper test cases anyway. We can in many situations minimize the amount of code we have to write and if we really do need, the full use of interfaces are available [15].

Dierk König & says that "the Web is full of heated discussions of whether static or dynamic typing is better. There are good arguments for either position. Static typing provides more information for optimization, more sanity checks at compile time, and better IDE support. It also reveals additional information about the meaning of variables or method parameters and allows method overloading" [15].

6.2 Closure

We've mentioned closures a couple of times already and we saw an example of their use earlier in *DemoClasses.groovy* on p.22. Perhaps it went unnoticed but closures are one of the most important and most useful features that Groovy brings to the Java platform and in Groovy, their usage is seen across the entire language.

The concept of closures is not a new one and they are similar to Java's inner classes but with less restrictions. More often closures are associated with functional languages by allowing us to execute a passed block of code that's specified elsewhere. This might sound a bit like calling a method, but they are different from methods. They don't need to be declared inside classes but can be declared anywhere. They act like methods in that they can take parameters (which can be closures themselves) and return values, but they are normal objects in that they can be returned, treated, assigned to variables and passed around like any other object.

A Groovy closure is code wrapped up as an object of type `groovy.lang.Closure`, defined and recognized by curly braces `{ // Code here }`. It's executed only when it's called and not when it's defined.

6.2.1 Create

```
/* With no parameter */
Closure simpleCloj1 = {
    println 'Hello, World!'
}

/* With 1 undefined parameter */
def simpleCloj2 = { obj ->
    println "Hello, $obj!"
}

/* With 1 defined parameter of type String */
def simpleCloj3 = { String obj ->
    println "Hello, $obj!"
}

/* If we are only passing one parameter, the argument can be
   omitted and we can use the keyword it for access */
def simpleCloj4 = {
    println "Hello, $it!"
}

/* Takes multiple parameters */
def twoParamsCloj = { obj1, obj2 ->
    println "$obj1, $obj2!"
}
```

6.2.2 Call

We can all a closure in three ways, `closure()`, `closure.call()` or `closure.doCall()`:

```
simpleCloj1()

simpleCloj2.call('World')

simpleCloj4.doCall('World')

twoParamsCloj('Hello', 'World')
```

Output:

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

As seen here, a closure can be called like a method, although sometimes we might use the other forms for clarity that it's a closure being called or if we need to use the Safe Navigation Operator.

Ok, that wasn't all that cool. The above wasn't very convincing of the powerfulness of closures. We could have accomplished the above with a simple method as well. But let's take a look at the following example written to work in Java, where we've created a method that reads from a file and then takes action on its content:

Java File - DemoFile.java

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  public class DemoFile {
4
5      public static void main(String args) {
6          readFileAndTakeAction('c:/text.txt', null);
7      }
8
9      /* Second parameter not used right now */
10     public static void readFileAndTakeAction(String fileName,
11                                              Object cloj){
12       BufferedReader bfIn = null;
13       try {
14           bfIn = new BufferedReader(new FileReader(fileName));
15
16           /* The actual reading & action */
17           String line = null;
18           while ((line = bfIn.readLine()) != null) {
19             System.out.println(line);
20           }
21
22           bfIn.close();
23       }
24       catch (Throwable e) {
25           e.printStackTrace();
26       }
27     }
28 }
```

On line 6 we call the method `readFileAndTakeAction()` which at this moment uses only the `fileName` passed to it. The method then reads from the file *text.txt* and prints out its content to the console on line 19.

If we now want to read from a file and do some other work on the content, we'd probably have to copy and paste the method[7], put a different name on it and change the content between the `while` loop on line 19. Unfortunately, we've just introduced redundancy to our code and changes are now harder to introduce and maintain. For example, what if we change our mind on how an exception needs to be handled?

With Groovy we don't have to work that hard. Closures allows reusability because we can pass a block of code (closure) to the method and execute it on line 19 instead. So let's do that:

We rename the file *DemoFile.java* to *DemoFile.groovy*, and change line 19 to:

```
19 cloj.call(line)
```

[7]Possibly created a switch statement that takes a String to determine what action to take. But that's just me.

`cloj` is the closure we are going to pass in the second parameter, and `line` is the variable being read from the file in the `while` loop. We can now from the main method call `readFileAndTakeAction()` and pass a closure to it in the second parameter:

```
def fileName = 'c:/text.txt'              // Not empty

/* Number of chars on every line */
readFileAndTakeAction(fileName, { print it.size() + ' ' } )

/* Number of total lines */
int i=0
def cloj = { i++ }
readFileAndTakeAction(fileName, cloj)
print i + ' '

/* Place the lines in a List */
List lst = []
readFileAndTakeAction(fileName, { myLine -> lst.add(myLine) } )
print lst.size()
```

Output:

```
15 15 13 11 4 4
```

We can see from the output that there are 4 lines in *text.txt* and that each line have 15, 15, 13 and 11 characters. We could continue give examples on how we could reuse that same method with different closures, but this seems unnecessary.

In the first example, we passed a closure directly to the method, and in it we used the `it` keyword refering to the passed variable *`line`* in `cloj.call(`*`line`*`)`. In the third example we named the same parameter `myLine`. In the second example we first assigned the `Closure` to a variable and then passed it on to the method.

Notice how we on the second and third example had access to and could manipulate the variable `i` and `lst` from within the passed closure. If a closure is defined inside a method, the closure will get access to all the variables that the containing method can access itself such as local variables, method arguments, class members and other methods [20].

The GDK is packed with tons of methods that makes life much easier thanks to closures. In the `File` class alone there are 28 methods that makes use of closures. And when it comes to our file example, similar functionality is of course already among those methods, allowing us to do the above 28 lines in 1 line:

```
new File('c://text.txt').eachLine { println it }
```

`eachLine()` is here a single parameter method in the `File` class that takes a closure, and since we stumbled in on it, we can also mention that **excep-**

tion handling is optional in Groovy. If the file doesn't exist, we'll get a `FileNotFoundException` which have to be caught in a `try`/`catch` block, but if we are certain the file does exist, we don't have to.

6.2.3 Getting Information

The `Closure` class makes available a few useful methods. For example, how many parameters a closure takes and what their types are, if declared.

```
def cloj = { int a, double b -> return (a + b) as double }

cloj.getMaximumNumberOfParameters() == 2         // True
println cloj.getParameterTypes()
```

Output:

```
[int, double]
```

6.2.4 Method Reference Pointer

The many similiarites between closures and methods are obvious and Groovy recognizes this and allows us to reuse methods as closures but this is limited to instance methods only. Below is an example:

```
class A {
    int num

    int sumWithNum(int a){
        num += a
    }
}
class B {
    int callCloj(Closure cloj, int a){
        cloj.call(a)
    }
}
A a  = new A(num:0)                          // Create an instance
B b  = new B()

/* sumWithNum() called from class B */
int returnedSum = b.callCloj( a.&sumWithNum, 90)    // Notice .&

println "Num from a  : $a.num"
println "Returned sum: $returnedSum"
```

Output:

```
Num in a    : 90
Returned sum: 90
```

In the above example, we passed a reference to `A`'s `sumWithNum()` method to `B`'s `callCloj()`. Notice that since we passed the method reference from the instance `a`, `a`'s property `num` will also be altered because the method `sumWithNum()` alters it.

In Section 7 on p.44 we'll see more powerful use cases of closures.

6.3 More on Methods

We've already covered much on how methods can be used in Groovy. Somethings we've used and pointed out already, other's been left out for this particular section.

6.3.1 Optional Parantheses

We can for cleaner syntax call methods that take one or more parameter by omitting the surronding parantheses `()`. For most part we've used the syntax with the parantheses, but on `print`, `println` and not long ago on `File`'s `eachLine` method we've omitted `()`. These are methods just like any other method:

```
println('Hello, World!')               // With parantheses

void method(def a, def b, def c){
    print(a); print ' '; print(b + ' '); print c + '\n'
}
method 'a', 'b', "c"                   // Without parantheses
method('d', 'e', "f")
method 1, 2, 3
```

Output:

```
Hello, World!
a b c
d e f
1 2 3
```

6.3.2 Positional Parameters

Parameters in a method can be defined with a default value, to be used when a call to that method is done with less parameters than it can take.

Groovy will look and match the parameters passed from left to right and if there are less incoming parameters than can be passed, the default supplied values will be used for those missing:

```
int method(a, b=100, c=1000){
    a+b+c
}
method 1, 2, 3 == 1+2+3                 // True
method(1, 2)   == 1+2+1000              // True
method(1)      == 1+100+1000            // True
```

6.3.3 Optional Parameters

We can define a method to take a dynamic amount of parameters:

```
void passed(a, Object[] optionals){
    print 'Passed in parameters: ' + a
    for(o in optionals)
        print ', ' + o
}
passed(1, 2, 3, 4, 5, "Hello!", 6, 7, 8, 9, 10)
```

Output:

```
Passed in parameters: 1, 2, 3, 4, 5, Hello!, 6, 7, 8, 9, 10
```

```
int sum(a, Object[] optionals){
    for(o in optionals)
        a+=o
    a
}
sum(1)            == 1                // True
sum(20, 1)        == 21               // True
sum 100,10,1      == 111              // True
sum 100,10,10,10  == 130              // True
```

If we want to call a method with a `List`, we can use the **spread operator** `*` which will distribute all items in a list onto the parameters of the method.

```
class C {
    int sum(a, Object[] optionals){
        for(o in optionals)
            a+=o
        a
    }

    /* Takes fixed amount of parameters */
    int mod(a, b){
        a%b
    }

    int doSomething(Closure cloj, Object[] optionals){
        // Do stuff here we don't want to repeat
        cloj.call(*optionals)          // Use of spread operator
    }
}

C c = new C()

println c.doSomething(c.&sum, 3, 3, 3, 3)
println c.doSomething(c.&mod, 10, 2)
```

Output:

```
12
0
```

The method `doSomething()` is now more dynamic and can pass on calls to any predefined method or closure. The spread operator can be used for other purposes as well.

6.3.4 Mapped Parameters

A method can take named parameters as a map similar to how the Groovy constructor can do. We define a method and call it:

```
/* Prints either firstName, lastName or both */
void printName(Map args){
    if( args.toPrint == 'lastName' )
        println args.lastName
    else if( args.toPrint == 'firstName' )
        println args.firstName
    else if( args.toPrint == 'both' )
        println args.firstName + ' ' + args.lastName;
}
printName toPrint:'lastName', lastName:'Bond', firstName:'James'
printName(toPrint:'both'    , lastName:'Bond', firstName:'James')
```

Output:

```
Bond
James Bond
```

6.3.5 Dynamic Method Call

We can call a method using a `String` as an identifier:

```
class D {
    int numberVariable = 10

    void methodOne(){
        println 'One'
    }
    void methodTwo(){
        println 'Two'
    }
}

void caller(def obj, def methodName){
    obj."$methodName"()              // Use of GString
}
```

```
caller(new D(), 'methodOne')
caller(new D(), 'methodTwo')
```

Output:

```
One
Two
```

This could be yet a way to dynamically call a different action in our file example earlier.

A property can be accessed in the same way, but this is usually done with the subscript operator instead:

```
def str = "numberVariable"
new D()."$str" == 10                // True
new D()[str]   == 10                // True (Subscript operator)
```

6.4 Meta-Object Protocol

Dierk König & al believes that, "in order to fully leverage the power of Groovy, it's beneficial to have a general understanding of how it works inside" [15] and performs its magic. "Meta-programming means writing programs that manipulate programs, including itself" (Venkat Subramaniam) [1]. The Meta-Object Protocol (MOP) defined in the interface `groovy.lang.MetaObjectProtocol` enables Groovy to dynamically change the behavior of classes and objects at runtime.

Classes written in Groovy `extends java.lang.Object` by default as well as implement the `groovy.lang.GroovyObject` interface:

```
public interface GroovyObject {
  Object     invokeMethod(String name, Object args);
  Object     getProperty(String property);
  void       setProperty(String property, Object newValue);
  MetaClass  getMetaClass();
  void       setMetaClass(MetaClass metaClass);
}
```

Every class, whether it's a POJO (Plain Old Java Object) or POGO also have an association to a `MetaClass` in Groovy. A `MetaClass` within Groovy defines the behaviour of any Groovy or Java class and provides all information about a class, such as its properties and methods. It can be used to invoke any method on a class but also used to add or borrow methods and properties, add or override constructors and more. The `MetaClass` also comes into 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, calls can be intercepted and routed to the right `MetaClass`. Figure 2 on p.43 at the end of this section illustrates how calls in Groovy are intercepted and handled.

6.4.1 Adding Methods & Properties

Adding methods[8] and properties to any class is simple.

```
/* Add method sayHello() to String class */
String.metaClass.sayHello = { lang ->
    if(lang == 'English') println 'Hello'
    else
    if(lang == 'Swedish') println 'Hej'
}

/* Add method isOdd() for class Integer */
Integer.metaClass.isOdd = {
    /* Delegate refers here to the caller Object */
    (delegate % 2) as Boolean
}
```

[8] What we meant to say was closures

```
/* We can dynamically add methods to our own classes as well */
class Person{
    String fName, lName
}
Person.metaClass.getFullName = { lName + ", " + fName }

'We can call say hello on any String now'.sayHello('Swedish')
println 3.isOdd()
println new Person(fName:'James', lName:'Bond').getFullName()
```

Output:

```
Hej
true
Bond, James
```

If we wish to add a method for all objects, we can simply add it to the `MetaClass` of `Object`. The method `isOdd()` that was added on `Integer` could be defined as a `Closure` in one place first and added to its cousins `Short` and `Long` as well.

Adding a **property** to any class is no different:

```
Integer.metaClass.numberOne = 1
99.numberOne == 1                // True
```

Note that we can expand and build not only on the entire class but on **standalone instances** with methods and properties as well:

```
String a = 'String instance a'
String b = 'String instance b'

/* Add property demoProperty on instance a */
a.metaClass.demoProperty = 'value'

a.demoProperty == 'value'        // True
b.demoProperty                   // MissingPropertyException
```

To add a `static` method to class `Integer`, we can do:

```
Integer.metaClass.static.isEven = { val -> !val.isOdd() }
Integer.isEven(5)  == false      // True
```

Notice that we where passing the value 5 in the method call, where we then could use it to call the instance method `isOdd()` defined earlier.

6.4.2 Add Constructor

To add a new constructor that takes a `Date` to the class `String` we can use the leftshift operator `<<`. To add or **override** a constructor we can use `=`. Overriding an already existing constructor using `<<` will result in an error.

```
String.metaClass.constructor << { Date date ->
    /* Get the constructor that takes String */
    def constructor = String.getConstructor(String)
    return constructor.newInstance( date.toString() )
}
String str = new String( new Date() )
println str
```

Output:

```
Thu Sep 03 19:10:29 CEST 2009
```

Notice how we can fetch a constructor using *reflection*. Of course in this case, returning `date.toString()` would have been enough.

6.4.3 Intercepting Method Calls

There are a number of ways we can intercept and act on method calls for any class. Intercepting is valuable when we want to change the behaviour of a method call, for example by routing to a new implementation, or perhaps to fix a bug in a method by checking its input first.

One way is to implement the `GroovyInterceptable` interface, which is a marker interface used to notify that all methods should be intercepted through the `invokeMethod()` mechanism of `GroovyObject`. We can then implement the method `invokeMethod()` which will hijack all method calls to the class, in which we can have some logic and act. This approach however, won't work if we are not the authors of the class and don't have the priviligies to modify it. We might also decide at runtime to start intercepting calls based on some condition or application state.

Another way is to use the `MetaClass` available in all classes. To intercept calls on a class we implement the method `invokeMethod()` through the `metaClass` object.

Say that we've decided that `String`'s `toString()` method must always return in *lowercase*. We can either do a complete re-implementation of the `toString()` method, or we can intercept the calls to it and make sure we call `toLowerCase()`:

```
String.metaClass.invokeMethod = { methodName, methodArgs ->
    if( methodName == 'toString' ){
        return delegate.toLowerCase()
    }
    else {
      /* Retrieve method for the given methodName and arguments */
      MetaMethod otherMethod =
         String.metaClass.getMetaMethod(methodName, methodArgs)

      /* Method exists in the String class. Call it. */
      if( otherMethod )
         return otherMethod.invoke(delegate, methodArgs)
      else
         return String.metaClass.invokeMissingMethod(delegate,
                                        methodName, methodArgs)
    }
}
```

We call `toString()`:

```
print 'AbC'.toString()
'XYZ'.methodDontExist()        // MissingMethodException
```

Output:

```
abc
```

When any method is being invoked now in the `String` class it will go through the logic defined in `invokeMethod()`. That is why it's important to handle and route other method calls as well, as done in the first `else` statement. There we attempt to fetch the called method using the name of the method and the supplied arguments. The arguments are necessary to identify the right method due to the possibility of the method being overloaded, such as `String`'s 9 different `valueOf()` methods.

Note that had the `toLowerCase()` method internally called `String`'s `toString()` we'd probably end up with an infinite loop between the `invokeMethod()` and `toLowerCase()` methods, to finally get a `java.lang.StackOverflowError`.

6.4.4 Getting Information

Discovering The Class

Every object in Java has a `getClass()` method. In Groovy, we can shorten the call to `class` and we've used this syntax several times already.

```
def str = '1'
println str.class
```

Output:

```
java.lang.String
```

Once we have the class we can ask it all sorts of questions.

Discovering Properties

```
class Person { String fName, lName }
```

On the **class**:

```
Person.properties.each { println it }
```

Output:

```
constructors=[Ljava.lang.reflect.Constructor;@142022d
superclass=class java.lang.Object
interface=false
primitive=false
...
```

On a `Person` **instance**:

```
new Person(fName:'James').properties.each { println it }
```

Output:

```
fName=James
class=class Person
lName=null
metaClass=org.codehaus.groovy.runtime.HandleMetaClass@8e2fb5...
```

Notice how the `metaClass` property have been added.

Discovering Methods

```
String.methods.each { println it }
```

Output:

```
public int java.lang.String.compareTo(java.lang.String)
public int java.lang.String.indexOf(java.lang.String,int)
public boolean java.lang.String.endsWith(java.lang.String)
...
```

Discovering Constructors

```
String.constructors.each { println it }
```

Output:

```
public java.lang.String()
public java.lang.String(java.lang.String)
public java.lang.String(char)
...
```

Discovering Interfaces

We can also get the interfaces a class implements:

```
String.interfaces.each { println it }
```

Output:

```
interface java.io.Serializable
interface java.lang.Comparable
interface java.lang.CharSequence
```

There are many more methods we can use to work with Groovy's MOP such as `hasProperty()`, `respondsTo()`, `setMetaProperty`, `invokeStaticMethod`, `isModified()` and more.

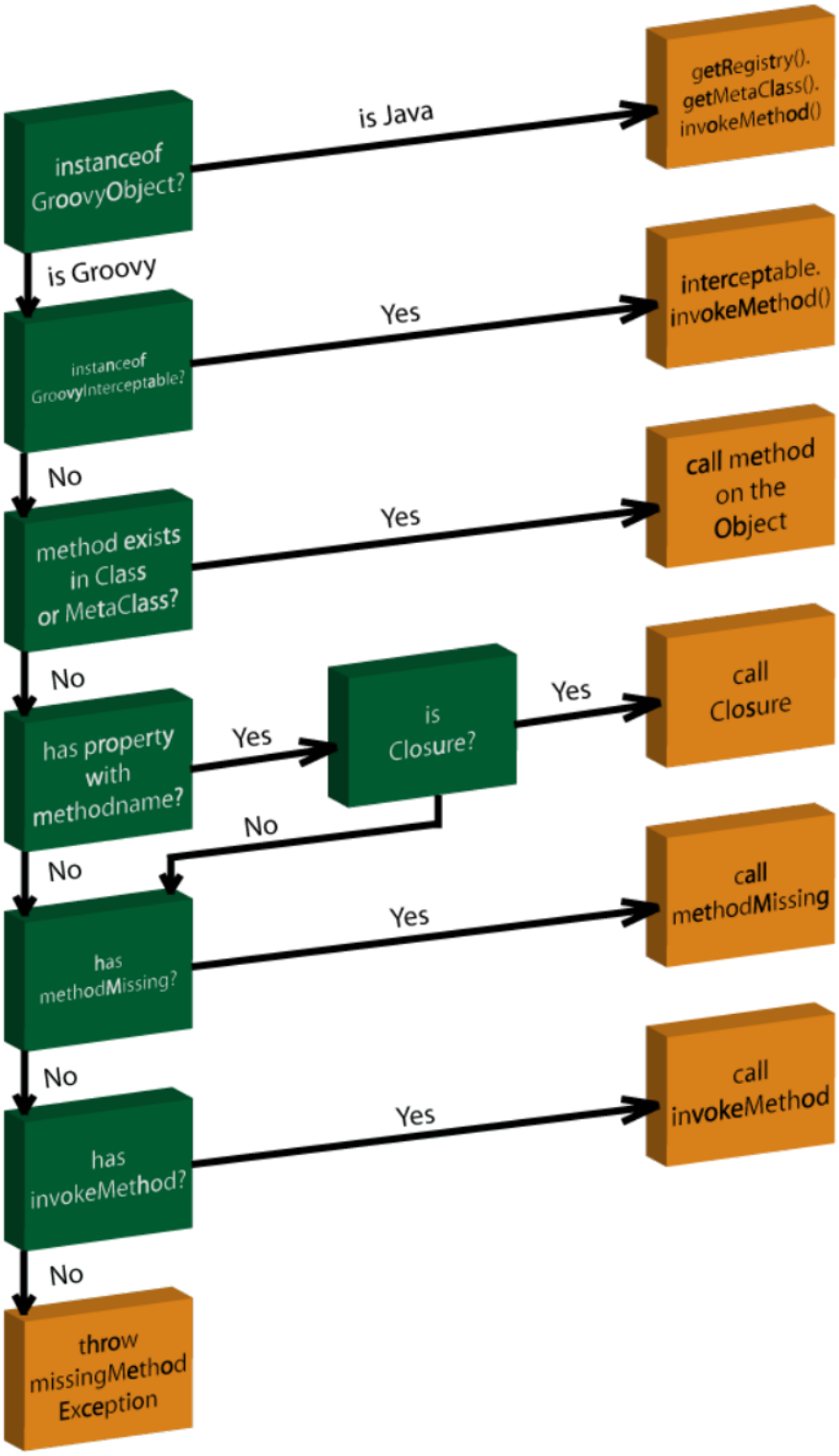

Figure 2: Working with the MOP [9]

7 Collections (List, Range, Map) and Iterative Object Methods

Working with collections in Groovy is much easier and more fun than it's in Java. It's easier to create, add, iterate and manipulate them.

Venkat Subramaniam says, "Groovy takes the already powerful Java collections and makes their API simpler and easier to use". Several convenience methods have been added, many effectively using closures as parameters [1].

With `Range` we can create a list of values on the fly, which can then be used or passed around as a `List` because `Range` extends `java.util.List`.

7.1 List

A `List` in Groovy is actually a `java.util.ArrayList`. It's however easier to declare and use:

```
/* Create a new List */
List lst = ['a', 'b', 'c']

lst.each { println it }
```

Output:

```
a
b
c
```

First we created a list with some initial values. Then we iterated over them with the iteration method `each()` and applied `println it` by passing code as a closure. All possible thanks to closures.

`lst` was on creation populated with values of type `String` but a `List` can have elements of any object type. It's not bound and can also be of mixed types.

There are a number of ways we can add, remove and modify the content of a `List`.

Add

```
1  def lst = []                    // Declared empty list
2
3  /* Add */
4  lst += 'c'                      // lst += ['c'] does the same
5  lst += ['d', 'e']
6  lst.addAll(0, ['a','b'])        // Add first
7
8  lst == ['a','b','c','d','e']    // True
9
10 lst.eachWithIndex { e, i -> println "${i}: ${e}" }
```

Output:

```
0: a
1: b
2: c
3: d
4: e
```

Notice how the iteration method `eachWithIndex()` takes a closure which in turn requires passing of two parameters. One for the element and one for the index. On line 5 it appears as of we attempted to add a `List` to another `List` but what happened whas that only the elements of the other was added.

If we instead wish to insert a `List` into another `List` we can use the shift operator `<<` or Java's `add()` method:

```
List lst= []
lst << ['b', 'c']
lst.add(0, ['a'])                          // Add first
lst += ['d', 'e']

lst == [['a'], ['b', 'c'], 'd', 'e']       // True

lst.flatten().eachWithIndex { e, i -> println "${i}: ${e}" }
```

Output:

```
0: a
1: b
2: c
3: d
4: e
```

Notice `lst` before the call to `flatten()` and then look at what's being output.

Remove

```
def lst = ['a','b','c','d','e','f', 'g']

/* Remove by content */
lst -= 'd'
lst -= ['f','x', 'y', new Integer(1), 2, 'e', "g"]

lst.reverseEach { println it }
```

Output:

```
c
b
a
```

`reverseEach()` takes a closure and goes through the elements from the end.

```
List lst = ['a','b','c','d','e']

/* Remove by index */
lst.removeRange(1,3)        // Removes 'b' and 'c' (Excluding to)
lst.remove(0) == 'a'        // Removes & returns the element

lst.isEmpty()               // False

lst.removeRange(1, 100)     // IndexOutOfBoundsException
lst.remove(100)             // IndexOutOfBoundsException
```

Read & Modify

```
List lst = [4, 6, 7, 8]

/* Modify */
lst[0] = 5                  // Change value on index

/* Read */
lst.first() == 5            // True
lst.last()  == 8            // True

lst[1]       == 6           // True (Get by index)
lst[-1]      == 8           // Negative index reads from the back
lst[1..3]    == [6, 7, 8]   // True (Using Range)
lst[3, 0, 1] == [8, 5, 6]

lst.pop()                   // Removes and returns last
lst.push(9)                 // Same as lst << 9 and lst.add(9)

lst[100]                    // Returns null
lst.get(100)                // IndexOutOfBoundsException
```

Many of the methods, such as `add()` and `addAll()` are the same methods that can be found in `java.util.ArrayList`. Remember that "Groovy is a complement of the Java programming language, not a replacement of it" [8].

Other

Groovy offers many convenience methods for operations that might be handy at occasion for a developer. Many of these can be found in `java.util.Collections`, often with a new approach but some are new. Here is a few in example:

```
/* All conditions below are true */

def lst = ['b', 'c', 'a', 'aa']

lst.sort() == ['a', 'aa', 'b', 'c']             // Modifies lst

lst.reverse() == ['c', 'b', 'aa', 'a']
```

```
lst.contains('aa') == true

lst.size() == 4

lst.find { it > 'b' } == 'c'

lst.findAll { it >= 'ab' } == ['b', 'c']

lst.collect { it.toUpperCase() } == ['B', 'C', 'A', 'AA']

[2, 4].collect { it*3 } == [6, 12]

[2, [3,5], 4].flatten() == [2, 3, 5, 4]

['c', 'b', 'a', 'a'].intersect(['a', 'x', 'b']) == ['a', 'b']

['a', 'b', 'a'].unique() == ['a', 'b']          // Modifies list

[1, 2, 3].sum()                 == 6
['Apple','p', 'i','e'].sum()    == 'Applepie'

['Apple','p', 'i','e'].join('') == 'Applepie'

List syncedList = lst.asSynchronized()
```

The iterator methods `find()` and `findAll()` will find the first, respectivly all values matching the closure condition.

`collect()` will invoke and apply the passed closure on all elements, collect the values into a collection and finally return the modified collection.

7.2 Range

Groovy offers a native datatype for ranges. We can store a range in a variable or create and use them on the fly.

A range is defined by a start and end point. Ranges are specified using the double dot `..` range operator between the left and right bound [15].

```
def rng = (1..4)

rng.each { print "$it " }
```

Output:

```
1 2 3 4
```

We defined a range `rng` using the `def` keyword from 1 up to and *including* 4 and iterated over them using each. `(1..<4)` will specify the range *excluding* the value on the right side, in this case 4.

`Range`'s are very flexible. Any datatype can be used, provided that they implement the `java.lang.Comparable` interface, `compareTo()`, as well as a `next()` and `previous()` method [15].

`Character` and `Date` for example have this functionality added already:

```
('d'..<'k').each { print "$it " }
```

Output:

```
d e f g h i j
```

With date:

```
Date today = new Date()                          // Friday today
Date sunday = today + 2

(today..sunday).each { print "${it.day} " }
```

Output:

```
5 6 0
```

A week in Java starts on Sunday(0) and ends on Saturday(6).

7.3 Map

Map's are useful when we want to work with an associative set of key and value pairs.

Create, Assign, Remove and Read

Creating a `Map` in Groovy is simple. There is no need to use `new` or specify any class name:

```
/* Create */
Map mp1 = [:]                           // Empty map
def mp2 = [a:1, 'b':2, c:3, "d":4]      // Map with key:value pairs

/* Assign */
mp1.a    = 1
mp1.'b'  = 2
mp1['c'] = 3
mp1     += [d:4]

mp1 == mp2                              // True

/* Remove by key */
mp1.remove('a')                         // Returns 1    (Removes a:1)

/* Remove by value */
mp1.values().remove(2)                  // Returns true (Removes b:2)

mp1 == [c:3, d:4]                       // True
```

We can see that a `Map` can be created, manipulated and read 'on the fly'.

Assigning and reading are done with the subscript operator `mp1['c']` or via the *dot.key* `mp1.a` syntax.

A key is by default of type `String` when a new `Map` is being declared, unless the key is wrapped with parenthesis `()` which means the key is an object:

```
Object a = new Object()
Map mp3 = [(a):'txt', a:1, b:2]

/* Read */
mp3.'a'  == 1                           // True (String key)
mp3.a    == 1                           // True (String key)
mp3.(a)  == 1                           // True (String key)
mp3['a'] == 1                           // True (String key)
mp3[a]   == 'txt'                       // True (Object a as key)

mp3['b'] == 2                           // True (String key)
mp3[b]   == 2                           // MissingPropertyException
```

To *read, assign or modify* an `object:value` pair we must use the subscript operator, and this without quotes.

Quotes are otherwise a requirement when using the subscript operator. Notice how `mp3['b']` works fine while `mp3[b]` throws a `MissingPropertyException` because we used an undefined object as a key

The *dot.key* syntax will always refer to a `String` key, no matter the parantheses.

LinkedHashMap

Groovy makes working with `Map`'s simpler and more elegant with the use of closures. We can use the iterative `each()` method for maps in two ways.

Passing one parameter means that it's an entry, passing two that it's a key and a value pair.

Of course, the order can never be guaranteed because maps in Groovy are by default of type `java.util.HashMap`, but a different type of `Map` can be declared explicitly by calling the respective constructor [15]. The previously mentioned operations should work with all `Map` types.

If order is a requirement, we can use a `java.util.LinkedHashMap`:

```
Map countries = new LinkedHashMap()
countries += [Sweden:'Stockholm', Lebanon:'Beirut', France:'Paris']

countries.each { country, capital ->
    println "$country:\t$capital"
}

println ''

countries.each {
    println "$it.key:\t$it.value"
}
```

Output:

```
Sweden:   Stockholm
Lebanon:  Beirut
France:   Paris

Sweden:   Stockholm
Lebanon:  Beirut
France:   Paris
```

Notice how we used the `each()` method for maps in two different ways. Either by naming the `key:value`, or just use the keyword `it` and refer to `it.key` and `it.value`. The `each()` method is available for `java.util.HashMap` as well.

Other

```
/* All conditions below are true */

/* All countries */
def keys = countries.keySet()

/* All capitals */
def vals = countries.values()

countries.containsValue('Stockholm')  == true

countries.containsKey('Monaco')       == false

countries.Monaco = 'Monaco'           // Add

/* Countries with the same name as their capitals */
countries.findAll { it.key == it.value } == ['Monaco':'Monaco']

countries.size() == 4

countries.isEmpty() == false

Map syncedMap = countries.asSynchronized()
```

8 Other

8.1 Groovy Switch

The `switch` statement in Java is restricted to `int`, `short`, `char` and `byte`. Since Java 5 there is also support for enums [15].

The Groovy `switch` statement is similar to Java's in approach and syntax, but can take any object in addition. The `case` labels will accept all objects that implements the `isCase()` method.

Class	*Condition for true*
Object	a == b
Class	a instanceof b
Collection	b.contains(a)
Range	b.contains(a)
Pattern	b matches in a?
String	a == b
Closure	b.call(a)

Table 2: Classes that implements isCase(b) in the GDK for switch(a)

Usage in example:

```
doSw('txt'); doSw(8); doSw(3); doSw(5); doSw(7); doSw(10.0)

void doSw(def a){
  switch(a){
    case 'txt'      : print '1'; break // String
    case {it+1==9}  : print '2'; break // Closure - Returns true?
    case [1,2,3]    : print '3'; break // List  - a in [1,2,3]  ?
    case 4..6       : print '4'; break // Range - a in [4,5,6]  ?
    case Integer    : print '5'; break // instanceof Integer    ?
    case ~/\d+.\d+/ : print '6'; break // Pattern - is Double   ?
    default         : print '-'
 }
}
```

Output:

```
123456
```

8.2 Groovy SQL

Working with SQL in Groovy is a true delight. By making use of closures, Groovy SQL will take care most of the work we normally have to deal with when working with JDBC and lets us focus on building and running queries instead.

Connect

We connect to MySql database:

```
import groovy.sql.Sql                    // Must be imported

/* Connect to database */
Sql sql = Sql.newInstance(
          'jdbc:mysql://localhost:3306/name_of_your_db', 'root',
          'your_password', 'com.mysql.jdbc.Driver')
```

Note that we'll need to have the JDBC drivers for MySql available[9].

Create

Below we create a table `member` with an `id` and two columns, `username` and `password` using the method `execute()`:

```
sql.execute '''
    CREATE TABLE IF NOT EXISTS member(
      id INT AUTO_INCREMENT PRIMARY KEY,
      username varchar(30) NOT NULL,
      password varchar(30) NOT NULL
    )'''
```

Insert

We populate the table with three members using the method `execute()`:

```
List members = [
    [username:'james', password:'password1']
    [username:'peter', password:'password2']
    [username:'sarah', password:'password3']
]
members.each{ memb ->
    sql.execute """INSERT INTO user(name, password)
                   VALUES (${memb.username}, ${memb.password})"""
}
```

[9]Not to mention having MySql installed as well

Read

Reading can be done with methods `query()`, `firstRow()`, `rows()` and `eachRow()`. Here is the latter two in example:

```
/* Fetch all matching rows */
List res = sql.rows '''SELECT * FROM member WHERE username = ?
                       AND password  = ?''', ['james', 'password1']

/* Fetch all matching rows & process with the closure */
sql.eachRow('SELECT * FROM member') { // Do something here }
```

Update

```
def username = 'sarah'
def newUName = 'maria'

def up = sql.executeUpdate """UPDATE member
    SET username = $newUName WHERE username = $username"""

def ex = sql.execute """UPDATE member
    SET username = ? WHERE username = ?""", [newUName, username]

println "Returned from update : $up"
println "Returned from execute: $ex"
```

Output:

```
Returned from update : 1
Returned from execute: false
```

The `execute()` method can be used to *create*, *insert*, *update* and *delete* and will `return true` for successful operation and `false` otherwise.

The `executeUpdate()` method will return the `number` of rows that was updated.

A query that's of type `GString` or a `String` with placeholders will when passed to the right method be used to produce a *prepared statement* and are therefore automatically secured against *SQL injection attacks*.

8.3 File

Groovy follows Java's approach by using the `File` class for both files and directories with a `File` object representing a location. Groovy also adds several convenience methods, effectivly using closures as parameters to the `File` class.

The best way to demonstrate how to use the `File` class is through examples. Keep in mind that there are much more that can be done and that we're only showing a portion of it, with focus on what Groovy brings to the table.

Traverse

```
File file = new File(/c:\program files\java/)
file.isDirectory() == true     // True
file.isFile()      == false    // False

/* Directories directly under c:\program files\java */
file.eachDir { println it.path }

/* Files & directories directly under ... */
file.eachFile { println it.path }
```

Output:

```
c:\program files\java\jdk1.6.0_13
c:\program files\java\jre6
c:\program files\java\jdk1.6.0_13
c:\program files\java\jre6
c:\program files\java\readme.txt
```

Notice that `eachFile()` will traverse both files and directories while `eachDir()` only directories.

```
int nbrOfDirs = 0
int nbrOfFilesAndDirs = 0
/* Traverse all directories in ... */
file.eachDirRecurse { nbrOfDirs++ }
/* Traverse all files & directories in ... */
file.eachFileRecurse { nbrOfFilesAndDirs++ }

println 'Number of directories : ' + nbrOfDirs
println 'Number of files and directories : ' + nbrOfFilesAndDirs
```

Output:

```
Number of directories : 512
Number of files and directories : 3868
```

```
int nbrOfFiles = 0
int nbrOfReadMeFiles = 0
/* Count only files. We check for file in Closure. */
file.eachFileRecurse {
    if( it.isFile() ) nbrOfFiles++
}
/* Count number of readme.txt files */
file.eachFileRecurse {
    if(it.isFile() && it.name.toLowerCase() == 'readme.txt')
        nbrOfReadMeFiles++
}

println 'Number of files : ' + nbrOfFiles
println 'Number of readme.txt files : ' + nbrOfReadMeFiles
```

Output:

```
Number of files : 3356
Number of readme.txt files : 41
```

Create & Write

We can create directories, files and write to files in a number of ways. Easiest is by demonstrating using examples:

```
String path = /d:\exjobb\myDir/

/* Create directory myDir in exjobb */
new File(path).mkdir()     // Returns true (If doesn't exist)

/* Create file one.txt in d:\exjobb\myDir */
File one = new File(path, 'one.txt')
one.createNewFile()        // Returns true (If doesn't exist)

/* All will also create the file if doesn't exist */
new File(path, 'two.txt') << 'Line 1 in two.txt'    // Append
new File(path, 'two.txt') << '\nLine 2 in two.txt'  // Append
new File(path, 'two.txt').write('') // Overwrites (Now empty)
```

Read

```
String javaPath  = /c:\program files\java\jdk1.6.0_13/
String myDirPath = /d:\exjobb\myDir/

File copyright   = new File(javaPath , 'COPYRIGHT')
File one         = new File(myDirPath, 'one.txt')

/* Copy content into one.txt */
one.write copyright.text                          // getText()

/* Compare the size between the two files */
```

```
copyright.size() == one.length()                // True

/* Read line by line & put in lst */
List lst = []
one.eachLine { line -> lst += line }

/* readlines() method returns content in a List */
one.readLines() == lst                          // True
```

Delete

```
/* Delete file */
new File(/d:\exjobb\myDir\one.txt\).delete()    // True

/* Delete directory with all files & subdirectories */
new File(/d:\exjobb\myDir/).deleteDir()         // True
```

There are many more ways to work with `File`, both through the GDK and in Java's standard `java.io.File`.

8.4 Exception Handling

Scott Davis points out that "exceptions such as `NullPointerException`, `ClassCastException`, and `IndexOutOfBoundsException` might be thrown by a method, but the compiler doesn't require us to wrap them in a `try`/`catch` block. The Java documentation for `java.lang.Error` says that we don't have to `catch` these sorts of exceptions *since these errors are abnormal conditions that should never occur*" [7].

He continues, it's nice that Java allows this subtle disctinction but it's unfortunate that developers don't get to decide this themselves. Especially when developers often silently accepts the autogenerated code their IDE autogenerates which is often an empty `catch` block with a TODO tag, just to keep going with their work [7]. The result is lots of empty `catch` blocks in many places.

Ultimately, Groovy gives us the power to decide. But with great power comes great responsibility [26]. So the reponsibility to handle this well is passed onto the developer which now have to know when it's good to `catch` and when not. Although we are never forced to `catch` an exception, it makes sense to `catch` an exception in a lot of scenarios.

8.5 Other

There are many more features in Groovy, but writing a 700 pages book (like *Groovy in Action*) is not the scope of this document. Hopefully you now have an idea of what Groovy can do in a Java environment.

9 Introduction to Grails

The web has gotten more complex over the years and to develope competetive Web applications is hard. Todays internet environment with applications in the Web 2.0 category involves and requires the knowledge of many technlogies such as HyperText Markup Language (HTML), Cascading Style Sheets (CSS), Asynchronous JavaScript and XML (Ajax), XML, Web Services, programming languages, design patterns, frameworks and databases to mention a few [4].

"The Java Enterprise Edition (JEE) that build on the solid foundation of the Java Platform, Standard Edition (Java SE), is the industry standard for implementing enterprise class service-oriented-arhitecture (SOA) and next-generation Web applications" [18]. Christopher M.Judd & al means that "Java EE has proven over and over again that it was not written with an application level of abstraction in mind but rather focuses on a much lower technical level" [4]. He continues, "While the platform has proven to be scalable and robust", the development cycle of coding, compiling, packaging, deploying, testing and debugging requires developers to switch context too frequently to allow for fast agile development with less productivity as a consequence [4]. Java EE also has a very steep learning curve, making it hard to adopt for developers outside the business environment.

For the last couple of years, the Java community have been trying to solve these issues with bulding applications using JEE by developing many different application framworks in order to better utilize JEE. Most popular of these are frameworks Spring and Hibernate. These frameworks provide an abstraction and a ligthweight approach to Java Web application development, but are still being critized for being overly complex.

That's where Grails comes into the picture.

Christopher M.Judd & al believes "Grails is the natural next step for Java EE developers. If Spring and Hibernate provided an abstraction over Java EE and simplified development, then Grails is an abstraction over Spring, Hibernate, and JEE". Grails was influenced by other dynamic frameworks such as Django, TurboGears [4] and especially the way Ruby on Rails pioneered the innovative coupling by combining a powerful programming language with a framework "that favors sensible defaults over complex configuration" (Jason Rudolph) [5].

"Developers praise the revolutionary Rails framework for its productivity level" (Jason Rudolph) [5]. But with many organizations already running on the safety of Java to power their applications, switching to Rails is often a long and risky path to go. Something as productive as Rails was therefore needed for the Java platform.

Grails is a complete Web framework built to take advantage of proven and established technologies such as JEE, Spring, Hibernate, SiteMesh and of course Java and Groovy on the JVM.

Grails uses and exposes each of these frameworks and their capabilities via a simplified interface to make them simpler to use but continues to allow the usage of them through their documented configuration and development capabilities

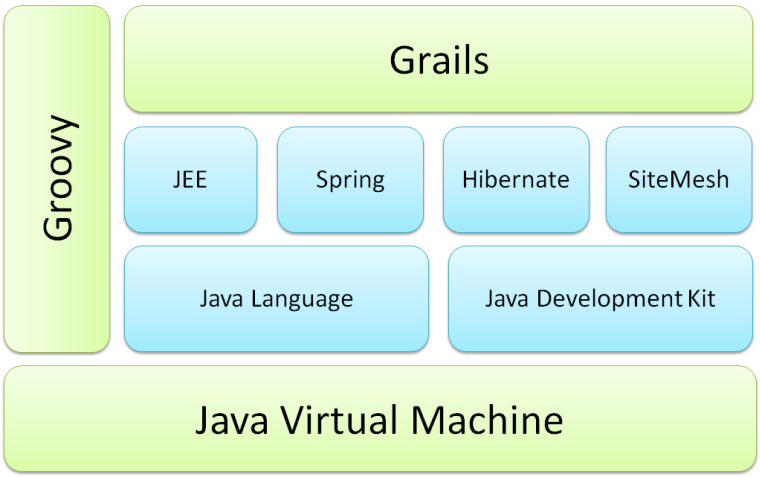

Figure 3: Grails framework

should we need to [11].

Grails applications are packaged as traditional compliant Java EE WAR (Web Application Archive) file that can be deployed on any Java application server such as Tomcat, JBoss or WebLogic. But Grails already come bundled with the powerful Jetty Server and the HSQLDB in-memory database, enabling us to be productive without any external, additional resources.

There is also a plugin system that enables developers to extend and embrace the Grails philosophy of convention over configuration. Christopher M.Judd & al describes this as an idea that was "based on the success seen by other open source projects, like the Firefox browser, in allowing the user community to add to the core platform" [4]. There are over 300 plugins available already today.

Graeme Rocher & al writes, "Grails embraces concepts such as Convention over Configuration (CoC), Don't Repeat Yourself (DRY) and sensible defaults that are enabled through the language of Groovy and an array of domain specific languages (DSLs)" [11]. With this Grails is able to provide a high productivity Web framework for the Java platform that is consistent, reduces confusion, is easy to learn and use which will ultimately make our lifes easier by enabling us to focus time and effort on developing our application.

What Groovy does for Java development, Grails does for Web development.

10 Getting Started

In this section we will try to introduce how the Grails framework is structured with som brief information. We'll also cover how to configure the database, create, and run a new application. But first you are going to need to have Grails installed.

10.1 Installing Grails

Installing Grails is easy. A prerequisite is having Groovy installed. Refer to Section 3.1 on p.9 if this isn't fullfilled.

This document is based on Grails version `1.0.3`. It's therefore recommended when going through this document to have this version installed in order for the topics and examples described to work as expected.

This guide for installing Grails 1.0.3 is applicable for Windows and Unix like systems. There are installers available but this guide is based on the **binary release** version in **zip** format, which is platform independent and the recommended way to install Grails.

1. Download the **binary release** version in **zip** format of Grails 1.0.3 from `http://www.grails.org/download/archive/Grails`.
2. Extract the content from the downloaded zip file into a location of your choice. In *c:/* will do just fine on a Windows system. You should now have a folder *c:/grails-1.0.3*.
3. Create the environment variable `GRAILS_HOME` and let it point to the extracted directory. For example *c:/grails-1.0.3*.
4. On a Windows system: Add *%GRAILS_HOME%/bin*
   On a Unix like system: Add *$GRAILS_HOME/bin*
   to the `PATH` environment variable.
5. Open up a new command prompt or terminal and enter
   » grails -version.
   If properly installed, you should see output similar to:

   ```
   Welcome to Grails 1.0.3 - http://grails.org/
   Licensed under Apache Standard License 2.0
   Grails home is set to: c:/grails-1.0.3
   ...
   ```

10.2 Editors for Groovy and Grails

Please refer to Section 3.3 on p.10.

10.3 Grails Commands

Assuming Grails is properly installed we should be able to run the command:

» **grails** *help*

Output:

```
...
grails bootstrap
grails bug-report
grails clean
grails compile
grails console
grails create-app
grails create-controller
grails create-domain-class
grails create-integration-test
grails create-plugin
grails create-script
grails create-service
grails create-tag-lib
grails create-unit-test
grails doc
grails generate-all
grails generate-controller
grails generate-views
grails help
grails init
grails install-plugin
grails install-templates
grails list-plugins
grails package
grails package-plugin
grails plugin-info
grails release-plugin
grails run-app
grails run-app-https
grails run-war
grails set-proxy
grails set-version
grails shell
grails stats
grails test-app
grails upgrade
grails war
```

These are the commands that can be run from the commad line in Grails. We will in this document use and demonstrate only a few of these, more precisely those that are highlighted.

10.4 Create Application & Grails Directory Structure

Let's start by creating a new application called *demoApp* in *d:/exjobb*.

» **cd d:/exjobb**

Create application *demoApp*:

» **grails create-app** *demoApp*

Output:

```
...
Created Grails Application at D:\exjobb/demoApp
```

The directory *demoApp* has the directory structure seen in figure 4.

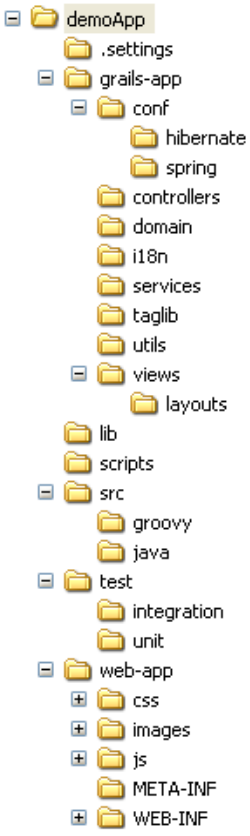

Figure 4: Grails directory structure

Below follows an overview of the structure and concept behind this with brief comments on some important directories and files in Grails.

grails-app

This is the main and core directory of the Grails application. Controllers, domain classes, views, services, internationalization(i18n) files, configuration files, taglibraries are all contained here.

conf

Contains 2 configuration files. *config.groovy* for various application settings, and *DataSource.groovy* for database configuration.

A startup script *Bootstrap.groovy* that enables us to prerun code upon start of the application.

A *URLMappings.groovy* file that allows us to alter how incoming requests should be delegated based on the URL.

controllers

Application controllers that handle and take action on incoming requests is contained in this directory.

domain

Groovy domain classes that uses GORM (Grails Object Relational Mapping) to map objects and properties from the Object Oriented world onto tables and columns in a relational database.

views

Holds Groovy Server Pages (GSP) or JSP views, templates and layouts which are responsible for rendering the interface, typically HTML in a Web application.

A subdirectory is normally created for each created controller to be used by us to hold the controllers related views and templates.

The file *error.gsp* is also contained in this directory and should be modified to show a customized error message to the user before going live with your application, should an error or exception occur. By default it will show valuable information about exceptions that happens during the development cycle. Errors can also be delegated to render another view through the *URLMappings.groovy* file in the *grails-app/conf* directory.

taglib

Contains custom dynamic tag libraries that we create, which provide a clean way to separate concerns between view and controller logic and allows us to create well formed markup code in views.

i18n

Contains internationalized message bundled files *message_xy.properties* with message codes that can be rendered to the user based on the `Locale` set for a user.

By default there are 11 files covering languages from English to Chinese that contains standard error messages. Other language files is created by creating a file and replacing *xy* with the appropiate locale, i.e. *message_ ar.properties* for Arabic.

src

Contains two directories to place additional Groovy and Java source files.

lib

Additional JARs required by the application such as JDBC drivers for the database goes in here.

test

Holds test classes for both integration and unit testing.

webapp

Static application resources such as static HTML, images, JavaScript and CSS files.

Grails already includes and integrates popular JavaScript libraries *Prototype* and *Scriptaculous* and makes their usage easier through already written tag libraries.

This folder also contains the file *index.html* that shows the 'Welcome to Grails' page which we'll see next.

Although Grails is an MVC framework that have models, views, and controllers to cleanly separate concerns, "if web applications were as simple as the MVC pattern, all this would be unnecessary. As it is, they are not, and Grails provides these features to ease commonly recurring problems" (Graeme Rocher) [10].

10.5 Run Application & Database Configuration

New and young developers often find it discouraging having to deal with several prerequisites such as finding, choosing, installing and working with configuration files. Often in unexplored fields before even gotten so far as to learn the actual framework. Surely some of those things are important at some point, but they're also a hinder that stops them from stepping into the learning cycle and getting creative.

So Grails comes bundled with everything we need to begin devoloping, testing and learning. Grails embeds the powerful web container Jetty and the temporary in-memory relational database HSQLDB which are already configured and ready to be harnessed from start.

We've already created a demo application. Let's run it now:

» **cd** *demoApp*
» **grails run-app**

Output:

```
...
Note: No plugin scripts found
Running script c:\grails\scripts\RunApp.groovy
Environment set to development
...
Server running. Browse to http://localhost:8080/demoApp
```

The application is now running on `localhost`, port `8080`. By pointing our browser to the address `http://localhost:8080/demoApp`, we should be able to see something similar to Figure 5.

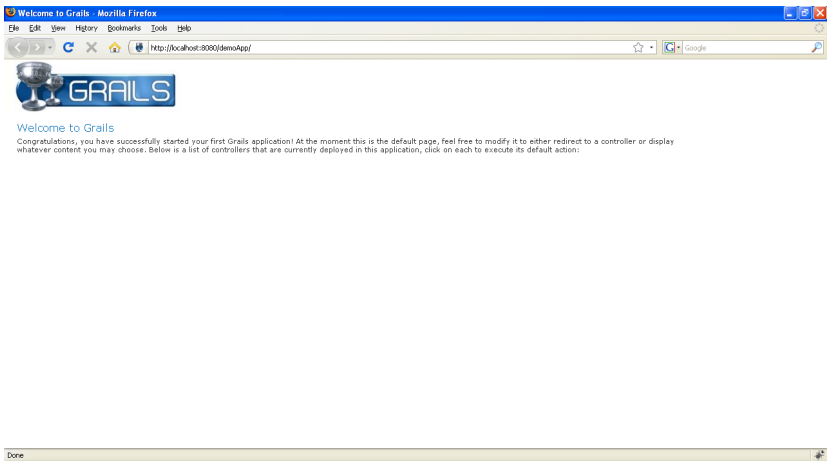

Figure 5: Running application

To skip having to type *demoApp* at the end of the URL, as should be in production mode anyway, we can set the property `app.name` in *application.properties*[10] like this:

```
app.name=/
```

The old value was `demoApp`. If we restart the application now, we should be able to reach it directly under `http://localhost:8080`.

To run the application on a **different port**:

» **grails run-app -Dserver.port=80**

[10] application.properties is located in the application top(root) directory

There is also a **Grails console** which can be a valuable resource when getting familiar with Grails. It can be used to run scripts and have access to many things in the Grails application.

» grails console

The highlighted line in the command output before, says that the environment is set to **development**(dev) mode. That's the default environment which is not intended to scale or support the load necessary in a **production**(prod) environment [4]. Grails also adds a third environment by default, **test**(test).

When configured in development or test mode, auto-reloading is enabled. This is to keep up with and make effective changes done in the codebase without the need for an application restart[11].

In production mode this feature is disabled to increase performance and minimize potential security risks [11].

Running Grails in a **different environment** is remarkably simple. For instance, the following command will run a Grails application with the production settings:

» grails prod run-app

Each environment can also be used to run separate database configurations. This is done in the *DataSource.groovy* file which can be seen in Appendix A.1.1.

Configuration of the data source is optional but the important parts to consider changing initially are lines 3-5 to set the driver class name, username and password. Also lines 16-17 for configuration of the development environment. Here's the last two properties on line 16-17 explained:

dbCreate

`create-drop`: Drop and create the database schema on every application start
`create`: Create database schema if it doesn't exist on application start and don't modify it if it does. Will delete existing data
`update`: Create & attempt to update existing database schema on application start

The default value is set to `create-drop` which can prove useful for testing, because we start off with a clean set of data each time. But let's change this to `update` so we don't have to re-populate the DB with data on every application restart:

```
16 dbCreate = "update" // one of 'create', 'create-drop','update'
```

[11] Although there are occasions where this is necesseary

url

The orginal value `jdbc:hsqldb:mem:devDB` is fine, but stores in memory only and will make stored information disappear on shutdown.

We need to change this so that data is written to a file instead. Then we are able to shutdown and restart our application and still reach our previously stored data.

```
17 url="jdbc:hsqldb:file:devDB"
```

HSQLDB in all honour, but at some point later on we are going to need to configure Grails with a more powerful database. Grails supports a wide range of avalaibale databases, from Oracle to MySQL. Here is the corresponding lines for setting Grails up with PostgreSQL:

```
3  driverClassName = "org.postgresql.Driver"
4  username = "postgres"
5  password = "your_password"

16 dbCreate = "update"
17 url = "jdbc:postgresql://localhost:5432/name_of_your_db"
```

Of course we'll need to add the PostgreSQL JDBC drivers *postgresql-8.3-604.jdbc4.jar* (for version 8.3-604) in the *lib* directory.

11 MVC model in Grails

Grails is an MVC framework, which means an application is partitioned into tiers following the MVC pattern with models, views, and controllers to separate business logic from presentation cleanly.

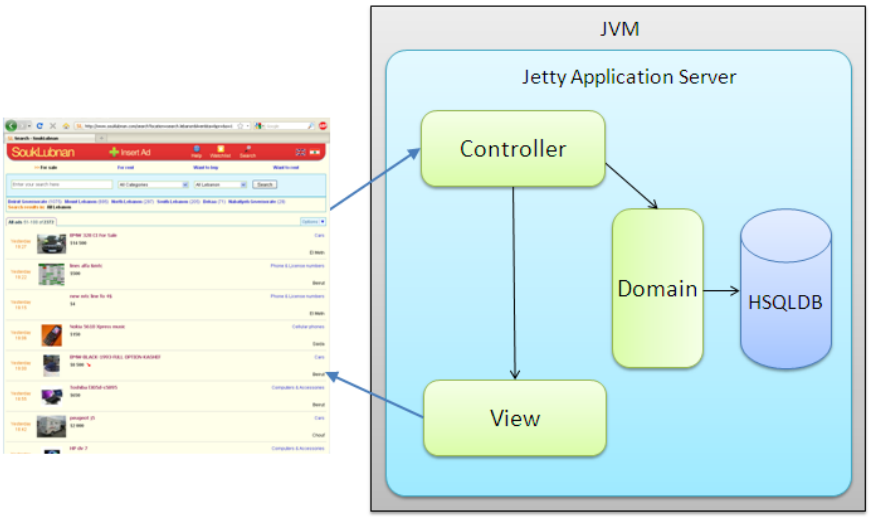

Figure 6: Grails default runtime environment [4]

A user performs a request through the browser to our application server. Based on the incoming URL, the request will be delegated to the right controller were we should have our initial logic. This could be logic to validate incoming parameters, check if a user is allowed access, interact with the database through our defined domain model, etc. In the controller we also decide what view render and pass the needed information to it. The view is responsible for rendering the interface, typically HTML in a web application.

This separation of logic into models, views and controllers enables us to change the look of our application without accidently modifying its behavior.

What's important to know is that there isn't one definition of MVC. Grails is built on Spring MVC since Spring is the underlying framework. "Spring MVC might not be the simplest framework to use but it's definitely one of the most extensible, making it perfect for Grails to build on" [22]. Grails doesn't try to reinvent the wheel but rather seeks to improve whenever there is room for improvement.

11.1 Domain

Like other Java MVC frameworks, Grails have models, referred to as domain classes but unlike other MVC models, "Grails domain classes are automatically persistable and can even generate the underlying database schema" (Christopher M.Judd & al) [4]. Grails treats the domain classes as the central and most

important component of the application. Domain classes resembles very much to what Java and Groovy developers are familiar with when defining classes but here we are actually defining the SQL schema.

Business logic in Java applications uses objects with properties, fields, and methods to represent data while databases store relational data in a table format consisting of rows and columns. There are pure Object Oriented Database Management Systems (ODBMS) that dump the concept of tables all together but Relational Database Management Systems (RDBMS) are still by far the most widely used databases [25].

Grails builds on Hibernate which provides an Object Relational Mapping (ORM) solution for applications. Explaining ORM, Graeme Rocher says, "ORM simply serves as a way to map objects from the object oriented world onto tables in a relational database. ORM provides an additional abstraction above SQL, allowing developers to think about their domain model instead of getting wrapped up in reams of SQL" [10], managing constraints, foreign keys and etc. This simplified approach has helped Hibernate achieve mass adoption, making it a de facto standard as an ORM solution [10].

Rather than building its own ORM solution from scratch, Grails wraps Hibernate in a Groovy API it calls Grails Object Relation Mapping (GORM). Instead of the mappings being defined in an external form, such as XML in Hibernate, which can be complex to manage, GORM takes the complexity out by providing a simple Domain Specific Language (DSL) on top of Hibernate that uses the convention in the classes themselves to perform the mapping [10].

To create a domain class, we can simply create a Groovy class and place it within the *grails-app/domain/* directory. We can also create a domain class by running:

» **grails create-domain-class** *member*

A domain class *Member.groovy* was created under *grails-app/domain/* as well as an integration test *MemberTests.groovy* under *test/integration/*.

The `Member` class is originally just an empty class declaration (`class Member { }`). Let's add a few properties to it:

Domain Class - Member.groovy

```
1  class Member {
2
3      String username, password, email, gender
4      int age
5      Date joined = new Date()
6
7      Set contacts
8      static hasMany = [contacts:Member]
9
10     static constraints = {
11         username(size:5..15, matches:"[a-z]+",
12                              blank:false, unique:true)
13         password(size:5..15, blank:false)
14         email(maxSize:255, email:true, blank:false)
15         gender(inList:['Male', 'Female'], blank:false)
16         age(range:18..120)
17     }
18 }
```

What we've done above is to define an SQL schema by adding a couple of properties that are very similar to what we are used to in Java. For example, `String` will by default be mapped to `varchar(255)`, `int` to `integer`, `Date` to `timestamp` which in this case will be set to current date on creation, because we have given it a default value.

On line 8 we declare a many-to-many relationship `contacts` in `static hasMany` of type `Member`. That is, a member can have many members in `contacts`. The relationship is also declared to be of type `Set`, so that a user can't have the same member in `contacts` more than once.

In our SQL schema we get two tables (relations), one main table, containing all the declared properties, and one for the relationship contacts with two fields (`member_id, member_contacts_id`).

Hibernate also adds two additional columns, `id(bigint)` and `version(bigint)` to the main table. The `id` is used as an identifier which by default is managed and incremented automatically. `version` is incremenented once for every update on the row and used to ensure transactional integrity and support optimistic locking for concurrent updates.

The constraints on lines 10-17 defines what is allowed to store in each property defined. For example, `username` must have a length between 5 to 15 characters, contain only characters `a-z`, can't be blank (separate from size), and must be unique in the table. Should a constraint be broken, the row can't be saved and the error can be rendered to the user. Notice the email validator. Grails have several predefined validators (See Section 12.7). We can also create our own custom validators in the domains as we shall also see later on. Constraints can affect how types are stored in the database, for example `username` is now of type `character varying(15)` rather than `varchar(255)` as is the default for unaltered `String` properties.

11.2 Controller & View

Controllers are the orchestrators of the Grails application. They typically take input from the user's web browser and are responsible for handling and coordinating incoming requests. They usually do some work with the requests such as directly interact with the domain model, redirect to a different action or controller, or prepare and send the response to a view which renders the appropiate information to the user. Basically, a controller handles requests and prepares the response. "A controller is prototyped and request scoped, meaning that a new instance is created per request" (Graeme Rocher & al) [11].

To create a controller, we can simply create a Groovy class with a name that ends with `Controller` and place it within the *grails-app/controllers/* directory. The class name must end with `Controller` by convention. A controller can also be created by running:

» **grails create-controller** *Demo*

A controller class *DemoController.groovy* was created for us under *grails-app/controllers/* as well as an integration test *DemoControllerTests.groovy* under *test/integration/*.

An empty directory *demo* was also created under *grails-app/views/*.

The `DemoController` class is originally just an empty class declaration with an empty closure action (`class DemoController { def index = { } }`). Let's add three actions to it:

Controller - DemoController.groovy

```
1  class DemoController {
2
3      def index = { }
4
5      def acOne   = {
6          render('Hello, world!')
7      }
8
9      def acTwo   = {
10         render("Hello ${params.id ?: 'noName'}!")
11     }
12
13     def acThree = {
14         render("""My name is ${params.firstName}
15            ${params.lastName} but the id ${params.id}
16            is my nickname.""")
17     }
18 }
```

We added the three actions, `acOne`, `acTwo` and `acThree` to the `DemoController` class.

Grails identifies by default from the URL (specified in *URLMappings.groovy*) what action should be run by the following pattern:
`http://.../controller/action/id`

If we run our application and point the browser to:

`http://localhost:8080/demo/acOne`
We should see the message *Hello, world!*

`http://localhost:8080/demo/acTwo`
We should see *Hello, noName!*

`http://localhost:8080/demo/acTwo/Moe`
We should see *Hello, Moe!*

`.../demo/acThree/Moe?firstName=Mohamed&lastName=Seifeddine`
We should see *My name is Mohamed Seifeddine but the id Moe is my nickname.*

In the last example we passed `id`, `firstName` and `lastName` as parameters in the URL which can all be read on the server side from the `params` object as can be seen in the action `acThree` above. The `params` object will also hold the input values from form submissions.

Grails provides the `render()` method to send content back to the user, but controllers are normally not supposed to render any response directly to the user (except for small Ajax responses). Glen Smith & al argue that "embedding HTML inside the code is always a bad idea. Not only is it difficult to read and maintain, but a graphic designer will need access to the source code in order to design the pages" [17].

They continue "The solution is to move the display logic into a separate file, which is known as the **view**, and Grails makes this simple". A view is responsible for rendering the interface, normally HTML in a web application. Views and controllers are strongly related though, since the input from the user and output from the application strongly relate. But controllers are supposed to prepare and choose how and what view should be rendered. Usually by interacting with `params`, `session`, domain model and then pass some information on that's used in the view.

Views in Grails are typically Groovy Server Pages (GSP) which is an extension of JSP but Grails supports both types. GSP however, are more flexible and convenient to work with than JSP. GSP files end with the extension *.gsp* and are located in the *grails-app/views/* directory or any subdirectory under that location.

Let's create a new file *index.gsp* in the directory *grails-app/views/demo/* with the following content:

View - /demo/index.gsp

```
1 <html>
2   <body>
3     <h1>Rendering from the index view!</h1>
```

```
4     </body>
5   </html>
```

If we now point the browser to: `http://localhost:8080/demo`
We should see the message *Rendering from the index view!* as a headline.

The reason to this is that when we point the browser no further than the controller name and without any action, the `index` action in that controller will by default get hit.

To use a different default action, we can add `def defaultAction = 'acOne'` to the controller and the `acOne` action will get hit instead.

Also, because the `index` action had an empty skeleton and we haven't explicitlty rendered anything, Grails will automatically try to render the view named the same as the action that was hit, and located in the controllers view folder. In the last example the */demo/index.gsp* view was rendered when we hit the `index` action in the `DemoController`.

To render a different view we can do:

```
render(view:'differentView')
```

Or

```
render(view:'/differentDir/differentView')
```

for a view in a different view directory.

Each GSP view also has access to a `model` which is basically a map with keys and values that can be passed from the controller and used in the view. Here we are passing two objects from the `index` action in `DemoController` to the */demo/index.gsp* view using `model`:

```
2  def index = {
3      render(view:'index', model:[objOne:'Yeeha', objTwo:'!'])
4  }
```

Or, since we're using the **default view** for this controller and action, just:

```
2  def index = {
3      [objOne:'Yeeha', objTwo:'!']
4  }
```

In */demo/index.gsp* we can take advantage of the passed `model` like this:

```
3  <h1>
4      Rendering from the index view! ${objOne}${objThree}${objTwo}
5  </h1>
```

If we now point the browser to: `http://localhost:8080/demo`
We should see the message *Rendering from the index view! Yeeha!* as a headline.

Notice the unpassed object `objThree` in the view. As mentioned earlier in the Groovy chapter, Grails will output `null` in views as empty string. This proves to be very useful when reusing views which differ slightly in content from different actions. Either there is a value/text to render or there isn't. So there is no need for `if` blocks to determine to output or not.

Views can include Groovy code and they also have access to the domain models although such usage should be limited and separated. Graeme Rocher means that "mixing scriptlets and markup code is most definitely recognized as a bad thing" and Grails provides an easy way for us to create custom tags just like JSP does to better separate logic but without having to sacrifice any agility [11]. Views can also be separated into view templates as we'll see in Section 14.3 on p.104.

12 More on Domain

In this section we'll try to demonstrate some basic operations for creating, reading, updating and deleting data using GORM, primarly on the domain class *Member.groovy* we created earlier. These operations can pretty much be called from anywhere in the Grails application and for testing, the Grails console will often work just fine[12].

We'll also try to introduce some other key topics deemed as good to know.

12.1 Create

Let's start by populating the database with three members:

```
def james = new Member(username:'james', password:'password1',
                       email:'james@lth.se', gender:'Male',
                       age:'21')
james.save()

Member peter = new Member(username:'peter', password:'password2',
                         email:'peter@lth.se', gender:'Male',
                         age:'25').save()

def sarah = new Member(username:'sarah', password:'password3',
                       email:'sarah@lth.se', gender:'Female',
                       age:'18').save(flush:true)
```

One thing with Hibernate is that when you call save, it doesn't neccessarily perform the SQL operations at that point but batches up SQL statements and executes them at the end, which is typically handled by Grails. This is not always something we want.

On occasion, we might want to control this ourselves. To do so we can provide the flush argument to the save method `save(flush:true)` which will flush not only the instance it's being used on but all pending SQL statements.

Grails also provides something useful called **binding**:

```
Map mp = [username:'erica', password:'password4',
          email:'erica@lth.se', gender:'Female', age:'17']

def erica = new Member(mp).save(flush:true)
```

Binding can be of good use when users are provided an HTML form with input fields named the same as those defined in the domain. We can for example have the following input in a form:

[12] Although there are some known issues with the console, such as handling constraints properly

```
<input type='text' name='username'/>
```

The parameter `Map params` can then be used to bind instead i.e `new Member(params)` rather than having to set each property manually.

When binding from a request parameter we need to be careful not to allow users to bind malicious data that end up being persisted to the database. This could be a property declared in the domain that we normally keep track of ourselfs, perhaps a visit counter.

The `bindData()` method allows the same data binding capability but also allows us to exclude certain properties from being set. To exclude the property `username` we can do:

```
def demo = new Member()
bindData(demo, params, ['username'])       // Exclude username
demo.save(flush:true)                      // Won't be saved
```

Perhaps you noticed earlier, that the member *erica* was violating the age constraint and therefore the instance wasn't successfully persisted to the database. The variable `erica` was therefore `null` which was returned from the `save(flush:true)` call.

If we wish to find out if an instance fullfills our constraints without saving, we call the method `validate()` which will `return true` or `false` and then use `hasFieldError('age')` or `errors.allErrors.each{...}` on that instance. There are taglibs to do this in a convient way in views as well, described in Section 12.7.

Now we have **three** members persisted in the database. To add *peter* and *sarah* to *james* `contacts` we can do:

```
james.addToContacts(peter)
james.contacts += sarah
james.save(flush:true)
```

`james`, `peter` and `sarah` are the variables declared in the beginning of this section.

12.2 Read

There are many ways we can read/fetch data from the database. We can use GORM Dynamic Finders, Criteria, Hibernate object oriented query language HQL(not covered here) or use SQL. Let's start with the basics:

get & getAll

```
Member.get(1)             // Retrieval by id
Member.getAll([1,2,8])    // Can take a list of id's
Member.getAll()           // Fetches all members in database
```

list

```
Member.list()             // Fetches all members in database
Member.list(max:10)
Member.list(offset:10, max:10)
Member.list(offset:10, max:10, sort:'username', order:'desc')
```

findBy & findAllBy

A dynamic finder that looks like a `static` method but acutally doesn't exist in the code level. A method is automagically generated at runtime based on properties of the domain class.

`findBy()` attempts to find a maximum of one instance:

```
Member.findByUsername('james')
Member.findByUsernameLike('%ame%')
Member.findByUsernameIlike('%aMe%')      // Case Insensitive Like
Member.findByAgeGreaterThan(20)
Member.findByJoinedLessThan(new Date())
```

`findAllBy()` is similar to `findBy()`, but will return all matching instances instead of only one:

```
Member.findAllByUsername('james')
Member.findAllByUsernameNotEqual('james')
Member.findAllByUsernameIsNotNull()
Member.findAllByAgeBetween(5, 30)
Member.findAllByJoinedLessThanEquals(new Date())
Member.findAllByJoinedLessThanAndGender(new Date(), 'Male')
```

count() & countBy()

```
Member.count() == 3                    // True
Member.countByGender('Male') == 2      // True
```

findWhere() & findAllWhere()

```
Member.findAllWhere(username:'james', gender:'Male', age:21)
```

The `findBy()`, `findAllBy()` and `countBy()` methods can compare on maximum **two** properties and use the comparators seen in Table 3.

Comparator	*Description*	*Argument*
LessThan	Less than the given value	One
LessThanEquals	Less than or equal a give value	One
GreaterThan	Greater than a given value	One
GreaterThanEquals	Greater than or equal a given value	One
Like	Equivalent to a SQL like expression	One
Ilike	Similar to a Like, except case insensitive	One
NotEqual	Negates equality	One
Between	Between two values	Two
IsNotNull	Not a null value	None
IsNull	Is a null value	None

Table 3: Comparators for findBy(), findAllBy() and countBy() [21]

12.2.1 Criteria

We can also use Hibernates Criteria which is an excellent way to build and run advanced queries. Criteria queries in pure Hibernate is a bit of a pain [4]. However, GORM have with the help of the Groovy language enabled a smoother way to use Hibernates Criteria API.

Criteria can be used either via `withCriteria()` or `createCriteria()`. The advantage of criteria queries is that they provide an almost complete way to query domains, even their relationships such as `contacts` in `Member`. Let's try to go through a fairly complex example using `withCriteria()` and try to explain some parts of it:

```
1  Member.withCriteria {
2
3      def demoVariable = 'We can use Groovy code in here!'
4
5      if(demoVariable.size() > 5) {
6          isNotNull('email')
7      }
8      else {
9          sizeEq('contacts', 100)
10     }
11
12     or{
13
14         and{
15             eq('gender', 'Male')
16             gt('age', 20)          // Greater than
17         }
18         and{
19             eq('gender', 'Female')
20             le('age', 18)          // Less than or equals
21         }
22     }
23
```

```
24      isNotEmpty('contacts')
25
26      contacts {
27          or{
28              eq('username', 'sarah')
29              eq('username', "doesn't_exist")
30          }
31      }
32
33      maxResults(10)
34      order('joined', 'asc')
35  }.each{ println it.username }
```

Output:

```
james
```

The real action happens between line 12-22 where we are fetching all members that are (male AND above 20) OR (female AND 18 or below). This will match all our three domain instances in the database.

At line 24 we are saying that the relationship `contacts` cannot be empty, where *james* is the only match.

At line 26-31 we are saying that the relationship `contacts` (defined of type `Member`) must contain a member with `username` either *sarah* or *doesn't_ exist*. Since *sarah* is among *james* contacts, *james* is still on.

For more information on how to use Criteria in Grails, please refer to [21].

12.2.2 SQL

Using SQL in Grails is easy and no different from standard Groovy SQL. The domains won't be involved and the communication is done directly with the database.

Since we have already defined what database to use in *DataSource.groovy* we can use that to connect with. Grails supports *dependency injection by convention*. In other words, we can use the property name representation of a service, to automatically **inject** them it into controllers, tag libraries, and so on[13]. But not everywhere. To use SQL in Grails we also have to `import` the Groovy SQL library:

```
import groovy.sql.Sql                          // Must be imported
class DemoController {

    def dataSource                             // Inject dataSource
    def demoAction = {

        Sql sql = new Sql(dataSource)          // Use of dataSource
        def res = sql.rows("""SELECT * FROM member
```

[13] Read `http://en.wikipedia.org/wiki/Dependency_injection`

```
            WHERE username = ?
            AND
            gender = ? """, ['james', 'Male'])

...
```

12.3 Update

Updating an instance

Updating an instance using GORM is straightforward.

If we have the instance already, we can make a change to it and then save. For example:

```
def memb = Member.findByUsername('sarah')
memb.username = 'maria'
memb.save(flush:true)

Member.findByUsername('james').contacts.each{println it.username}
```

Output:

```
peter
maria
```

As you can see the update will be reflected in *james* contacts. On update the `id` of the updated element stays the same, but the `version` will be incremented once.

Updating the database schema

Updating the database schema is not always as straightforward.

If we wish to add or remove a separate domain, then that's not a problem. To remove a property in a domain isn't a problem either, as long as it's no longer used in the code. The column will still be there in the table and it's up to us to drop the column, although this is not neccessary. The reason for not dropping automatically is likely to prevent accidental loss of data triggered by a small change done in the domain.

The problem lies in that we cannot always introduce a new property in an already populated domain without doing some extra work. Most databases will fill the new column with NULL values in the already existing tuples, and when a tuple is *read* with GORM this can become an issue. For most types this is not a problem either, because we can add the constraint `nullable:true` until those values have been updated but on types such as numbers which cannot be `null`, this won't work. We need to change these column values manually, for example by using SQL. This can for example be done in *BootStrap.groovy*, a Groovy Script, with the help of the Autobase or LiquiBase Grails plugins or in some other way outside the Grails application.

Let's add a new optional property `city` to the `Member` class. We can do this as a `String` but if we wish to add coordinates to each city, or add cities to other domains, a domain of its own is then justified. So we create a new domain class:

» **grails create-domain-class** *city*

Let's add a few properties to it:

Domain Class - City.groovy

```
1  class City {
2
3      String cityName
4      double latitude, longitud
5
6      static constraints = {
7          cityName(blank:false)
8      }
9  }
```

We can now add `City city` to the `Member` class and add to its constraints `city(nullable:true)` to make it optional(nullable).

Let's populate the database with two cities and update *james* to be from the city of Lund.

```
def lund = new City(cityName:'Lund',
                    latitude:55.707352,
                    longitud:13.197124).save()

           new City(cityName:'Göteborg',
                    latitude:57.708733,
                    longitud:11.975098).save()

def james = Member.findByUsername('james')
james.city = lund
james.save(flush:true)
```

12.4 Set, List and Map

12.4.1 Set

When we define a `hasMany` relationship with GORM it's a `Set` by default and doesn't have to be explicitly specified, although this can be done for clarity. A `Set` is an unordered collection that doesn't contain duplicates. We've already defined our *many-to-many* relationship `contacts` in *Member.groovy* to be of such type.

The problem with `Set` is that there is no ordering when accessing the collection. The order is random, and if we try to output *james* contacts several times, we'll see that the order does change. However[14], as of Grails 1.0.4, sorting on a property is possible by a simple add to `static mapping` which is described in Section 12.8 on p.89.

12.4.2 List

If we wish to keep objects in the order they where added and be able to reference them by index like an array we can define the type as a `List` instead:

```
class Member {

    List lstContacts
    static hasMany = [lstContacts:Member]

    ...
}
```

We can then do:

```
def james = Member.findByUsername('james')
james.addToLstContacts(james)
james.lstContacts   += james
james.lstContacts[2] = james
james.save(flush:true)

print james.lstContacts[0].username  + " " +
      james.lstContacts.size()
```

Output:

```
james 3
```

We added *james* to his own `lstContacts` three times for the sake of keeping the example short and simple. Of course, any other member could have been added instead.

[14] SortedSet in 1.0.3 is also a possibility but is somewhat buggy

12.4.3 Map

A map can be defined the same way as a standard map but it's keys must always be of type `String`.

If we wish to define a *simple map* with string/value pairs, there is no need to add it to the `hasMany` property unlike with `Set` and `List`. But then **both** the key and value must be of type `String`.

If we want a `Map` of key:object, then `hasMany` is required because we have to define the one and only object type.

To continue on our example with `contacts`, a `Map` can prove useful for example if we where to let members name their contacts themselves.

```
class Member {

    Map simpleMap  = [:]  // key:value must both be of type String
    Map mpContacts
    static hasMany = [mpContacts:Member]

    ...
}
```

We could then add to `simpleMap` and `mpContacts` like this:

```
def james = Member.findByUsername('james')

james.simpleMap  += [bgColor:'blue', showAge:'false']

james.mpContacts += ['My Best Friend':
                      Member.findByUsername('peter')]

james.save(flush:true)
```

In `simpleMap` here we stored what appears to be some settings for the `Member` *james*. We also added *peter* to *james* `mpContacts` under the alias key `'My Best Friend'`.

12.5 Relations

Before we continue, we need to introduce a new simple domain class `Blogpost` with the following content:

Domain Class - Blogpost.groovy

```
class Blogpost {
    String content
    Date added = new Date()
    Date updated

    static belongsTo = [member:Member]
 // static belongsTo = [Member]                /* Alt */
 // Member member                              /* Alt */

    static mapping = {
        content type:'text'   // So content can be >> 255 chars
    }
    static constraints = {
        content(blank:false)
    }
    def beforeUpdate = {
        updated = new Date()
    }
}
```

We also add a one-to-many relationship as a `List` to `hasMany` in *Member.groovy*, so that the class now looks like:

Domain Class - Member.groovy

```
class Member {
    String username, password, email, gender
    int age
    Date joined = new Date()
    City city
    List blog
    static hasMany = [contacts:Member, blog:Blogpost]

    static constraints = {
        city(nullable:true)
        username(size:5..15, matches:"[a-z]+",
                             blank:false, unique:true)
        password(size:5..15, blank:false)
        email(maxSize:255, email:true, blank:false)
        gender(inList:['Male', 'Female'], blank:false)
        age(range:18..120)
    }
}
```

12.5.1 Owner

GORM allows us to define who the owner is of a relationship so that saves, updates and deletes will cascade from the owning class to its possessions (the other side of the relationship).

On line 6 in `Blogpost` we are saying that an instance of `Blogpost belongsTo` an instance of the `Member` class. If we for example, where to let *maria* write and append a blogpost to her `List blog`, she would be the owner of the `Blogpost`:

```
def maria = Member.findByUsername('maria')
maria.addToBlog( new Blogpost(content:'My first blogpost!') )
maria.save(flush:true)
```

Note that we didn't have to explicitly save the new blogpost. The save on `maria` will be cascaded and the new blogpost will be saved as well. This is however not explicit to `belongsTo` but default behavior in GORM.

On line 7 in `Blogpost` we are showing a different way that `belongsTo` can be declared. The first declaration on line 6 allows us to reach the owning instance.

For example, if we where to present blogposts on a common page, we could with the declaration on line 6, reach and present their owners easily:

```
/* Newest blogposts */
Blogpost.list(max:10, offset:0, sort:'added', order:'desc').each {

println """\
Written by: $it.member.username
$it.content
Added: $it.added"""

}
```

Output:

```
Written by: maria
My first blogpost!
Added: 2009-08-20 14:20:53.031
```

All blogposts where listed. Notice that we used `it.member.username` to refer to the owning `member` of the `Blogpost`. The declaration on line 8 will for example not allow for such back reference.

If we where to delete *maria*'s account with the declaration on line 7 or 8, the deletion will be cascaded and all her blogposts will be deleted as well.

If we do not wish to cascade deletes but still like a back reference, we can simply declare it as seen on line 9 where GORM will automatically map an added `Blogpost` to `blog` in `Member` to this property.

12.5.2 One-to-one

A one-to-one relationship is defined using a property of the type of another domain class. The example `City city` in the `Member` class is of such type.

12.5.3 One-to-Many

A one-to-many relationship is when one class, has many instances of another class. For example, a `Member` can have many `Blogpost`'s.

If there are two or more properties of the **same type** on the many side we can use `static mappedBy` to specify how the collection is mapped:

```
class Airport {

    String name

    static hasMany  = [outFlights : Flight,
                       inFlights  : Flight]

    static mappedBy = [outFlights : 'departureAirport',
                       inFlights  : 'destinationAirport']

    ...
}

class Flight {

    Airport departureAirport
    Airport destinationAirport

    ...
}
```

This can be used like this:

```
Flight  flight = new Flight(...)
Airport landve = Airport.findByName('Landvetter')
Airport beirut = Airport.findByName('Beirut')

landve.addToOutFlights(flight)
beirut.addToInFlights(flight)
beirut.save(flush:true)

println "Going from: $flight.departureAirport.name"
println "Going   to: $flight.destinationAirport.name"
```

Output:

```
Going from: Landvetter
Going   to: Beirut
```

In the example above, we started off by creating a new `flight`. We then added the `flight` to the `outFlights` of `Airport` *Landvetter* and to `inFlights` of the `Airport` *Beirut*. Notice then how we in the end can reach where we're going, from and to, by using only the `flight` instance.

12.5.4 Many-to-Many

Grails supports many-to-many relationships by defining a `hasMany` on both sides of the relationship. A movie can have many actors, and an actor can be in many movies for example. In our case, we have a many-to-many relationship between members.

12.6 Delete

To delete an instance we can use the `delete()` method. It can be used without or with the `flush` argument:

```
def maria = Member.findByUsername('maria')
maria.delete(flush:true)

Blogpost.count() == 0            // True
```

When we deleted *maria* the deletetion was cascaded and we also deleted her blogpost because of `belongsTo = [member:Member]` in `Blogpost`.

12.7 Constraints

Constraints have been covered pretty much already. In table 4 we see a list of already written and ready to be used constraints.

Name	*Example*	*Comment*
blank	login(blank:false)	
creditCard	cardNumber(creditCard:true)	
email	email(email:true)	
inList	login(inList:['Bob', 'Eve'])	
length	login(length:5..15)	For String or Array
minLength	login(minLength:5)	
maxLength	login(maxLength:15)	
min	age(min:new Date())	Minimum value
max	age(max:new Date())	
notEqual	login(notEqual:'Bob')	Not equal value
nullable	age(nullable:false)	
matches	login(matches:/[a-zA-Z]/)	Matching using regex
range	age(range:18..25)	Valid values
size	staff(size:5..15)	Restricting size of collection
minSize	staff(minSize:5)	
scale	salary(scale:2)	Scale for float numbers
unique	login(unique:true)	
url	homePage(url:true)	If String is an URL address

Table 4: Domain constraints

Each constraint have one or several **error codes**, defined by the following pattern:

```
[DomainName].[PropertyName].[ConstraintCode]
```

The value of these codes needs to be defined in one of the message language files in the internationalization directory *grails-app/i18n/*, for example as:

```
member.username.blank=Username cannot be empty.
member.gender.not.inList=Please specify your sex.
member.age.range.toosmall=Minimum age is 18.
member.age.range.toobig=Are you sure you'r alive?

myValid.passw.cant.be.username=Password can't be your username. \
Please choose a different password.
```

The last error error code is one defined by us in the following **custom validator** added to *Member.groovy*:

```
static constraints = {

  password(length:5..15, blank:false, validator: { input, memb ->

          if( input == memb.username )
              return 'myValid.passw.cant.be.username'
    })
  ...
```

The first argument passed to the closure is the value of the property being validated, in this case `password`. The second argument gives access to the domain instance being validated. The second argument is often useful if validation requires the inspection of the instances other properties as seen here.

On error, we `return` a *custom error code* that can be rendered to the user, preferably from within a view using these tags[15]:

```
<g:eachError bean="${memb}">
    <div>
        <g:message error="${it}"/>
    </div>
</g:eachError>
```

Or if we do not wish to render all errors in one place, but perhaps on a specific place for certain properties, here as an example for `password`:

```
<g:eachError bean="${memb}" field="password">
    <div>
        <g:message error="${it}"/>
    </div>
</g:eachError>
```

In these two last examples we're assuming a domain instance `Member memb` has been passed to the view.

12.8 Mapping

What haven't been mentioned before is that properties written in CamelCase such as `cityName` in `City`, are named differently in the database. The column names are transformed into snake_case, that is, lowercased and separated by an underscore. So in the `city` table we have a column named `city_name` instead and have to use this when communicating with the database using SQL.

As we've already seen, Grails does a good job of mapping the domain model to a relational database without requiring any kind of external mapping file.

But if we need to tailor the way GORM maps onto legacy schemas, performs caching or we're not happy with the conventions defined by GORM for tables, column names etc, we can define custom mappings using the `static mapping`

[15]There is also a hasError tag that's could prove useful. We'll discuss tags in greater detail in Section 14.2 on p.99

block defined within the domain class, as seen on line 10-12 in *Blogpost.groovy* in Section 12.5 on p.84.

What we done there is to map `content` from `varchar(255)` to `SQL TEXT` or `CLOB` type depending on the database being used.

There are many things that can be set and altered in the `static mapping` block and below are a few useful examples:

```
static mapping = {
    /* Rename the table in the database */
    table 'renamed_city_table'

    /* Rename the column city_name */
    cityName column:'renamed_city_name'

    /* Custom database identity */
    id generator:'assigned'              // Other: hilo, uuid ...

    /* Indices */
    latitude index:'lat_idx'
    longitud index:'lat_idx, long_idx'

    /* Disable versioning */
    version false
}
```

12.9 Other

12.9.1 Events

GORM supports registration of events as closures that get fired when certain events occurs. To add an event, simply add one of the relevant closures `beforeInsert`, `beforeDelete`, `beforeUpdate` or `onLoad` to the domain class [21]. In later Grails versions there is also the `afterInsert`, `afterUpdate`, `afterDelete` and `beforeLoad` events. None of the closures take any parameters or `return` any values, so properties need to be either in the domain instance self or retrievable by the current thread.

On line 16-18 in `Blogpost.groovy` in Section 12.5 on p.84 we've added the `beforeUpdate` closure where the variable `updated` of type `Date` will be set to the current date on the instance being updated.

12.9.2 Methods

We can also add methods to the domain model in the same way as in standard Groovy classes. If declared as `static` they can be reached with and without an instance, and if not they can be reached with an instance only.

13 More on Controller

In Section 11.2 on p.71 we briefly discussed what controllers are and demonstrated some basic things. In this section we'll try to introduce what else is out there that might be good to know when working with controllers.

13.1 Scope

Grails supports different scopes to store information in with all lasting a different length of time. "Grails lets us reference them explictly so that we can choose to store data for as long as we need" (Glen Smith & al) [17].

13.1.1 Request

The `request` scope holds objects for the duration of the currently executing request and will hold that data only until the view has finished rendering, making the `request` scope stateless.

The `request` object is an instance of Javas Servlet API's `HttpServletRequest`. The `HttpServletRequest` class is useful for things such as obtaining request headers and establishing information about the user when a request comes in to the server. It manages parameters and data from form submission. The `paramaters` property in `request` is for example used to generate the `params` instance seen earlier in Section 11.2[16].

The `request` scope is great when we want to store data that's only useful during the current request, for example to share data between a controller and view, or view and wiew-templates. Glen Smith & al explains that "when data was passed as a `Map` in the `model` from controller to view earlier, we where implicitly making use of the `request` scope" [17].

Grails enhances the `HttpServletRequest` by adding several properties and methods that makes its API easier to use. For example, properties in `request` are normally accessible via `request.getAttribute('propertyName')`. In Grails we can access and set the same properties using the subscript operator or via the *dot.key* syntax as well.

Below are just a few standalone examples that gives an idea of how `request` can be used:

```
if( request.method == 'POST' || request.method == 'GET' ) {...}
/* OR */
if( request.post || request.get ) {...}


/* Assign variables to be kept only during this user request */
request.myVariable  = new String('My Object')
```

[16]The params instance will in addition also contain information on what controller and action was requested

```
request.myVariable == 'My Object'   // True

request.remoteAddr                  // Ip of the incoming request
```

13.1.2 Session

The `session` scope allows us to associate data on the server with individual users. Objects placed into `session` are kept until the user `session` is invalidated, either manually for example on logout or through expiration, usually by inactivity. The default expiration time is 30 minutes but will be prolonged as long as the user does new requests to the server.

The `session` object is an instance of Javas Servlet API's `HttpSession` class, and is really easy to use. Below is a simple `login` and `logout` action that makes use of `session`:

```
1  def login = {
2      if(request.post && params.username){
3
4          def memb = Member.findByUsername(params.username)
5          /* Success */
6          if(memb && memp.password == params.password){
7              session.loggedIn = memb
8              return redirect(action:'settings')         // Redirect
9          }
10     }
11
12     render(view:'login')                              // Not nessecary
13 }
14
15 def logout = {
16     session.removeAttribute('loggedIn')
17     redirect(uri:'/')                        // Redirect to startpage
18 }
```

We can see the use of `session` on line 7 and 16.

On line 12 we explicitly say that the view 'login' should be rendered if the above conditions fail. Had we not, the same view would still have been rendered.

A note on the `if` statement on line 6. If we instead would've used the Safe Navigation Operator like this:

```
if(memb?.password == params.password)
```

It's possible that we would have entered the block if a `Member` is not found `memb == null` and a `password` field is not submitted because of form manipulation making `params.password == null`.

13.1.3 Flash

Flash scope is a concept introduced by Rails [28]. The `flash` object is a `Map` and can therefore be used to store key:value pairs which are transparently stored in `session`. Objects placed into `flash` are kept for the duration of the current request and the next request only. They are automatically cleared out when the next request completes.

The flash could be used together with `redirect` instead of a `chain` (described next) although a common use case is to store a message when some sort of validation fails:

```
def profile = {
    def memb = Member.findByUsername(params.username)
    if(!memb) {
        flash.message = "Member $params.username not found"
        return redirect(controller:'search')
    }

    [memb:memb] // Render 'profile' view & pass memb to it
}
```

When a `Member` is not found, we are redirected to `/search/defaultAction` where we can take advantage and show `flash.message`. Either in the controller or more likely in a view for the search results.

13.2 Redirect & Chain

13.2.1 Redirect

Often, an action needs to be redirected to another controller/action. Actions can be redirected using the `redirect()` method available in all controllers.

The method takes a `Map` as an argument and can be used in a couple of ways. Internally the `redirect()` method uses the `sendRedirect()` method in Javas Servlet API's `HttpServletResponse`.

We can redirect to another action within the same controller, action in a different controller, a URI for a resource in the Web application or to an absolute URL.

```
redirect(action:'login')
redirect(controller:'search')
redirect(controller:'search', action:'results')

redirect(uri:'/file/in/web-app')
redirect(url:'http://www.cs.lth.se')
```

On completion of a redirect, the data in the previous `request` like parameters or submitted form data isn't preserved but is instead lost. Parameters can be optionally passed from one action to the next using the `id` and/or the `params` argument:

```
redirect(action:'login', id:'someId')
redirect(action:'login', params:[a:1, b:2])
redirect(action:'login', id:'anId', params:[a:1, b:2, c:3])
```

A redirect will happen also in the users browser and the new action can be seen in the adress field of the browser, as well as the parameters passed to the new action. The latter can be undesired, for instance when we're redirecting a POST request consisting of a form submission.

13.2.2 Chain

Although we can pass parameters in a `redirect()`, it has limited usage and is not complete because parameters can only have text based values. They are also visible to the user in the browsers address field. Surely, we can stuff objects in the `flash` scope before redirecting, but Grails provides a `chain()` method specifically to pass objects between one action to another.

`chain()` and `redirect()` are very similar. Except for one thing. `chain()` allows the `model` to be preserved from one action to the next. The model can be accessed in the next controller/action through the `chainModel`. This property will exist only in actions following a call to `chain()`.

Below is an example with three actions that uses `chain()`.

```
def one = {
    chain(action:'two',   model:[a:new String('1')])
}
def two = {
    chain(action:'three', model:[b:new Integer(2)])
}
def three = {
    if( chainModel?.a == '1' && chainModel?.b == 2 )
       render('Success' + " " + chainModel.a + " " + chainModel.b)
    else
       render('Failure')
}
```

If we'd call action `one` from the browser, we should see *Success 1 2* and if we'd call `three` directly, we should see *Failure*.

Like the `redirect()` method, we can also pass parameters using `chain()`.

One thing worth mentioning here is that when we call `redirect()`, `chain()` or `render()` (wether rendering a simple text output or a view), the execution doesn't necessary stop there.

If we'd for example remove the `else` statement in action `three` and call action `one` again, we should see *Success 1 2Failure*. To stop the execution from going futher down in this example, we can either call `return render(...)` in the `if` block or on a separate line, like this:

```
def three = {
    if( chainModel?.a == '1' && chainModel?.b == 2 ){
       render('Success' + " " + chainModel.a + " " + chainModel.b)
       return
    }
    render('Failure')
}
```

This version is equivalent to the previous one.

13.3 Interceptors

Grails provides a mechanism called action interceptors. There are currently two types of interceptors that enables actions in a controller to be intercepted before or after they are executed.

The **beforeInterceptor** allows us to run logic before an action is executed, and `afterInterceptor` after an action is executed. An interceptor can be defined for all actions in the body of a controller like this:

```
def beforeInterceptor = {
    if(!session.loggedIn){
        redirect(controller:'other', action:'login')
        return false
    }
}
```

An interceptor will by default apply to all actions in a controller. If the interceptor returns `false`, the action will not be executed. In the above defined `beforeInterceptor` we say that `session.loggedIn` needs to be set for access to any action in the same controller[17].

We can however specify what actions an interceptor applies to using a `Map` and a reference to a method[18] to call:

```
def beforeInterceptor = [action:this.&authMethod,
                         except:['login', 'help'] ]

def authMethod(){
  if(!session.loggedIn){ redirect(action:'login'); return false }
}
```

The `beforeInterceptor` will now be invoked before all actions `except` for actions `login` and `help`. We also pass a method reference pointer to `authMethod()` to be called when the `except` condition is met.

If we change `except` to `only`, we 'll have the opposite effect, with the method being invoked only for actions `login`, and `help`.

[17] So we had to make redirect() in this example go to a different controller

[18] Instead of a closure which is an action and therefore accessible to the outside world

The **afterInterceptor** is defined in a similar way but will run after an action has executed and before a view has been rendered.

It also differs in that it takes two arguments. The resulting `model` as the first and `modelAndView` in the second argument, enabling us to manipulate our response based on those two properties:

```
/* Applies to all actions. Can be conditioned like beforeInt... */
def afterInterceptor = { model, modelAndView ->

       /* We can read and alter the resulting model */
       if(model.a > 100) {
            model.a = 0
            redirect(action:'login')
            return false
       }

       println "View to be rendered:     ${modelAndView.viewName}"

       modelAndView.viewName = "/differentDir/differentView"

       println "View to be rendered now: ${modelAndView.viewName}"
}
```

If an interceptor is likely to apply to more than one controller, we can **extend** and place it in a base controller or better, write a `Filter`. A filter can be applied to multiple controllers or URIs without the need to change the logic of each controller. We'll briefly discuss filters in Section 15.1 on p.108.

14 More on View

We demonstrated the view model earlier on page 72 where we showed how we separate the HTML output into a GSP. But that was very basic. We have yet to give a solid introduction to views and GSP in particular.

Grails supports the creation of views in a JSP or GSP. Although JSP is powerful, GSP is nontheless the evolution of the JSP view technology and the recommended way to work with views in Grails. Things that are difficult to do in JSP such as writing tag libraries are much simpler and accessible in GSP [17]. Graeme Rocher says, "GSP provides a mechanism for creating custom tags just as JSP does but without sacrificing any agility" [10].

GSP is also different from JSP because it fully takes advantage of the Groovy runtime environment capabilities, dynamic method dispatching, enabled support for the use of `GString` and offer an expressive syntax for maps and lists that makes it perfect and valid as its own view technology [11].

In Grails, there is a number of objects made available to a view. These include `request`, `session`, `flash`, `params` and `out` to name a few. The model passed from the controller, our domains and tag libraries are among many other things accessible in a view. Like JSP, we can add page directives, scriplets and comments in a GSP.

In this section we'll try to introduce and demonstrate key topics and concepts involving working with GSP.

14.1 Code in GSP

14.1.1 Page Directive

Graeme Rocher & a explains the page directive as "an instruction that appears at the top of a GSP that performs an action that the page relies on. As an example, it could set the content type, perform an import, or set a page property, which could even be container-specific" [11].

The `contentType` directive is used to set the type of the content of our response in a GSP allowing the GSP to output content other than HTML, such as XML or plain text.

We place the directive at the top of the page in the same way it's done in a JSP starting with `<%@`:

```
<%@ page contentType='text/xml; charset=UTF-8' %>
```

The `import` directive allows us to `import` classes into the page and is similar to the `import` statement in a Java or Groovy class. In the top of the page we can for example add:

```
<%@ page import='groovy.sql.Sql' %>
```

14.1.2 Groovy Code

Scripting

We declare a scriptlet block in a GSP the same way as done in a JSP, using `<% ... %>` syntax:

```
1 <html>
2     <body>
3         <%
4             def var = 'show'
5             if(var == 'show')
6                 out << 'Hello World!'
7         %>
8    </body>
9 </html>
```

Between lines 3-7 we've basically entered the world of Groovy and can type almost[19] any Groovy code we want. On line 6 we write to the response by using the output stream writer `out`[20].

We can write to the output stream writer in a number of ways:

```
1 <html><body>
2     Hello World!
3     <% print  'Hello World!' %>
4     <% out << 'Hello World!' %>
5     <%= 'Hello World!' %>
6     ${'Hello World!'}
7 </body><html>
```

Of course, to output text we just write it as it is and seen on line 2. The recommended way to output *variabled* content is by using a `GString`.

Commenting

There are a number of ways to use comment in a view. The most common and likely way is by using one of the following three:

```
<html><body>
     <%-- JSP  style comment  --%>
     %{-- GSP  style comment  --}%
     <!-- HTML style comment  -->
</body></html>
```

JSP and GSP style for commenting is used for comments that should not be sent in the rendered response whereas the HTML comment will be sent but

[19] We can't for example add methods in a view but similar effect can be achieved by the use of closures

[20] Note that in these examples, we're not interested in writing valid HTML markup code

something the browser chooses not to display, but which can be seen in the source code.

JSP and GSP style of commenting also complement each other. One of these styles of commenting should therefore be used more frequently than the other, like when commenting out single lines or commenting on code in a view. If we then want to comment out a larger block of code that for example already contains JSP style comments, we can use the GSP style to comment from start to the end of that block, thus eliminating the need for us to remove the JSP style comments for that code block.

14.2 Tags in GSP

We've mentioned the term tag a couple of times already and we used them in Section 12.7 on p.89 for rendering erorrs.

Tags are like view methods and are normally favored over the use of scriptlets because they provide a clean way to separate view and controller logic and allows us to create well formed markup code.

Grails has a wide range of custom tags built in for performing basic operations such as validating, looping, creating and so on. Grails also offers a way for us to create our own custom tags as we shall also see on p.102.

14.2.1 Grails Tag Library

As just mentioned, there are many tags that ship with Grails. They are in fact too many to all be listed here but below we try to cover a couple of them.

Each GSP tag requires the prefix `g:` so that it's recognized as a GSP tag.

Logical Tags

In this example, we demonstrate how the logical tags `if`, `else` and `elseif` can be used in a view:

```
<html><body>
    <g:if test="${true == false}">
        Not coming in here
    </g:if>
    <g:elseif test="${10 > 100 && params.var == 'show'}">
        Not coming in here
    </g:elseif>
    <g:else>
        Coming in here!
    </g:else>
</body></html>
```

Notice the condiction parameter `test` being passed in the `if` and `elseif` tags.

Iterative Tags

The are five default iterative tags: `each`, `while`, `collect`, `findAll` and `grep`. Most used ones are likely the `each` and `while` tag:

```
<%-- Declaring content for this example --%>
<% List lst = ['a', 'b', 'c'] %>
<html><body>
    <g:each in="${lst}" >
        ${it}
    </g:each>

    <g:each in="${lst}" var="e" status="i" >
        ${i}: ${e}
    </g:each>

    <% int j = 0 %>
    <g:while test="${j < lst.size()}">
        ${j}: ${lst[j++]}
    </g:while>
</body></html>
```

The `while` tag requires a condition to be passed in the parameter `test`. We are also responsible for both initializing and incrementing the variable `j`.

For the `each` tag, the parameters `status` and `var` are optional. However, if we use the parameter `status` then we have to use `var` as well.

Good practice is however to always specify `var` because of potential issues when doing nested calls to tags that also uses `it`. `status` is always optional and should be used if we need access to an iteration index.

Assignment Tags

In the previous example we declared `List lst` and `int j` using Groovy code even though this is recognized as a bad thing. Grails actually offer a way to set or alter the value of a variable using the `set` tag in a GSP page.

We use the `set` tag by passing the name of the variable to be set in parameter `var` and the value to assign to it in parameter `value`:

```
<g:set var="lst" value="['a','b','c']"/>
<g:set var="j" value="${0}"/>

<g:set var="str1" value="txt"/>
<g:set var="str2">
    <p>Text here will be assigned to str2
    including the left space and newlines.
    <g:if test="${true}">
        Add this to str2 as well.
    </g:if>
```

```
    </p>
</g:set>
```

In the last `set` tag we assigned to variable `str2` a text block by making use of the `body` of the tag and omitting passing the parameter `value`. Such assigning can prove to be useful when we need to declare an HTML block that's to be used or passed on to several places. Also notice how we've output text to `str2` conditionally.

Linking Tags

There are three linking tags: `link`, `createLink` and `createLinkTo`.

By using the **link** tag we can create an HTML `a` tag with `href` being set based on some predefined parameters we pass to it such as `action`, `controller`, `id`, `params` and a couple of others[21]. We can pass any HTML attributes we want as well, such as `class`, `style` and etc:

```
<g:link controller="search" action="results" class="_a"
        params="[o:10, max:50]" >10</g:link>
```

The **createLink** tag is very similar to `link`[22] and takes the same parameters but will create an URL only, without the `a` tag, `href` and other attributes. It can be useful if we for example want to create links the HTML way, or add a URL to a form, perhaps assign a URL to a JavaScript variable and etc:

```
<a class="_a" href="<g:createLink controller="search"
   action="results" params="[o:10, max:50]"/>" >10</a>
```

Here we've actually nested the tag inside the `href` attribute which is not very attractive. Many tags in Grails are actually *dynamic* in that they can be called as a method as well. The last example can and is usually written like this:

```
<a class="_a" href="${createLink(controller:'search',
   action:'results', params:[o:10, max:50])}" >10</a>
```

The call to `createLink()` here is done from within a `GString` and without the use of `g.` as a prefix. Tags can be called as methods from a controller as well, but then we have to use `g.` as a prefix, i.e.
`g.createLink(controller:'search', ...)`.

All the linking examples above will result in this same link:

```
<a class="_a" href="/search/results?o=10&max=50">10</a>
```

[21] Which are all optional and will by default link to current controller and action
[22] In fact, link calls createLink in its own implementation

We use the **createLinkTo** tag to link to resources within the *web-app* directory. Great for linking to JavaScript, CSS and image files and will generate the URL part only like the `createLink` tag. The most important parameters to use is `dir` and `file`:

```
<img   src="${createLinkTo(dir:'images', file:'mercedes.jpg')}"/>
<link href="${createLinkTo(dir:'css/mycss', file:'main.css')}"
      type="text/css" rel="stylesheet" />
```

The image *mercedes.jpg* needs to be available in directory *web-app/images/* and the stylesheet *main.css* in directory *web-app/css/mycss/*.

There are a number of other tags such as the form tags (`form`, `textField`, `select` and etc), `paginate`, `render`, `cookie`, `message` and so on.

There is also couple of useful Ajax tags that'll be introduced later in Section 15.2 on page 109.

14.2.2 Creating Custom Tags

Being able to create custom tags in JSP is a wonderful and powerful feature. Unfortunately, for all their wonderful attributes they are very complicated to implement. The reason for this are understandable and because it attempts to make possible creation of tags for every scenario we might want to use a tag [11].

Grails allows the creation of simple, logical and iterative custom tags through its simple dynamic tag library mechanism. "The benefit of Grails tags is that they require no additional configuration, no updating of TLD descriptors and can be autoreloaded at runtime without a server restart" [14].

To be able to create a tag we must first create a tag library. We do that by simply creating a Groovy class with a name that ends with `TagLib` and place it within the *grails-app/taglib/* directory. The class name must end with `TagLib` by convention. A tag library can also be created by running:

» **grails create-tag-lib** *demo*

A tag library class *DemoTagLib.groovy* was created for us under *grails-app/taglib/* as well as an integration test *DemoTagLibTests.groovy* under *test/itegration/*.

The `DemoTagLib` class is originally just an empty class declaration (`class DemoTagLib { }`). Let's add two tags to it:

Tag Library - DemoTagLib.groovy

```
import java.text.DecimalFormat as DF          // Type aliasing
class DemoTagLib {

    def outputNumber = { args ->
        /* Triple grouping + 2 decimals as default format */
        def format = args.format ?: "#,###.00"
        DF dF = new DF(format)

        out << dF.format(args.number)
    }

    def repeat = {args, body ->
        args.times?.toInteger().times{ nbr ->
            /* Call the body of the repeat tag */
            out << body( (args.index):nbr )
        }
    }
}
```

By looking at the implementation of `outputNumber`, we can see that it handles up to two parameters, `number` and `format`, the latter being optional and use the default value if not passed from the calling tag.

The second tag `repeat` handles two parameters and a `body`. The `body` is called and passed a key:value pair consisting of the incoming tag parameter `index` as a key and `nbr` of the iterative method `times()` as a value.

We can call the two tags from a view like this:

```
<html><body>
    <%-- Outputs: 1,234,567.90 --%>
    <g:outputNumber number="${1234567.899}" />
    <%-- Outputs: 1,23,45,68   --%>
    <g:outputNumber number="${1234567.899}" format="#,##"/>

    <g:repeat times="3" index="myIndex">
        On iteration: ${myIndex}
    </g:repeat>
</body></html>
```

`outputNumber` is a simple tag and doesn't work with a `body` like the `repeat` tag.

The `body` is the part between the opening and closing tag. Its type `GroovyPageTagBody extend Closure` and can therefore be invoked and passed objects to that are *directly* made available for use in the tag `body`. Note that the `body` of a tag is not executed on declaration but only when invoked, just like a `Closure`.

These tags are available under the `g:` tag library just like the tags that are bundled with Grails. By default, all our own custom tags are put in the `g` namespace.

To avoid naming conflicts with built in tags or for other reasons we can define a namespace of our own by adding this to the class `DemoTagLib`:

```
static namespace = 'dt'
```

We can now call the tags as `<dt:outputNumber ... />` and `<dt:repeat ... />`.

14.3 Template

A template is a GSP file located in the *grails-app/views/* directory or any sub-directory under that location and whose filename, by convention starts with an underscore, i.e. *_filename.gsp*.

A template can contain the same type of content as a normal view GSP and therefore allows us to split, separate and reuse pieces of our views inorder to avoid redundancy and achieve cleaner code. Several pages in an application might require the same piece of content, such as a footer:

Template - /common/_footer.gsp

```
<div>
    <span>$MyEnterprise© ${year}</span>
    <a href="/info/privacy">Privacy</a>
    <a href="/info/about"  >About</a>
    <a href="/info/help"   >Help</a>
    <a href="/info/contact">Contact</a>
</div>
```

We can now call the template from our views using the `render` tag and the relative path from *grails-app/views/* like this:

```
<html><body>
    ...
    <g:render template="/common/footer" model="[year:2009]"/>
</body></html>
```

Notice how we by using the `model` parameter of the `render` tag can pass objects to the template.

A template rendered from within a view by default has direct access to the `model` passed from the controller. So in the above example, we can instead pass `year` from a controller and omit passing it from the tag and still get the same result[23].

[23]Design wise, it would be better to store the year someplace else and not rely on it being passed at all. But something had to be passed to make this this example complete

We can also render templates from a controller using the `render()` method should we need to, perhaps in an Ajax response:

```
def someAction = {
    render template:'/common/footer', model:[year:2009]
}
```

14.4 Layout

Grails uses the SiteMesh decorator engine to help assemble the view for web pages. Decorators, in Grails known as layouts, are of good use when we require a consistent look across our many pages. A layout is a GSP file that's located in the *grails-app/views/layouts/* directory or any subdirectory under that location. You can think of layouts as views with placeholders we can pass certain content in to.

Take the footer example in the previous section. As it is now, it requires us to call the footer template from every view that want to use it. That's indeed redundant and if we decide to move the template to another direcoty or pass another object to it, we'd have to alter every view that uses the footer.

Let's start by creating a file named *demoLayout.gsp* in the just mentioned directory to illustrate their usefulness:

Template - /layouts/demoLayout.gsp

```
<html>
    <head>
       <title><g:layoutTitle default="My Webpage"/></title>
       <g:layoutHead/>
       <g:javascript library="prototype"/>
    </head>
    <body onLoad="${pageProperty(name:'body.onLoad')}">
       ABCD
       <g:layoutBody/>
       <g:render template="/common/footer" model="[year:2009]"/>
    </body>
</html>
```

Grails provides five tags to work with layouts. In *demoLayout.gsp*, we've used `layoutTitle`, `layoutHead`, `pageProperty` and `layoutBody`.

We make use of this layout by adding a `meta` tag to the header of our view[24]:

View - anyView.gsp

```
<html>
    <head>
        <meta name="layout" content="demoLayout"/>
        <title>MyTitle</title>
        <g:javascript src="main.js"/>
    </head>
    <body onLoad="doSomething()">
        EFGH
    </body>
</html>
```

When *anyView* has rendered, its content will be merged with *demoLayout*. The resulting output rendered to the user will be *equivalent* to:

"Resulting Output"

```
      <title>MyTitle</title>
      <g:javascript src="main.js"/>
      <g:javascript library="prototype"/>
    </head>
    <body onLoad="doSomething()">
      ABCD
      EFGH
      <g:render template="/common/footer" model="[year:2009]"/>
    </body>
</html>
```

By comparing the resulting output with *demoLayout.gsp* and *anyView.gsp* we

[24]There is another way as well

can notice several things.

The `layoutTitle` tag will insert the title from a view if there is one specified, otherwise use the default value supplied to it.

The `layoutHead` tag will merge the headers of a view and the layout. Some things are ignored, such as the layout `meta` and `title` tag from the view.

`pageProperty` gives access to attributes of certain tags from the view. In this example the `onLoad` attribute from the `body` tag.

The content in the `body` of the view will be inserted where the `layoutBody` is called in the layout.

There's also an `applyLayout` tag that's used to apply a layout to either a template, a piece of code supplied in the `body` or to an arbitrary URL.

15 Other

15.1 Filters

In Section 13.3 on p.95 we showed how the before and after interceptors can be used in a controller. They are however primarly useful when applied to a few controllers and become difficult to manage in a large application.

Filters allows us in a similar way to define logic to run `before` and `after` an action but offers in addition an easy way to apply that logic from a centralized place onto several controllers and actions, as well as for specific *URI*s. There is also an `afterView` interceptor to place logic in that will run after a view called from an action has been rendered.

To create a filter, we simply create a Groovy class with a name that ends with `Filters` and place it within the *grails-app/conf/* directory. The name must end with `Filters` by convention.

Let's create a class named *DemoFilters.groovy* in the just mentioned directory. Below we've defined the two filters `all` and `loginRequired`:

Filter - DemoFilters.groovy

```
1  class DemoFilters {
2   def filters = {
3
4     all(controller:'*', action:'*'){
5       before = {
6          // Perhaps count made incoming requests?
7       }
8
9       /* We can read and alter from the action resulting model */
10      after = { model ->
11         // Perhaps add a default object to the model?
12      }
13
14      afterView = {
15         // Perhaps measure total time to process a request?
16      }
17    }
18
19    loginRequired(controller:'user', action:'(settings|delete)'){
20      before = {
21        if(!session.loggedIn){
22           redirect(action:'login')
23           return false          // Abort execution of action
24        }
25      }
26    }
27  }
28 }
```

As seen in the class above, we can add several filters to the same filter class by defining them separately inside the `filters` closure.

By using the wildcard character `*` on line 4 for both `controller` and `action`, the filter `all` will be applied for all actions across all controllers.

If a wildcard is not present then we can use a regular expression as a matcher, as seen on line 19 where the `loginRequired` filter will be applied only for the two actions `settings` and `delete` in the `UserController`.

When actions `settings` and `delete` in the `UserController` are requested, both `before` interceptors in `all` and `loginRequired` will execute. The order we define the filters determines the order in which they are executed. Returning `false` as done on line 23 ensures that the action and remaining filters are not executed if there are any after.

For example, had the `loginRequired` filter been placed first, the `before` logic defined in `all` will not be executed for all incoming requests because of the possible breaking of `return false` in `loginRequired` and would lead to incorrect counting.

Note that filters do not have the same functionality as controllers. They have access to objects `actionName`, `controllerName`, `request`, `response`, `flash`, `session`, `params` and methods `render()` and `redirect()` to name a few. Access to the `g` tag library is for example not available in a filter class.

15.2 Ajax

Ajax or *Asynchronous JavaScript and XML* is the collective name for interrelated Web developing techniques used on the client side to create rich and interactive Web applications. Grails offers a number of built in tags to make working with Ajax easy and a pleasant experience.

We can make use of the Grails included JavaScript library Prototype to achieve this, but Grails supports many other JavaScript libraries such as Yahoo UI, Dojo Toolkit, jQuery, Google Webkit and etc as well. To use one of them instead, we just need to install their corresponding Grails plugin[25]. For example, to use YUI we do:

» **grails install-plugin** *yui*

In the header tag of our views we need to specify the library we're using by adding the following tag:

```
<g:javascript library="prototype"/>
```

To make use of the Grails included *effects* library Scriptaculous, we simply replace `prototype` with `scriptaculous` because Scriptaculous by default make use of Prototype. To use YUI instead, we replace `prototype` with `yui`[26].

[25] See http://grails.org/Plugins for more information
[26] Provided that the YUI plugin is installed as well

The Ajax tags provided by Grails are `remoteLink`, `remoteField`, `formRemote`, `submitToRemote`, and `remoteFunction`. They all have in common a set of optional parameters and JavaScript events that will fire depending on the outcome of the made Ajax request.

Parameter	*Takes*
method	HTTP method to use. Default is `POST`.
before	JavaScript code to execute before the Ajax request is sent.
update	Either a `String` with the element `id` to update on success(failure ignored) or a `Map` with the ids to update for success and failure states, i.e. `[success:'sId', failure:'fId']`.
onLoading	JavaScript code to execute when the Ajax request connection has opened. Perhaps to show a spinner?
onFailure	JavaScript code to execute when the Ajax response fails.
onSuccess	JavaScript code to execute when the Ajax response is recieved and before possible updates.
onComplete	JavaScript code to execute when the Ajax response is recieved and after possible updates. Will always run, no matter if the request succeeds or fails. Perhaps hide the spinner?

Table 5: **Some** common parameters for Grails Ajax tags

The **remoteLink** tag in its simplest form is very straight forward to use:

```
1  <html>
2      <head>
3          <g:javascript library="prototype" />
4      </head>
5  <body>
6      <g:remoteLink controller="demo"
7                    action="myAction"
8                    update="myId"    >Click here!</g:remoteLink>
9
10     <div id="myId"></div>
11 </body>
12 </html>
```

The `remoteLink` tag defined between line 6-8 will create an HTML link that will make an Ajax request to `http://.../demo/myAction` when clicked. The rendered response from `myAction` will then be inserted into the element that has the `id` defined in the `update` parameter. In this case, it'll be into the `div` on line 10.

Let's try a slightly more complex example by demonstrating the use of **formRemote** with a **remoteField** tag included in it by creating a simple login form.

The password field and submit button should not be present at start but sent from the server when we've found a matching `Member` in the database for the username provided and removed otherwise.

Once the password submitted is correct, we login the member and send back a reply that the login was successful:

View - /demo/login.gsp

```
1  <html><head><g:javascript library="prototype"/></head><body>
2
3  <%-- Ajax remote form --%>
4  <g:formRemote name="login_form" id="login_form"
5                url="[controller:'demo', action:'login']"
6                onSuccess="formSubmissionResponse(e)">
7
8       <%-- Ajax remote field for entering the username --%>
9      <g:remoteField name="username"
10                    controller="demo"
11                    action="checkUsername"
12                    update="password_and_submit" />
13
14     <div id="password_and_submit"></div>
15 </g:formRemote>
16
17 <%-- JavaScript code below is called onSuccess from above --%>
18 <script type="text/javascript">
19     function formSubmissionResponse(e){
20         var json = e.responseJSON
21         if( json.loginSuccess == true ){
22             Element.remove('login_form')
23             /* Member profile could've been recieved if sent &
24                then placed somewhere on the page from here */
25         }
26         alert(json.reply)
27     }
28 </script>
29 </body></html>
```

On line 9-12, the `remoteField` tag declared will create an HTML `input` tag of type `text` that will send its `value` to the specified remote location when it changes[27]. In this case to the `checkUsername` action in `DemoController`.

The response sent from the `checkUsername` action (see next page) will be placed into the `div` on line 14 above.

Once the password field and submit button are present, and the response from

[27] In reality, it's on the onKeyUp event (when a keyboard key is released)

the submitted form comes back successfully, the `onSuccess` event on line 6 will trigger the `formSubmissionResponse(e)` where `e` is of type `XMLHTTPResponse`.

On line 20 in `formSubmissionResponse()` the `responseJSON` sent from the `login` action in `DemoController` is read to decide whether to remove the `form` altogether. On line 26 the reply message also sent, is displayed to the user.

The `checkUsername` and `login` action in `DemoController`:

Controller - DemoController

```
1  class DemoController {
2
3    def checkUsername = {
4      def memb = Member.findByUsername(params.value)
5      if(memb)
6         /* Textbased response.
7            Available at client as e.responseText */
8         render {
9            input(type:'password', name:'password')
10           input(type:'submit',  value:'Login!')
11        }
12     render('')  // Include an empty response
13   }
14
15   def login = {
16     if(request.post){       /* Form submission */
17        def memb = Member.findByUsername(params.username)
18
19        if(memb && memb.password == params.password){
20           session.loggedIn = memb
21           /* JSON (JavaScript Object Notation) response.
22              Available at client as e.responseJSON */
23           render(contentType:'application/json'){
24              loginSuccess(true)
25              reply("You are now logged in ${memb.username}!") }
26         }
27         else{
28           render(contentType:'application/json'){
29              loginSuccess(false)
30              reply("Wrong username or password!") }
31         }
32     }
33     /* Will render the login view for non-post requests */
34   }
35 }
```

How the above code works should be pretty straight forward to figure out by now. Notice how we can `render()` HTML markup code as text on line 9-11. Also notice the use of a JSON response to transmit *structured data* over a network connection on line 23-25.

Left are the **submitToRemote** and **remoteFunction** tags which are used in a similar matter as the ones described above. The `remoteFunction` tag is however probably the most useful tag of all thse since it allows us to create a remote function to be used as we please.

15.3 Deploying

When the application is ready to be deployed onto a production system, it's recommended to do so with a Web Application Archive (WAR) file, which is basically a JAR file with a defined directory and file structure.

A WAR file can be deployed on any Java application server of our choice, such as Tomcat, JBoss or Weblogic. A WAR is something that we normally have to build ourselfs and can be a rather verbose task to perform.

Grails, being obsessed with making lives easier, doesn't bail out on us on the last step either. Grails allows us to create a fully working WAR file simply by running the following command:

» grails war

Output:

```
...
Environment set to production
...
Done creating WAR D:\exjobb\demoApp//-0.1.war
```

A WAR file *-0.1.war* was created for us under the top(root) directory. *0.1* is the application version defined on line 2 in the *application.properties* file. The WAR file can now be placed and used on the Java application server of our choice.

However, Grails already comes bundled with the powerful Jetty Server and we can of course use it to run our application from a WAR file by running:

» grails run-war

Output:

```
...
Environment set to development
...
Server running. Browse to http://localhost:8080/
```

As can be seen, the default environment and port is the same as for the `run-app` command. We can run with other settings in the same way described in Section 10.5 on p.65.

Other important things to consider when running on the JVM, is to run with the `-server` option and with good memory settings. This is achieved by setting the environment variable `JAVA_OPTS` before running the application.

On a Windows system:

```
>> set JAVA_OPTS=-server -Xms512m -Xmx512m -XX:MaxPermSize=99m
```

On a Unix like system:

```
>> export JAVA_OPTS="-server -Xms512m -Xmx512m -XX:MaxPermSize=99m"
```

For information on the meaning of these and other available JVM settings, please refer to
`http://java.sun.com/performance/reference/whitepapers/tuning.html`,
`http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp` and
`http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html`.

15.4 Other

There are many more features in Grails, but writing a 570 pages book (like *Definitive Guide to Grails, Second Edition*) is not the scope of this document. Hopefully you now have an idea of what Grails can do for Web application devlopment in a Java environment.

A Appendix

A.1 Files

A.1.1 grails-app/conf/DataSource.groovy

```
dataSource {
    pooled = true
    driverClassName = "org.hsqldb.jdbcDriver"
    username = "sa"
    password = ""
}
hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
    cache.provider_class='com.opensymphony.oscache.hibernate.OSCacheProvider'
}
// environment specific settings
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'create-drop','update'
            url = "jdbc:hsqldb:mem:devDB"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:file:prodDb;shutdown=true"
        }
    }
}
```

References

[1] Venkat Subramaniam
Programming Groovy. Dynamic Productivity for the Java Developer
The Pragmatic Programmers 2008

[2] Ian F. Darwin
Groovy, Java's New Scripting Language
`http://onjava.com/pub/a/onjava/2004/09/29/groovy.html`

[3] Koenig with Glover, King, Laforge and Skeet
What Groovy Can Do For You
`http://www.developer.com/open/article.php/10930_3657751_1`

[4] Christopher M. Judd, Joseph Faisal Nusairat and James Shingler
Beginning Groovy and Grails. From Novice to Professional
Apress 2008

[5] Jason Rudolph
Geting Started With Grails
InfoQ 2006

[6] *Groovy User Guide*
`http://groovy.codehaus.org/User+Guide`

[7] Scott Davis
Groovy Recipes, Greasing the Wheels of Java

[8] *Java Specification Requests. The Groovy Programming Language Java*
`http://jcp.org/en/jsr/detail?id=241`

[9] Youssef El Messaoudi, Gaya Kessler, Marco Kuiper, Jaap Mengers, Bart van Zeeland
Getting Groovy in an SOA

[10] Graeme Rocher
Definitive Guide to Grails
Apress 2006

[11] Graeme Rocher
Definitive Guide to Grails, Second Edition
Apress 2009

[12] *Your way to Groovy*
`http://www.developer.com/lang/article.php/3657751`

[13] *Duck typing*
`http://en.wikipedia.org/wiki/Duck_typing`

[14] *Dynamic Tag Libraries*
`http://grails.org/Dynamic+Tag+Libraries`

[15] Dierk König
Groovy In Action
Manning 2007

[16] *Interface Metaclass*
http://groovy.codehaus.org/api/groovy/lang/MetaClass.html

[17] Glen Smith, Peter Ledbrook
Grails In Action Mannig 2009

[18] *Java EE at a Glance*
http://java.sun.com/javaee

[19] *Forbes - VMware buying SpringSource*
http://www.forbes.com/feeds/afx/2009/08/10/afx6762406.html

[20] Bashar Abdul Jawad
Groovy and Grails Recipes Apress 2008

[21] *Grails Reference 1.0.3*
http://grails.org/doc/1.0.3

[22] *Spring MVC Integration*
http://grails.org/Developer+-+Spring+MVC+Integration

[23] *What is Hibernate?*
http://www.datadirect.com/developer/jdbc/hibernate/what-hib/index.ssp

[24] *The future of J2EE*
http://news.cnet.com/The-future-of-J2EE/2010-1001_3-5106960.html

[25] *Whatever Happened to Obect-Oriented Databases?*
http://www.leavcom.com/db_08_00.htm

[26] *Spidermans grandfather in Spiderman 1*

[27] *From Java to Groovy in a few easy steps*
http://groovy.dzone.com/news/java-groovy-few-easy-steps

[28] *Controllers Scopes*
http://grails.org/Controllers+-+Controller+Scopes