Table of Contents

Dedication

I would like to start offering my gratitude to my parents, as they funded my studies and my cost of living during the first years of the university.

To Laura, for constantly chasing me to finish this work and get the degree.

To all the companies and people that I have worked with across my 14+ years of experience.

To the open-source community, for being so grateful with the job I have done, and encouraging me to continue working on it.

To Alejandro Calderón, for allowing me to complete this work with such short notice and his dedication towards my success in doing it.

Abstract

Nowadays, most of the software is built using the REST architecture. In addition to that, developers are creating applications where the backend is nomally serving JSON over HTTTP, whereas the front-end is a separate application built in Javascript, that consumes such REST endpoints.

One of the restrictions of REST is that the systems must be **stateless**, in such a way that the client must transfer its state to the server on every request. In this scenario, doing authentication and authorisation in a stateless way is tricky, as the vast majority of the existing solutions are stateful implementations based on the HTTP session.

For this reason was created **Spring Security REST**, a plugin for the Grails framework based on Spring Security that enables developers to secure their applications in a pure REST, stateless way.

Chapter 1. Introduction

This chapter describes the goals that motivated the development of this software. It also specifies the format of this document.

1.1. Motivation

Back in 2012, I was researching about the death era of monolithic applications, and how the industry was moving towards a separation of front-end and backend. I talked to the community about this research in the conference session *"Single-page applications and Grails"* [spi-greach], given at Greach Madrid in January. 2013.

Traditionally, software developers have been used to create applications where the views and the business logic, although probably separated in software layers according to best practices and design patterns, are packaged together in the resulting application deployable artifact. This approach has several disadvantages:

- **Front-end** and **backend** developers have to work in the same project, using probably a server-side based framework such as Spring MVC for Java EE or Rails for Ruby. This will require so called full-stack developers, as the views are not simply HTML, but a framework-specific view templating technology.
- **Deploying** and **scaling** such monolithic applications is complicated, as it all have to consider the whole package.
- **UI changes** cannot be made and deployed independently of the business logic, due to the way of packaging and deployment. This reduces or even prevents doing frequent interface changes to improve conversation, A/B testing, etc.

Those problems were not significant to the development industry because the majority of the developers were considering themselves **full-stack developers**: proficient in doing backend and front-end. However, a technology was created and it changed the game rules: **Node JS**.

Node JS not only enabled developers for the first time to use the same language for front-end and backend (Javascript), but also set the foundation for a large number of tools, such as Grunt or Gulp as bulding tools, Bower as dependency manager and Yeoman as project creation and scaffolding tool. Those projects attracted a large number of developers that were highly skilled in HTML5, CSS3 and Javascript, and for the first time in forever had a specific toolset where no backend was involved.

On the other hand, backend development was experiencing the rise of the new buzzword of the moment: **microservices**. After several years since REST architecture was first introduced, and more and more teams trying to adopt such architecture with relative success, the industry finally agreed on a preferred way of building the business logic: they should be REST API's, producing and consuming JSON over HTTP.

The last necessary contributors in this shift were the frontend development frameworks, like Ember but specially Angular JS (built by Google), that effectively enabled the possibility of creating applications where the UI was entirely executed in the browser, and the communication with the server was using the mentioned JSON over HTTP.

The API's that the backends were converted into had a special requirement per the definition of REST (*REpresentational State Transfer*): they had to be **stateless**.

In my job at that moment, as Web Architect, I introduced the necessary changes in the company to break down a existing set of monolithic Grails-based applications, into Grails REST backends plus Angular JS front-ends.

In another conference in December 2013, the Groovy & Grails eXchange (GGX) in London, I was talking to the audience about this idea to separate front-ends and backends ([spi-ggx]), and eventually one of the attendees asked a relevant question:

> How do you handle authentication/authorization with this setup?.
>
> — @javazquez, 12th December 2013

More or less, developers managed to create stateless business logic, using the front-end frameworks to keep client state and ensuring that such state was sent to the server on each and every round-trip. However, authentication and authorisation was different. **Security** in a web application has traditionally been a stateful service: it heavily relies on the HTTP Session, which itself represents a state in the server side. Therefore, there was a friction when applying stateless restrictions to operations like login or logout.

Therefore, to not only answer that attendee but also provide a potential solution, I continued my research on this subject that ended up in the development of **Spring Security REST**: a plugin for Grails framework, on top of Spring Security, that offered developers a way to implement authentication and authorization in a stateless way, as well as it made the necessary transformations over core Spring Security to make it more REST-friendly.

1.2. Goals

The goals of this project are closely related to the motivations described above, and are the following:

1. Provide an implementation for the Grails framework.
2. Make core Spring Security, which only offered form-based authentication using the HTTP Session, more REST-friendly.
3. Offer developers a way to implement authentication and authorization in a stateless way.
4. Analise, design, implement and test a robust solution, stable and highly covered by tests.
5. Contributing back to the community by creating an open-source solution.

This document describes in detail all the stages of the process followed to achieve the mentioned goals.

1.3. Rest of the document

This document is structured in the following way:

1. **Introduction**. This chapter explains the motivations that originated the need to create this project, the goals that were pretended to achieve with it, as well as this document structure.
2. **State of the art**. In this chapter an evaluation of the starting point is performed, as well as a comparision of the existing solutions, to conclude that none is able to achieve all the goals.
3. **Analysis, design, development and deployment**. Here are described in detail the requirements that the system must complete, including documentation about the use cases.
4. **Evaluation**. It contains an analysis of the project success, based on different metrics.
5. **Planning and budget**. In this chapter it is described the planning of the different tasks performed to complete the project. It also contains a budget estimation, considering the cost of the resources used to develop the project.
6. **Conclusions and future work**. It is explained how the goals have been met, what is the result of having applied the software development process, which are the personal improvements obtained, as well as possible future improvements.
7. **Appendices**. It contains the official software documentation.

Chapter 2. State of the art

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

2.1. 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.

2.1.1. 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!"
```

2.2. Grails framework

2.3. OAuth 2.0

2.4. Asciidoctor

2.5. Similar solution #1

2.6. Similar solution #2

2.7. Comparision

Chapter 3. Analysis, design, development and deployment

3.1. Introduction

3.2. Analysis

3.3. Design

3.4. Development

3.5. Deployment

Chapter 4. Evaluation

Chapter 5. Planning and budget

5.1. Planning

5.2. Budget

Chapter 6. Conclusions and future improvements

6.1. Conclusions

6.1.1. Product

6.1.2. Process

6.1.3. Personal

6.2. Personal improvements

Appendix A: Documentation

Bibliography

- [spi-greach] Álvaro Sánchez-Mariscal. Single-page applications and Grails. Greach Madrid. January 2013.
- [revolution] Álvaro Sánchez-Mariscal. Embrace the front-end revolution. Codemotion Rome. March 2014.
- [spi-ggx] Álvaro Sánchez-Mariscal. Developing SPI applications using Grails and AngularJS. Groovy & Grails eXchange London. Decembre 2013.