Table of Contents

This chapter explains the starting point with regards to authentication mechanisms in RESTful applications, as well as a comparision between the existing alternatives.

Introduction

In the Grails framework, the option to perform authentication and authorisation is the Spring Security Core plugin. Before going deeper into it, let’s describe first the foundations of the project.

The Groovy programming language

Groovy is developed to be a feature rich Java friendly programming language. The idea is to bring features we can find in dynamic programming languages like Python, Ruby to the Java platform. The Java platform is widely supported and a lot of developers know Java.

The Groovy web site gives one of the best definitions of Groovy: Groovy is an agile dynamic language for the Java Platform with many features that are inspired by languages like Python, Ruby and Smalltalk, making them available to Java developers using a Java-like syntax.

Groovy is closely tied to the Java platform. This means Groovy has a perfect fit for Java developers, because we get advanced language features like closures, dynamic typing and the meta object protocol within the Java platform. Also we can reuse Java libraries in our Groovy code.

Groovy is often called a scripting language, but this is not quite true. We can write scripts with Groovy, but also full blown applications. Groovy is very flexible.

Features

- Dynamic language
- Duck typing
- It is compiled into the Java Virtual Machine • Support closures
- Support operators-overload

Differences with Java

Groovy tries to be as natural as possible for Java developers. Here we list all the major differences between Java and Groovy.

Default imports

All these packages and classes are imported by default, i.e. you do not have to use an explicit import statement to use them:

```
java.io.*
java.lang.*
java.math.BigDecimal
java.math.BigInteger
java.net.*
java.util.*
groovy.lang.*
groovy.util.*
```

Multi-methods

In Groovy, the methods which will be invoked are chosen at runtime. This is called runtime dispatch or multi-methods. It means that the method will be chosen based on the types of the arguments at runtime. In Java, this is the opposite: methods are chosen at compile time, based on the declared types.

The following code, written as Java code, can be compiled in both Java and Groovy, but it will behave differently:

```
int method(String arg) {
    return 1;
}
int method(Object arg) {
    return 2;
}
Object o = "Object";
int result = method(o);
```

In Java, you would have:

```
assertEquals(2, result);
```

Whereas in Groovy:

```
assertEquals(1, result);
```

That is because Java will use the static information type, which is that o is declared as an `Object`, whereas Groovy will choose at runtime, when the method is actually called. Since it is called with a `String`, then the String version is called.

Array initialisers

In Groovy, the `{ ...¬ }` block is reserved for closures. That means that you cannot create array literals with this syntax:

```
int[] array = { 1, 2, 3}
```

You actually have to use:

```
int[] array = [1,2,3]
```

Package scope visibility

In Groovy, omitting a modifier on a field doesn't result in a package-private field like in Java:

```
class Person {
    String name
}
```

Instead, it is used to create a property, that is to say a private field, an associated getter and an associated setter.

It is possible to create a package-private field by annotating it with `@PackageScope`:

```
class Person {
    @PackageScope String name
}
```

ARM blocks

ARM (Automatic Resource Management) block from Java 7 are not supported in Groovy. Instead, Groovy provides various methods relying on closures, which have the same effect while being more idiomatic. For example:

```
Path file = Paths.get("/path/to/file");
Charset charset = Charset.forName("UTF-8");
try (BufferedReader reader = Files.newBufferedReader(file, charset))
{
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

can be written like this:

```
new File('/path/to/file').eachLine('UTF-8') {
   println it
}
```

or, if you want a version closer to Java:

```
new File('/path/to/file').withReader('UTF-8') { reader ->
   reader.eachLine {
       println it
   }
}
```

Inner classes

The implementation of anonymous inner classes and nested classes follows the Java lead, but you should not take out the Java Language Spec and keep shaking the head about things that are different. The implementation done looks much like what we do for `groovy.lang.Closure`, with some benefits and some differences. Accessing private fields and methods for example can become a problem, but on the other hand local variables don't have to be final.

- Static inner classes

Here's an example of static inner class:

```
class A {
    static class B {}
}

new A.B()
```

The usage of static inner classes is the best supported one. If you absolutely need an inner class, you should make it a static one.

- Anonymous Inner Classes

```
import java.util.concurrent.CountDownLatch
import java.util.concurrent.TimeUnit

CountDownLatch called = new CountDownLatch(1)

Timer timer = new Timer()
timer.schedule(new TimerTask() {
    void run() {
        called.countDown()
    }
}, 0)

assert called.await(10, TimeUnit.SECONDS)
```

- Creating Instances of Non-Static Inner Classes

In Java you can do this:

```
public class Y {
    public class X {}
    public X foo() {
        return new X();
    }
    public static X createX(Y y) {
        return y.new X();
    }
}
```

Groovy doesn’t support the `y.new X()` syntax. Instead, you have to write `new X(y)`, like in the code below:

```
public class Y {
    public class X {}
    public X foo() {
        return new X()
    }
    public static X createX(Y y) {
        return new X(y)
    }
}
```

Caution though, Groovy supports calling methods with one parameter without giving an argument. The parameter will then have the value null. Basically the same rules apply to calling a constructor. There is a danger that you will write `new X()` instead of `new X(this)` for example. Since this might also be the regular way we have not yet found a good way to prevent this problem.

Lambdas

Java 8 supports lambdas and method references:

```
Runnable run = () -> System.out.println("Run");
list.forEach(System.out::println);
```

Java 8 lambdas can be more or less considered as anonymous inner classes. Groovy doesn’t support that syntax, but has closures instead:

```
Runnable run = { println 'run' }
list.each { println it } // or list.each(this.&println)
```

GStrings

As double-quoted string literals are interpreted as `GString` values, Groovy may fail with compile error or produce subtly different code if a class with `String` literal containing a dollar character is compiled with Groovy and Java compiler.

While typically, Groovy will auto-cast between `GString` and `String` if an API declares the type of a parameter, beware of Java APIs that accept an Object parameter and then check the actual type.

String and Character literals

Singly-quoted literals in Groovy are used for `String`, and double-quoted result in `String` or `GString`, depending whether there is interpolation in the literal.

```
assert 'c'.getClass()==String
assert "c".getClass()==String
assert "c${1}".getClass() in GString
```

Groovy will automatically cast a single-character `String` to char when assigning to a variable of type `char`. When calling methods with arguments of type char we need to either cast explicitly or make sure the value has been cast in advance.

```
char a='a'
assert Character.digit(a, 16)==10 : 'But Groovy does boxing'
assert Character.digit((char) 'a', 16)==10

try {
  assert Character.digit('a', 16)==10
  assert false: 'Need explicit cast'
} catch(MissingMethodException e) {
}
```

Groovy supports two styles of casting and in the case of casting to `char` there are subtle differences when casting a multi-char strings. The Groovy style cast is more lenient and will take the first character, while the C-style cast will fail with exception.

```
// for single char strings, both are the same
assert ((char) "c").class==Character
assert ("c" as char).class==Character

// for multi char strings they are not
try {
  ((char) 'cx') == 'c'
  assert false: 'will fail - not castable'
} catch(GroovyCastException e) {
}
assert ('cx' as char) == 'c'
assert 'cx'.asType(char) == 'c'
```

Behaviour of ==

In Java == means equality of primitive types or identity for objects. In Groovy == translates to `a.compareTo(b)==0`, if they are `Comparable`, and `a.equals(b)` otherwise. To check for identity, there is `is`. E.g. `a.is(b)`.

Different keywords

There are a few more keywords in Groovy than in Java. Don’t use them for variable names etc.

- `in`.
- `trait`.

Syntax

- Indentation is not mandatory
- Semicolons are not mandatory. They can be used for writing more than one statement in a line
- It uses a Java-like bracket syntax

In Listing 1. Hello world in Groovy we can see how a *hello world* looks like in Groovy.

Listing 1. Hello world in Groovy

```
println "Hello world!"
```

Grails framework

OAuth 2.0

Asciidoctor

Similar solution #1

Similar solution #2

Comparision