Table of Contents

Colophon

Álvaro Sánchez-Mariscal Arnaiz

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 [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:

Listing 1. Default imports in Groovy

```
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:

Listing 2. Method dispatching code in Java/Groovy

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

In Java, you would have:

Listing 3. Method dispatch in Java

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

Whereas in Groovy:

Listing 4. Method dispatch 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:

Listing 5. Invalid syntax to declare an array in Groovy

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

You actually have to use:

Listing 6. Array declaration in Groovy

```
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:

Listing 7. Default visibility in Groovy

```
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`:

Listing 8. Package scope visibility in Groovy

```
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:

Listing 9. Automatic resource management in Java

```
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:

Listing 10. Using Groovy closures as an ARM alternative

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

or, if you want a version closer to Java:

Listing 11. ARM in Groovy, alternative edition

```
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:

Listing 12. Static inner class in Groovy

```
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

Listing 13. Anonymous inner class in Groovy

```
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:

Listing 14. Non-static inner classes in Groovy

```
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:

Listing 15. Instantiating non-static inner classes in Groovy

```
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:

Listing 16. Lambdas in Java

```
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:

Listing 17. Closures in Groovy as an alternative to Java lambdas

```
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.

Listing 18. String definitions in Groovy

```
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.

Listing 19. Groovy String to char conversion

```
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.

Listing 20. String casting in Groovy

```
// 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`.

2.1.2. Examples

Listing 21. Hello world in Groovy

```
println "hello world" // (1)
```

(1) The simplest Groovy code is a script. As this is going to be compiled to Java anyways, the Groovy compiler takes care of all the noise required by Java: it generates a class with a `main` method calling your code.

Listing 22. Basic syntax elements in Groovy

```
class Team { // (1)

    String name // (1)
    BigDecimal budget
    List<Player> squad = [] // (2)

}

class Player { String name, int age }

Team realMadrid = new Team(name: 'Real Madrid CF', players:[new Player(name:
'Cristiano Ronaldo'), ...]) (3)
realMadrid.budget = 100_000_000 // (4)

def youngPlayers = realMadrid.players.collect { it.age < 20} // (5)
```

(1) Groovy applies common sense default for visibilities: public for classes and methods and private for attributes

(2) There is native grammar syntax for lists (`[a,b,c]`) and maps (`[a: b, c: d]`)

(3) Groovy enhances constructors with named parameters

(4) Getters and setters are auto-generated and hidden. Property-access assignments are calling implicit setters. It also honours Java 7 Project Coin's features such as numeral literal formatting

(5) Groovy super-vitamines the JDK with many methods in the collections, file, etc API's with the so called GDK.

2.2. Grails framework

Java web development as it stands today is dramatically more complicated than it needs to be. Most modern web frameworks in the Java space are over complicated and don't embrace the Don't Repeat Yourself (DRY) principles.

Dynamic frameworks like Rails, Django and TurboGears helped pave the way to a more modern way of thinking about web applications. Grails builds on these concepts and dramatically reduces the complexity of building web applications on the Java platform. What makes it different, however, is that it does so by building on already established Java technologies like Spring and Hibernate.

Grails [grails] is a full stack framework and attempts to solve as many pieces of the web development puzzle through the core technology and its associated plugins. Included out the box are things like:

- An easy to use Object Relational Mapping (ORM) layer built on Hibernate
- An expressive view technology called Groovy Server Pages (GSP)
- A controller layer built on Spring MVC
- An interactive command line environment and build system based on Gradle
- An embedded Tomcat container which is configured for on the fly reloading
- Dependency injection with the inbuilt Spring container
- Support for internationalization (i18n) built on Spring's core MessageSource concept
- A transactional service layer built on Spring's transaction abstraction

All of these are made easy to use through the power of the Groovy language and the extensive use of Domain Specific Languages (DSLs)

2.2.1. Domains

Grails has an abstract domain layer called GORM: Grails Object Relational Mapping. Despite of it's name, it has support for multiple data storage systems, bot SQL and No-SQL, such as Hibernate (for relational databases), MongoDB, CouchDB, Neo4j and Redis.

This is an example of how Grails domains work:

Listing 23. Grails GORM examples

```
class Person {
    String name
    Integer age
    Date lastVisit

    static hasMany = [pets: Pet]
}

class Pet {
    String name

    static belongsTo = [owner: Person]
}

//Create
def p = new Person(name: "Fred", age: 40, lastVisit: new Date())
p.save()

//Read
def fred = Person.get(1)
assert 'Fred' == fred.name

//Update
def bob = Person.findByName("Fred")
bob.name = "Bob"
bob.save(flush: true)
assert 'Bob' == Person.get(1).name

//Delete
bob.delete()

//Relationships
def john == new Person(name: "John", age: 40, lastVisit: new Date()).addToPets
(name: "floppy").save()
assert Pet.get(1).name == "floppy"
```

2.2.2. The web layer

Controllers

A controller handles requests and creates or prepares the response. A controller can generate the response directly or delegate to a view.

Listing 24. Grails Controller examples

```
import grails.converters.JSON

class PersonController {

    /** Accessible through /person */
    def index() {
        List<Person> people = Person.list()

        //Generates a JSON response directly
        render people as JSON
    }

    /** Accessible through /person/show/123 */
    def show(Long id) {
        Person person = Person.get(id)
        if (person) {
            //Delegates to /grails-app/views/person/show.gsp, where the model is
the map [person: person]
            return [person: person]
        } else {
            response.sendError(404)
        }
    }
}
```

Views

Groovy Servers Pages (or GSP for short) is Grails' view technology. It is designed to be familiar for users of technologies such as ASP and JSP, but to be far more flexible and intuitive.

GSP's live in the `grails-app/views` directory and are typically rendered automatically (by convention) or with the render method such as:

```
render(view: "index")
```

A GSP is typically a mix of mark-up and GSP tags which aid in view rendering.

Although it is possible to have Groovy logic embedded in your GSP, the practice is strongly discouraged. Mixing mark-up and code is a bad thing and most GSP pages contain no code and needn't do so.

A GSP typically has a "model" which is a set of variables that are used for view rendering. The model is passed to the GSP view from a controller. Given the `show` action from the controllers example, a GSP could look like this:

Listing 25. Grails GSP example

```
<html>
  <body>
    <h1>${person.name}</h1>
    <p>Pets:</p>
    <ul>
      <g:each in="${person.pets}" var="pet">
        <li>${pet.name}</li>
      </g:each>
    </ul>
  </body>
</html>
```

2.3. OAuth 2.0

The OAuth 2.0 authorization framework [oauth2] enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849.

2.3.1. Roles

OAuth defines four roles:

1. **Resource owner**: an entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.
2. **Resource server**: the server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
3. **Client**: an application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).
4. **Authorization server**: the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

The interaction between the authorization server and resource server is beyond the scope of the specification. The authorization server may be the same server as the resource server or a separate entity. A single authorization server may issue access tokens accepted by multiple resource servers.

[oauth flow] | *oauth-flow.png*

Figure 1. OAuth 2.0 flow

1. The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via the authorization server as an

intermediary.

2. The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.
3. The client requests an access token by authenticating with the authorization server and presenting the authorization grant.
4. The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.
5. The client requests the protected resource from the resource server and authenticates by presenting the access token.
6. The resource server validates the access token, and if valid, serves the request.

2.4. Asciidoctor

This document has been produced using Asciidoctor.

Asciidoctor[asciidoctor] is a fast text processor and publishing toolchain for converting AsciiDoc content to HTML5, DocBook 5 (or 4.5) and other formats. Asciidoctor is written in Ruby, packaged as a RubyGem and published to RubyGems.org.

AsciiDoc belongs to the family of lightweight markup languages, the most renowned of which is Markdown. AsciiDoc stands out from this group because it supports all the structural elements necessary for drafting articles, technical manuals, books, presentations and prose. In fact, it's capable of meeting even the most advanced publishing requirements and technical semantics.

Here's a basic example of an AsciiDoc document:

Listing 26. Sample AsciiDoc document

```
= Introduction to AsciiDoc
Doc Writer <doc@example.com>

A preface about http://asciidoc.org[AsciiDoc].

== First Section

* item 1
* item 2

[source,ruby]
puts "Hello, World!"
```

2.5. Gradle

Gradle [gradle] is the build system used to produce this document in different formats, such as PDF and HTML.

Gradle provides:

- A very flexible general purpose build tool like Ant.
- Switchable, build-by-convention frameworks a la Maven. But we never lock you in!
- Very powerful support for multi-project builds.
- Very powerful dependency management (based on Apache Ivy).
- Full support for your existing Maven or Ivy repository infrastructure.
- Support for transitive dependency management without the need for remote repositories or pom.xml and ivy.xml files.
- Ant tasks and builds as first class citizens.
- Groovy build scripts.
- A rich domain model for describing your build.

You run a Gradle build using the `gradle` command. The `gradle` command looks for a file called `build.gradle` in the current directory. This `build.gradle` file is called a build script, although strictly speaking it is a build configuration script. The build script defines a project and its tasks.

Listing 27. Gradle build script example

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
```

Listing 28. Output of the previous build script

```
$ gradle -q hello
Hello world!
```

2.6. Similar solution: Spring Security

Spring Security is an authentication and access-control framework. It is the de-facto standard for securing Spring-based applications. It has the following features:

- Comprehensive and extensible support for both Authentication and Authorization.
- Protection against attacks like session fixation, clickjacking, cross site request forgery, etc.
- Servlet API integration.
- Optional integration with Spring Web MVC.

At an authentication level, Spring Security supports a wide range of authentication models. Most of these authentication models are either provided by third parties, or are developed by relevant standards bodies such as the Internet Engineering Task Force. In addition, Spring Security provides its own set of authentication features. Specifically, Spring Security currently supports authentication integration with all of these technologies:

- HTTP BASIC authentication headers (an IETF RFC-based standard).
- HTTP Digest authentication headers (an IETF RFC-based standard).
- HTTP X.509 client certificate exchange (an IETF RFC-based standard).
- LDAP (a very common approach to cross-platform authentication needs, especially in large environments).
- Form-based authentication (for simple user interface needs).
- OpenID authentication.
- Authentication based on pre-established request headers (such as Computer Associates Siteminder).
- JA-SIG Central Authentication Service (otherwise known as CAS, which is a popular open source single sign-on system).
- Transparent authentication context propagation for Remote Method Invocation (RMI) and HttpInvoker (a Spring remoting protocol).
- Automatic "remember-me" authentication (so you can tick a box to avoid re-authentication for a predetermined period of time).

- Anonymous authentication (allowing every unauthenticated call to automatically assume a particular security identity).
- Run-as authentication (which is useful if one call should proceed with a different security identity).
- Java Authentication and Authorization Service (JAAS)
- JEE container autentication (so you can still use Container Managed Authentication if desired).
- Kerberos.

It also integrates with a variety of technologies via third party implementations, such as Grails.

2.6.1. The Spring Security Core Grails plugin

For the Grails framework, there is an official plugin for authentication and authorisation: the Spring Security Core Grails plugin [ss-core]. It basically simplifies the usage of Spring Security in a Grails application.

Although there are alternative plugins for security services, we can consider Spring Security Core the de-facto option as it is the official and supported plugin.

2.7. Conclusion

Spring Security is not a viable solution due to 2 problems: the HTTP session usage and the lack of REST support.

2.7.1. HTTP Session

Spring Security is heavily based on the HTTP session. No matter what authentication mechanism the developer has configured, the core authentication component, the `SecurityContext` will end up being backed by the HTTP session.

An HTTP session is a sequence of network request-response transactions. An HTTP client initiates a request by establishing a TCP connection to a particular port on a server, An HTTP server listening on that port waits for a client's request message. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a message of its own. The body of this message is typically the requested resource, although an error message or other information may also be returned.

A session token is a unique identifier that is generated and sent from a server to a client to identify the current interaction session. The client usually stores and sends the token as an HTTP cookie and/or sends it as a parameter in GET or POST queries. The reason to use session tokens is that the client only has to handle the identifier—all session data is stored on the server (usually in a database, to which the client does not have direct access) linked to that identifier.

Particularly in Java-based web applications, HTTP sessions are in-memory data structures. There are alternative implementations where the HTTP Session is stored in databases or distributed caches. However, they require additional configuration; there are also some caveats when using them, so in-memory storage is the option for most of the users.

This state on the server side has the following problems:

1. Makes the solution stateful, thus violating a basic principle of a RESTful architecture: services must be stateless, where is the client transferring its state to the server on every roundtrip.
2. It doesn't play well with microservices as it makes the solution much more difficult to scale, as the HTTP session is physically stored in the application server instance that first served that particular user. For this to work, the web/proxy servers or load balancers in front of the application server must use sticky sessions.

A sticky session is bound (*sticky*) to a particular application server instance, so that all the requests from that user will be routed directly to such instance. In order to do so, proxies/balancers store a cookie when the first request is sent to an instance. In following requests, the balancer will be able to route them to this particular instance by reading the cookie.

In a microservice architecture, it should be possible to spin up multiple instances of a microservice to cope with load increments. And any instance should be able to respond to any request. With sticky HTTP sessions, this is not possible.

2.7.2. Lack of REST support

In REST, data is normally sent and received in JSON format. For an authentication request, this is how we would like it to work:

Listing 29. Ideal authentication request in a RESTful architecture

```
POST /login HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
    "username": "admin",
    "password": "secret"
}
```

However, this is how it works in Spring Security:

Listing 30. Actual authentication request in Spring Security

```
POST /j_spring_security_check HTTP/1.1
Host: api.example.com
Content-Type: application/x-www-form-urlencoded

j_username=admin&j_password=secret
```

As you can see, it's far from RESTful.

For all the reasons explained above, Spring Security is not a viable solution, therefore creating Spring Security REST is an actual need for the development community.

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: Official Documentation

A.1. Introduction to the Spring Security REST plugin

The Spring Security REST Grails plugin allows you to use Spring Security for a stateless, token-based, RESTful authentication.

This plugin depends on Spring Security Core 2.x. Make sure your application is compatible with that version first. There is a feature request, that may be addressed in the future if there is enough community interest / love :)

This plugin is only for Grails 2.x, and it requires at least Java 7.

The default behaviour of Spring Security is to store the authenticated principal in the HTTP session. However, in a RESTful scenario, we need to make sure our server is stateless.

The typical flow could be the following:

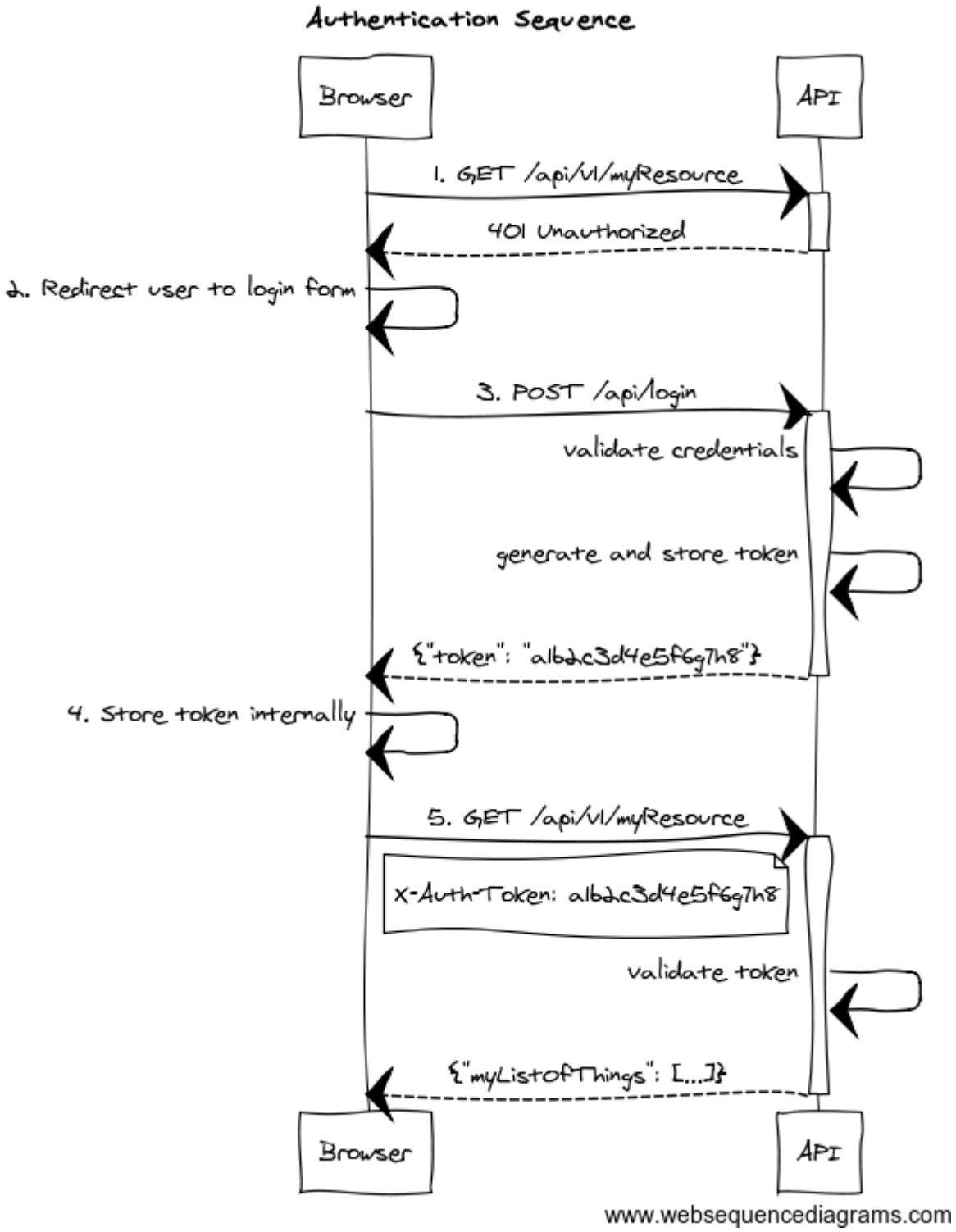

Figure 2. Sequence diagram overview of the plugin behavior

1. The client application requests and endpoint that requires authentication, so the server responds with a 401 response.
2. The client redirects the user to the login form.

3. The user enter credentials, and the client sends a request to the authentication endpoint. The server validates credentials, and if valid, generates, stores and sends back a token to the client.
4. The client then stores the token internally. It will be sent on every API method request.
5. The client sends again a request to the protected resource, passing the token as an HTTP header.
6. The server validates the token, and if valid, executes the actual operation requested.

As per the REST definition, the client is transferring its state on every request so the server is truly stateless.

This plugin helps you to wire your existing Spring Security authentication mechanism, provides you with ready-to-use token generation strategies and comes prepackaged with JWT, Memcached, GORM, Redis and Grails Cache support for token storage.

A.1.1. Release History

You can view all releases at https://github.com/alvarosanchez/grails-spring-security-rest/releases.

- 19 August 2015
    - 1.5.2
- 6 May 2015
    - 1.5.1
- 6 May 2015
    - 1.5.0
- 21 April 2015
    - 1.5.0.RC4
- 13 April 2015
    - 1.5.0.RC3
- 9 April 2015
    - 1.5.0.RC2
- 2 April 2015
    - 1.5.0.RC1
- 1 April 2015
    - 1.5.0.M3
- 24 February 2015
    - 1.5.0.M2
- 3 February 2015
    - 1.5.0.M1
- 28 January 2015
    - 1.4.1
- 12 November 2014
    - 1.4.1.RC2

- 20 October 2014
    - 1.4.1.RC1
- 12 August 2014
    - 1.4.0
- 14 July 2014
    - 1.4.0.RC5
- 4 July 2014
    - 1.4.0.RC4
- 24 June 2014
    - 1.4.0.RC3
- 24 June 2014
    - 1.4.0.RC2
- 20 June 2014
    - 1.4.0.RC1
- 11 June 2014
    - 1.4.0.M3
- 1 June 2014
    - 1.4.0.M2
- 29 May 2014
    - 1.4.0.M1
- 23 April 2014
    - 1.3.4
- 16 April 2014
    - 1.3.3
- 3 April 2014
    - 1.3.2
- 18 March 2014
    - 1.3.1

- 4 March 2014
    - 1.3.0
- 17 February 2014
    - 1.2.5
- 10 February 2014
    - 1.2.4
- 4 February 2014
    - 1.2.3
- 31 January 2014
    - 1.2.2
- 15 January 2014
    - 1.2.0
- 14 January 2014
    - 1.1.0
- 13 January 2014
    - 1.0.1
- 12 January 2014
    - 1.0.0
- 10 January 2014
    - 1.0.0.RC2
- 31 December 2013
    - Initial 1.0.0.RC1 release.

A.2. What's new in 1.5?

A.2.1. JWT support

JWT is fully supported and is now the default token "storage" mechanism. If you still want to use your previous storage (such as Memcached or GORM), make sure you explicitly set to `true` one of the following properties:

Table 1. New configuration values in 1.5

Config key	Default value
`grails.plugin.springsecurity.rest.token.storage.useMemcached`	`false`
`grails.plugin.springsecurity.rest.token.storage.useGorm`	`false`
`grails.plugin.springsecurity.rest.token.storage.useGrailsCache`	`false`

If switching over JWT, the logout behavior is not available anymore. Read the documentation on how to implement your own logout strategy if you want to.

A.2.2. Redis support

Redis can now be used as token storage service. Thanks to Philipp Eschenbach for his initial contribution.

A.2.3. New package base

Packages `com.odobo.grails.plugin.springsecurity.rest.*` have been refactored to simply `grails.plugin.springsecurity.rest.*`. Make sure to double-check your imports when upgrading to 1.5.

A.2.4. Other minor changes

- The plugin now uses Grails 2.4.4, and the build and tests are executed with Java 8.
- Documentation for older versions is now published at http://alvarosanchez.github.io/grails-spring-security-rest

A.3. What's new in 1.4?

A.3.1. Full compatibility with Spring Security core.

Up to previous releases, this plugin was overriding "stateful" Spring Security core beans, to ensure a stateless behaviour. After some users reported issues integrating this plugin with existing installations, version 1.4 now follows a more friendly approach.

A new chapter has been created explaining how to configure the filter chains appropriately.

A.3.2. RFC 6750 Bearer Token support by default

Now, the token validation and rendering aligns with the RFC 6750 Bearer Token spec. If you want to keep the old behaviour, simply disable it by setting `grails.plugin.springsecurity.rest.token.validation.useBearerToken = false`

A.3.3. Credentials are extracted from JSON by default

It makes more sense in a REST application. The old behaviour can still be used by using the corresponding configuration property.

A.3.4. Anonymous access is allowed

In case you want to enable anonymous access (read: not authenticated) to certain URL patterns, you can do so. Take a look at the [new chapter in the documentation|guide:tokenValidation].

A.3.5. Other minor changes

- Upgraded dependencies:
  - `spring-security-core:2.0-RC3`.
  - `cors:1.1.6`.

A.4. Configuration

Once the plugin is installed, you are ready to go. The default configuration is to use signed JWT's for tokens. However, you can select any other token storage strategy:

1. Grails Cache.
2. Memcached.
3. GORM.
4. Redis.
5. Provide your own: implement TokenStorageService and register it in `resources.groovy` as `tokenStorageService`.

A.4.1. Plugin configuration

This plugin depends on Spring Security Core 2.x. Make sure your application is compatible with that version first.

This plugin is compatible by default with Spring Security core traditional, form-based authentication. The important thing to remember is: you have to separate the filter chains, so different filters are applied on each case.

The stateless, token-based approach of this plugin is incompatible with the HTTP session-based approach of Spring Security, core, so the trick is to identify what URL patterns have to be stateless, and what others have to be stateful (if any).

To configure the chains properly, you can use the `grails.plugin.springsecurity.filterChain.chainMap` property:

Listing 31. `grails.plugin.springsecurity.filterChain.chainMap`

```
grails.plugin.springsecurity.filterChain.chainMap = [
    '/api/**': 'JOINED_FILTERS,-exceptionTranslationFilter,-
authenticationProcessingFilter,-securityContextPersistenceFilter,-
rememberMeAuthenticationFilter',  // Stateless chain
    '/**': 'JOINED_FILTERS,-restTokenValidationFilter,-
restExceptionTranslationFilter'
// Traditional chain
]
```

To understand this syntax, please read the Spring Security Core documentation. Long story short: `JOINED_FILTERS` refers to all the configured filters. The minus (-) notation means all the previous values but the neglected one.

So the first chain applies all the filters except the stateful ones. The second one applies all the filters but the stateless ones.

Make sure that the stateless chain applies not only to your REST controllers, but also to the URL's where this plugin filters are listening: by default, `/api/login` for authentication, `/api/logout` for logout and `/api/validate` for token validation.

The difference is that, in a traditional form-based authentication, Spring Security will respond with an HTTP 302 redirect to the login controller. That doesn't work for an API, so in the stateless approach, an HTTP 401 response will be sent back.

A.5. Events

The Spring Security Rest plugin fires events exactly like Spring Security Core does.

A.5.1. Event Notification

You can set up event notifications in two ways. The sections that follow describe each approach in more detail.

1. Register an event listener, ignoring events that do not interest you. Spring allows only partial event subscription; you use generics to register the class of events that interest you, and you are notified of that class and all subclasses.
2. Register one or more callback closures in `grails-app/conf/Config.groovy` that take advantage of the plugin's `grails.plugin.springsecurity.rest.RestSecurityEventListener`. The listener does the filtering for you.

AuthenticationEventPublisher

Spring Security REST publishes events using an AuthenticationEventPublisher which in turn fire events using the ApplicationEventPublisher. By default no events are fired since the `AuthenticationEventPublisher` instance registered is a `grails.plugin.springsecurity.rest.authentication.NullRestAuthenticationEventPublisher`. But you can enable event publishing by setting `grails.plugin.springsecurity.useSecurityEventListener = true` in `grails-app/conf/Config.groovy`.

You can use the `useSecurityEventListener` setting to temporarily disable and enable the callbacks, or enable them per-environment.

Token Creation

Currently the Spring Security REST plugin supports a single event in addition to the default spring security events. The event is fired whenever a new token is created. See `grails.plugin.springsecurity.rest.RestTokenCreationEvent`

Every time a token is successfully submitted, an `AuthenticationSuccessEvent` will be fired.

A.5.2. Registering an Event Listener

Enable events with `grails.plugin.springsecurity.useSecurityEventListener = true` and create one or more Groovy or Java classes, for example:

Listing 32. Custom event listener

```
package com.foo.bar

import org.springframework.context.ApplicationListener
import grails.plugin.springsecurity.rest.RestTokenCreationEvent

class MySecurityEventListener
      implements ApplicationListener<RestTokenCreationEvent> {

   void onApplicationEvent(RestTokenCreationEvent event) {
      // The access token is a delegate of the event, so you have access to an
instance of `grails.plugin.springsecurity.rest.token.AccessToken`
   }
}
```

Register the class in `grails-app/conf/spring/resources.groovy`:

Listing 33. Event listener registration

```
import com.foo.bar.MySecurityEventListener

beans = {
   mySecurityEventListener(MySecurityEventListener)
}
```

A.5.3. Registering Callback Closures

Alternatively, enable events with `grails.plugin.springsecurity.useSecurityEventListener = true` and register one or more callback closure(s) in `grails-app/conf/Config.groovy` and let `SecurityEventListener` do the filtering.

Implement the event handlers that you need, for example:

Listing 34. Callback closures

```
grails.plugin.springsecurity.useSecurityEventListener = true

grails.plugin.springsecurity.onRestTokenCreationEvent = { e, appCtx ->
   // handle RestTokenCreationEvent
}
```

None of these closures are required; if none are configured, nothing will be called. Just implement the event handlers that you need.

A.6. Authentication Endpoint

The authentication filter uses the default `authenticationManager` bean, which in turn uses all the registered authentication providers. See the Spring Security Core guide for more information about how to define your own providers. Note that you can easily plug any Spring Security sub-plugin (like the LDAP one) to use a different authentication strategy.

If the authentication is successful, a token generator is used to generate a token, and a token storage implementation is used to store the token. Finally, the JSON response sent back to the client is rendered by a `restAuthenticationTokenJsonRenderer` bean. See the token rendering documentation for more details.

This authentication filter will only be applied to the above configured URL and can also be disabled, in case a different approach for token creation is followed. In the rest of the cases, the request will continue through the filter chain, reaching Spring Security Core filters. Bear in mind that, by default, Spring Security Core 2.x locks down all URL's unless a explicit securiy rule has been specified for each of them.

See Spring Security Core documentation for more information.

The following are the `Config.groovy` properties available:

Table 2. Authentication configuration options

Config key	Default value
`grails.plugin.springsecurity.rest.login.active`	`true`
`grails.plugin.springsecurity.rest.login.endpointUrl`	`/api/login`
`grails.plugin.springsecurity.rest.login.failureStatusCode`	`401`

A.6.1. Extracting credentials from the request

The plugin supports 2 ways of extracting the username and password: using request parameters, and using a JSON payload. To align with the RESTful principles, JSON payload is the default behaviour.

From a JSON request

Table 3. JSON credentials extraction configuration properties

Config key	Default value
`grails.plugin.springsecurity.rest.login.useJsonCredentials`	`true`
`grails.plugin.springsecurity.rest.login.usernamePropertyName`	`username`
`grails.plugin.springsecurity.rest.login.passwordPropertyName`	`password`

The default implementation expects a request like this:

Listing 35. Example JSON authentication request

```
{
    "username": "john.doe",
    "password": "dontTellAnybody"
}
```

If you use `usernamePropertyName` and `passwordPropertyName` properties mentioned above, your JSON request can look like:

Listing 36. Custom JSON authentication request

```
{
    "login": "john.doe",
    "pwd": "dontTellAnybody"
}
```

With the following config:

Listing 37. Custom JSON authentication configuration properties

```
grails.plugin.springsecurity.rest.login.usernamePropertyName = 'login'
grails.plugin.springsecurity.rest.login.passwordPropertyName = 'pwd'
```

If your JSON request format is different, you can plug your own implementation by defining a class which extends `http://alvarosanchez.github.io/grails-spring-security-rest/latest/docs/gapi/grails/plugin/springsecurity/rest/credentials/AbstractJsonPayloadCredentialsExtractor.html[AbstractJsonPayloadCredentialsExtractor]`. The default implementation looks like this:

Listing 38. `DefaultJsonPayloadCredentialsExtractor`

```
@Slf4j
class DefaultJsonPayloadCredentialsExtractor extends
AbstractJsonPayloadCredentialsExtractor {

    String usernamePropertyName
    String passwordPropertyName

    UsernamePasswordAuthenticationToken extractCredentials(HttpServletRequest
httpServletRequest) {
        def jsonBody = getJsonBody(httpServletRequest)

        if (jsonBody) {
            String username = jsonBody."${usernamePropertyName}"
            String password = jsonBody."${passwordPropertyName}"

            log.debug "Extracted credentials from JSON payload. Username:
${username}, password: ${password?.size()?'[PROTECTED]':'[MISSING]'}"

            new UsernamePasswordAuthenticationToken(username, password)
        } else {
            log.debug "No JSON body sent in the request"
            return null
        }
    }

}
```

Once you are done, register it in `resources.groovy` with the name `credentialsExtractor`.

From request parameters

Note that the name of the parameters can also be customised:

Table 4. Parameter extraction configuration options

Config key	Default value
`grails.plugin.springsecurity.rest.login.useRequestParamsCredentials`	`false`
`grails.plugin.springsecurity.rest.login.usernamePropertyName`	`username`
`grails.plugin.springsecurity.rest.login.passwordPropertyName`	`password`

A.6.2. Logout

Logout is not possible when using JWT tokens (the default strategy), as no state is kept in the server. If you still want to have logout, you can provide your own implementation by creating a subclass of JwtTokenStorageService and overriding the methods `storeToken` and `removeToken`.

Then, register your implementation in `resources.groovy` as `tokenStorageService`.

However, a more rational approach would be just to remove the token from the client (eg, browser's local storage) and let the tokens expire (they will expire anyway, unlike with other storages like Memcached or Redis where they get refreshed on every access).

The logout filter exposes an endpoint for deleting tokens. It will read the token from an HTTP header. If found, will delete it from the storage, sending a 200 response. Otherwise, it will send a 404 response.

You can configure it in `Config.groovy` using this properties:

Table 5. Logout configuration options

Config key	Default value
`grails.plugin.springsecurity.rest.logout.endpointUrl`	`/api/logout`
`grails.plugin.springsecurity.rest.token.validation.headerName`	`X-Auth-Token`

A.7. Token Generation

By default, the plugin generates JWT tokens. Note that when using JWT, you can't plug any other token generator.

If you are not using JWT, but any stateful strategy like Memcached or GORM, the following strategies are available:

- Using java.security.SecureRandom.
- Using java.util.UUID.

The strategy used is configurable in `Config.groovy`:

Table 6. Token generation configuration options

Config key	Default value
`grails.plugin.springsecurity.rest.token.generation.useSecureRandom`	`true`
`grails.plugin.springsecurity.rest.token.generation.useUUID`	`false`

Both of them generate tokens of 32 alphanumeric characters.

That should be enough for most of the human beings. But if you still want to provide your own implementation, simply write a class implementing TokenGenerator and wire it up in `resources.groovy` as `tokenGenerator`.

A.8. Token Storage

The tokens are stored on the server using a `tokenStorageService` bean. The plugin comes with out-of-the-box support for JWT, Memcached, GORM and Grails Cache, but you can use your own strategy implementing the TokenStorageService interface.

The default implementation, JWT, is stateless. Nothing is really stored. However, the plugin still gives a chance to the other implementations to store the principal if they need to.

A.8.1. JSON Web Token

JSON Web Token (JWT) is an IETF standard (in progress) which defines a secure way to encapsulate arbitrary data that can be sent over unsecure URL's.

Generally speaking, JWT's can be useful in the following use cases:

- When generating "one click" action emails, like "delete this comment" or "add this to favorites". Instead of giving the users URL's like `/comment/delete/123`, you can give them something like `/comment/delete/<JWT_TOKEN>`, where the `JWT_TOKEN` contains encapsulated information about the user and the comment, in a safe way, so authentication is not required.
- To achieve single sign-on, by sharing a JWT across applications.

In the context of authentication and authorization, JWT will help you implement a stateless implementation, as the principal information is stored directly in the JWT.

How does a JWT looks like?

Figure 3. JWT example

Header

A base64-encoded JSON like:

Listing 39. JWT header

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

Claims

A base64-encoded JSON like:

Listing 40. JWT claims

```
{
    "exp": 1422990129,
    "sub": "jimi",
    "roles": [
        "ROLE_ADMIN",
        "ROLE_USER"
    ],
    "iat": 1422986529
}
```

Signature

Depends on the algorithm specified on the header, it can be a digital signature of the base64-encoded header and claims, or an encryption of them using RSA.

Signed JWT's

By default, this plugin uses signed JWT's as specified by the JSON Web Signature specification. More specifically, the algorithm used is HMAC SHA-256 with a specified shared secret. The relevant configuration properties are:

Table 7. JWT configuration options for signing

Config key	Default value
`grails.plugin.springsecurity.rest.token.storage.useJwt`	`true`
`grails.plugin.springsecurity.rest.token.storage.jwt.useSignedJwt`	`true`
`grails.plugin.springsecurity.rest.token.storage.jwt.secret`	`'qrD6h8K6S9503Q06Y6Rfk21TErImPYqa'`
`grails.plugin.springsecurity.rest.token.storage.jwt.expiration`	`3600`

Encrypted JWT's

Grails's `grails-docs` artifact includes a version of `com.lowagie:itext` which in turns bring old BouncyCastle's libraries into the classpath. To avoid the problem, you have to override explicitly that dependency:

Listing 41. Excluding BouncyCastle's libraries

```
build("com.lowagie:itext:2.0.8") { excludes "bouncycastle:bcprov-
jdk14:138", "org.bouncycastle:bcprov-jdk14:1.38" }
{code}
```

In the previous strategy, the claims are just signed, so it prevents an attacker to tamper its contents to introduce malicious data or try a privilege escalation by adding more roles. However, the claims can be decoded just by using Base 64.

If the claims contains sensitive information, you can use a JSON Web Encryption algorithm to prevent them to be decoded. Particularly, this plugin uses RSAES OAEP for key encryption and AES GCM (Galois/Counter Mode) algorithm with a 256 bit key for content encryption.

By default, RSA public/private keys ar generated every time the application runs. This means that generated tokens won't be decrypted across executions of the application. So better create your own key pair using OpenSSL:

Listing 42. Certificate generation for JWT

```
openssl genrsa -out private_key.pem 2048
openssl pkcs8 -topk8 -inform PEM -outform DER -in private_key.pem -out
private_key.der -nocrypt
openssl rsa -in private_key.pem -pubout -outform DER -out public_key.der
```

Then, configure the keys properly, along with the rest of the configuration:

Table 8. JWT configuration options for signing

Config key	Default value
grails.plugin.springsecurity.rest.token.storage.useJwt	true
grails.plugin.springsecurity.rest.token.storage.jwt.useEncryptedJwt	false
grails.plugin.springsecurity.rest.token.storage.jwt.privateKeyPath	null
grails.plugin.springsecurity.rest.token.storage.jwt.publicKeyPath	null

Example configuration:

Listing 43. JWT encryption example configuration

```
grails.plugin.springsecurity.rest.token.storage.jwt.useEncryptedJwt = true
grails.plugin.springsecurity.rest.token.storage.jwt.privateKeyPath =
'/path/to/private_key.der'
grails.plugin.springsecurity.rest.token.storage.jwt.publicKeyPath =
'/path/to/public_key.der'
```

The performance of encryption algorithms is much slower compared with signing ones. If you are considering encrypting you JWT's, think if you really need it.

Token expiration and refresh tokens

When using JWT, issued access tokens expire after a period of time, and they are paired with refresh tokens, eg:

Listing 44. Sample access token response, including refresh token


```
{
   "username": "jimi",
   "roles": [
      "ROLE_ADMIN",
      "ROLE_USER"
   ],
   "expires_in": 3600,
   "token_type": "Bearer",
   "refresh_token":
"eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.fUaSWIdZakFX7CyimRIPhuw0sfevgmwL2x
zm5H0TuaqwKx24EafCO0TruGKG-lN-
wGCITssnF2LQTqRzQGp0PoLXHfUJ0kkz5rBl6LtnRu7cdD1ZUNYXLJtFjQ3IATzoo15tPafRPyStG1Qm7-
1L0VxquhrLxkkpti0F1_VTytZAq8ltFrnxM4ahJUwS7eriivvdLqmHtnwuXw0kBXEseIyCkiyKklWDJAcD
_P_gHoQJvSCoXedlr7Pp0n6LEUrRWJ2Hb-
Zyt9dWqWDxm9nyDeEVtEZGcQtpgCGgbXxaUpULIy5nvrbRzXSNyT6iXhK1CLqiFVkfh-Y-
DHXdB6Q4sg.uYdpxl835KnlkqC5.gBgSnPWZOo6FINovJNG7Xx2RuS09QJbU4-_J4EgZQkygt8xE-
HfdYaOmtmJLjGJR1XKoaRsuX1gNjFoCZgqWAon6.Zsrk52dkjskSVQLXZBQooQ",
   "access_token": "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.n-
gGe65x0SlSXS3fTG8ZLdXvv6b5_1pDvkcGyCjFy-
vm1VhaBEQL5p3hc6iUcCAcuyrqzGk95lV9dHCv46cNfCiUFHWfbEcd4nqScIxBbc28xO9L1mNLnZ0G1rx1
Mx1L0Y_ZPoSxDXpJaHCT28cdZffHLxx2B9ioIClgdlYBAJ5Oz8VT39-
D0QSomS6QhFqmcpbDsXrsKxs545Pn-TIlu-fSQ4wpIvAxusOKB6CV2EYKqBplMBrh-
3btE8WksVcX2N3LsrcMhrKxSKi93c06MZh6JzSLWe5bl9hvUvBdEuwDrk-
fQgD3ZlmjjoevRWYhv_kslW1PlqUHYmKOQ7csUw.3mvvsFWikEjZzExA.YixjnnzzcPRy_uUpgPv5zqOfs
hv3pUwfrME0AijpsB7u9CmJe94g6f2y_3vqUps-5weKKGZyk3ZtnwEbPVAk9-HZt-
Y27SbZl4JNCFEOLVsMsK8.h4j9BdFXuWKKez6xxRAwJA"
}
```

Refresh tokens never expire, and can be used to obtain a new access token by sending a POST request to the `/oauth/access_token` endpoint:

Listing 45. Sample HTTP request to obtain an access token

```
POST /myApp/oauth/access_token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00
ifQ....
```

As you can see, is a form request with 2 parameters:

- `grant_type`: must be `refres_token` always.
- `refresh_token`: the refresh token provided earlier.

As refresh tokens never expire, they must be securely stored in your client application. See section 10.4 of the OAuth 2.0 spec for more information.

Memcached

To use Memcached, simply define the following configuration properties to match your environments accordingly:

Table 9. Memcached configuration options

Config key	Default value
`grails.plugin.springsecurity.rest.token.storage.useMemcached`	`false`
`grails.plugin.springsecurity.rest.token.storage.memcached.hosts`	`localhost:11211`
`grails.plugin.springsecurity.rest.token.storage.memcached.username`	`''`
`grails.plugin.springsecurity.rest.token.storage.memcached.password`	`''`
`grails.plugin.springsecurity.rest.token.storage.memcached.expiration`	`3600`

For development, if you have Memcached installed locally with the default settings, just define `grails.plugin.springsecurity.rest.token.storage.useMemcached = true`. It should work.

In Memcached tokens will expire automatically after the configured timeout (1h by default). They get refreshed on every access

GORM

To use GORM, these are the relevant configuration properties:

Table 10. GORM configuration options

Config key	Default value
grails.plugin.springsecurity.rest.token.storage.useGorm	false
grails.plugin.springsecurity.rest.token.storage.gorm.tokenDomainClassName	null
grails.plugin.springsecurity.rest.token.storage.gorm.tokenValuePropertyName	tokenValue
grails.plugin.springsecurity.rest.token.storage.gorm.usernamePropertyName	username

The relevant domain class should look something like this:

Listing 46. Authentication token domain class example

```
package org.example.product

class AuthenticationToken {

    String tokenValue
    String username

    static mapping = {
        version false
    }
}
```

For the `tokenDomainClassName` configuration you must enter a fully qualified class name. In the case of the example above: `grails.plugin.springsecurity.rest.token.storage.gorm.tokenDomainClassName = 'org.example.product.AuthenticationToken'`

A few things to take into consideration when using GORM for token storage:

- Instead of storing the whole `UserDetails` object, probably only the username is needed. This is because applications using this strategy will probably have the standard User and Role domain classes. When the token is verified the username is passed to the default `userDetailsService` bean, which in the case of the default Spring Security Core GORM implementation will fetch the information from the mentioned domain classes.
- GORM's optimistic locking feature is likely unnecessary and may cause performance issues.
- You'll have to handle token expiration by yourself via Quartz jobs or a similar mechanism. There are various ways you might go about this.

GORM Token Expiration Examples

Adding a GORM autoTimestamp property like `lastUpdated` or `dateCreated` and sorting out stale or old tokens with Quartz jobs are the most obvious routes. Each has its drawbacks though.

`dateCreated` is useful if you want tokens to expire a set time after they are issued. However, API users who didn't pay attention to when their token was issued may find themselves needing a new token unexpectedly.

```
Date dateCreated
```

`lastUpdated` requires a change to the token domain instance in order to be triggered. Something as simple as an access counter may work as a strategy to keepTokens fresh, but doing a write to a disk based database on each token access may be something you would prefer to avoid for the sake of performance.

```
Date lastUpdated
Integer accessCount = 0

def afterLoad() {
  accessCount++
}
```

Simply using your own date or timestamp is also a valid option.

```
Date refreshed = new Date()

def afterLoad() {
    // if being accessed and it is more than a day since last marked as refreshed
    // and it hasn't been wiped out by Quartz job (it exists, duh)
    // then refresh it
    if (refreshed < new Date() -1) {
        refreshed = new Date()
        it.save()
    }
}
```

Here is an example quartz job to go with the custom refresh timestamp above:

```
class RemoveStaleTokensJob {
    static triggers = {
        cron name: 'every4hours', cronExpression: '0 0 */4 * * *'
    }

    void execute() {
        AuthenticationToken.executeUpdate('delete AuthenticationToken a where
a.refreshed < ?' [new Date()-1])
    }
}
```

Redis

To use Redis as a token store simply you just have to enable it in you configuration by setting `useRedis` to `true` (see table below).

You have to have the `redis` plugin installed in order to be able to use Redis as your token store. Refer to the Redis plugin documentation for more details about how to configure it.

Configuration options for Redis:

Table 11. Redis configuration options

Config key	Default value
`grails.plugin.springsecurity.rest.token.storage.useRedis`	`false`
`grails.plugin.springsecurity.rest.token.storage.redis.expiration`	`3600`

A.8.2. Grails Cache

To use Grails Cache, simply define a cache name:

Table 12. Redis configuration options

Config key	Default value
`grails.plugin.springsecurity.rest.token.storage.useGrailsCache`	`false`
`grails.plugin.springsecurity.rest.token.storage.grailsCacheName`	`null`

The cache name should correspond to a name specified in the [cache DSL|http://grails-plugins.github.io/grails-cache/docs/manual/guide/usage.html#dsl].

Token expiration / eviction / TTL

By default, Spring Cache abstraction does not support expiration. It depends on the specific support of the actual providers. Grails has several plugins for this:

- Core: unsupported.
- Ehcache: supported.
- Redis: unsupported.
- Gemfire: unsupported.

There is a bug in `:cache-ehcache:1.0.0` plugin that will cause issues. It's recommended that you use the latest version. See #89 for more information.

A.9. Token Rendering

By default, this plugin renders the token in RFC 6750 Bearer Token format:

Sample Access token response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
    "access_token"
:"eyJhbGciOiJIUzI1NiJ9.eyJleHAiOjE0MjI5OTU5MjIsInN1YiI6ImppbWkiLCJyb2xlcyI6WyJST0x
FX0FETUlOIiwiUk9MRV9VU0VSIl0sImlhdCI6MTQyMjk5MjMyMn0.rA7A2Gwt14LaYMpxNRtrCdO24RGrf
HtZXY9fIjV8x8o",
    "token_type":"Bearer",
    "username": "john.doe",
    "roles": [
        "ROLE_ADMIN",
        "ROLE_USER"
    ]
}
```

As per the RFC, `access_token` and `token_type` property names cannot be customised.

The JSON structure can be customised with the following configuration keys:

Table 13. Token rendering configuration options

Config key	Default value
grails.plugin.springsecurity.rest.token.rendering.usernamePropertyName	username
grails.plugin.springsecurity.rest.token.rendering.authoritiesPropertyName	roles

E.g., with the following configuration:

```
grails.plugin.springsecurity.rest.token.rendering.usernamePropertyName = 'login'
grails.plugin.springsecurity.rest.token.rendering.authoritiesPropertyName =
'permissions'
```

The output will look like:

```
{
    "access_token"
:"eyJhbGciOiJIUzI1NiJ9.eyJleHAiOjE0MjI5OTU5MjIsInN1YiI6ImppbWkiLCJyb2xlcyI6WyJST0x
FX0FETUlOIiwiUk9MRV9VU0VSIl0sImlhdCI6MTQyMjk5MjMyMn0.rA7A2Gwt14LaYMpxNRtrCdO24RGrf
HtZXY9fIjV8x8o",
    "token_type":"Bearer",
    "login": "john.doe",
    "permissions": [
        "ROLE_ADMIN",
        "ROLE_USER"
    ]
}
```

A.9.1. Disabling bearer tokens support for full response customisation

In order to fully customise the response, you need first to disable bearer tokens support by setting `grails.plugin.springsecurity.rest.token.validation.useBearerToken = false`. That will enable you to use this additional property:

Table 14. Token rendering configuration options

Config key	Default value
`grails.plugin.springsecurity.rest.token.rendering.tokenPropertyName`	`access_token`

Disabling bearer token support impacts the way tokens are extracted from the HTTP request. Please, read carefully the chapter about token validation first.

If you want your own implementation, simply create a class implementing AccessTokenJsonRenderer and wire it up in `resources.groovy` with name `accessTokenJsonRenderer`.

The principal object stored in the security context, and passed to the JSON renderer, is coming from the configured authentication providers. In most cases, this will be a `UserDetails` object retrieved using the `userDetailsService` bean.

If you want to render additional information in your JSON response, you have to:

1. Configure an alternative `userDetailsService` bean that retrieves the additional information you want, and put it in a principal object.
2. Configure an alternative `accessTokenJsonRenderer` that reads that information from the `restAuthenticationToken.principal` object.

A.10. Token Validation Filter

The token validation filter looks for the token in the request and then tries to validate it using the configured token storage implementation.

If the validation is successful, the principal object is stored in the security context. This allows you to use in your application `@Secured`, `springSecurityService.principal` and so on.

`springSecurityService.currentUser` expects a `grails.plugin.springsecurity.userdetails.GrailsUser` to perform a DB query. However, this plugins stores in the security context just a principal `Object`, because it does not assume you are using domain classes to store the users. Use `springSecurityService.principal` instead.

This plugin supports RFC 6750 Bearer Token specification out-of-the-box.

A.10.1. Sending tokens in the request

The token can be sent in the `Authorization` request header:

Listing 47. Accessing a protected resource using `Authorization` *request header*

```
GET /protectedResource HTTP/1.1
Host: server.example.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiJ9.eyJleHAiOjE0MjI5OTU5MjIsInN1YiI6ImppbWkiLCJyb2xlcyI6WyJST0xFX
0FETUlOIiwiUk9MRV9VU0VSIl0sImlhdCI6MTQyMjk5MjMyMn0.rA7A2Gwt14LaYMpxNRtrCdO24RGrfHt
ZXY9fIjV8x8o
```

Or using form-encoded body parameters:

Listing 48. Accessing a protected resource using body parameters

```
POST /protectedResource HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

access_token=eyJhbGciOiJIUzI1NiJ9.eyJleHAiOjE0MjI5OTU5MjIsInN1YiI6ImppbWkiLCJyb2xl
cyI6WyJST0xFX0FETUlOIiwiUk9MRV9VU0VSIl0sImlhdCI6MTQyMjk5MjMyMn0.rA7A2Gwt14LaYMpxNR
trCdO24RGrfHtZXY9fIjV8x8o
```

If you need to use the `GET` HTTP method (to render images in an `img` tag, for example), you can also send the access token in a query string parameter named `access_token`:

If you disable the bearer token support, you can customise it further:

```
grails.plugin.springsecurity.rest.token.validation.useBearerToken = false
grails.plugin.springsecurity.rest.token.validation.headerName = 'X-Auth-Token'
```

If you still want to have full access and read the token from a different part of the request, you can implement a TokenReader and register it in your `resources.groovy` as `tokenReader`.

You must disable bearer token support to register your own `tokenReader` implementation.

A.10.2. Anonymous access

If you want to enable anonymous access to URL's where this plugin's filters are applied, you need to:

1. Configure `enableAnonymousAccess = true` (see table below).
2. Make sure that the `anonymousAuthenticationFilter` is applied before `restTokenValidationFilter`. See how to configure filters for more details.

For example, with this configuration:

Sample configuration to allow anonymous access

```
grails {
    plugin {
        springsecurity {
            filterChain {
                chainMap = [
                    '/api/guest/**':
'anonymousAuthenticationFilter,restTokenValidationFilter,restExceptionTranslationF
ilter,filterInvocationInterceptor',
                    '/api/**': 'JOINED_FILTERS,-anonymousAuthenticationFilter,-
exceptionTranslationFilter,-authenticationProcessingFilter,-
securityContextPersistenceFilter',
                    '/**': 'JOINED_FILTERS,-restTokenValidationFilter,-
restExceptionTranslationFilter'
                ]
            }

            //Other Spring Security settings
            //...

            rest {
                token {
                    validation {
                        enableAnonymousAccess = true
                    }
                }
            }
        }
    }
}
```

The following chains are configured:

1. `/api/guest/**` is a stateless chain that allows anonymous access when no token is sent. If however a token is on the request, it will be validated.
2. `/api/**` is a stateless chain that doesn't allow anonymous access. Thus, the token will always be required, and if missing, a Bad Request reponse will be sent back to the client.
3. `/**` (read: everything else) is a traditional stateful chain.

A.10.3. Validation Endpoint

There is also an endpoint available that you can call in case you want to know if a given token is valid. It looks for the token in a HTTP header as well, and if the token is still valid, it renders guide:authentication[its JSON representation]. If the token does not exist, it will render a `grails.plugin.springsecurity.rest.login.failureStatusCode` response (`401` by default).

The relevant configuration properties for the validation endpoint are:

Table 15. Validation endpoint configuration options

Config key	Default value
`grails.plugin.springsecurity.rest.token.validation.active`	`true`
`grails.plugin.springsecurity.rest.token.validation.headerName`	`X-Auth-Token`
`grails.plugin.springsecurity.rest.token.validation.endpointUrl`	`/api/validate`

Note that `headerName` is only considered if `grails.plugin.springsecurity.rest.token.validation.useBearerToken` is set to `false`. Otherwise (the default approach), as per RFC 6750, the header name will be `Authorization` and the value will be `Bearer TOKEN_VALUE`.

A.11. CORS support

This plugin comes pre-installed with the CORS plugin, which enables Cross-Origin Resource Sharing. Refer to the plugin documentation to learn how to configure it.

The CORS plugin activates itself by default. If you don't want it for some environments, you can use `cors.enabled = false` within the appropriate environment block in your `Config.groovy`.

If you don't want CORS support at all, you can skip the plugin by excluding it when defining this plugin in your `BuildConfig.groovy`:

Listing 49. Excluding the CORS plugin

```
compile ':spring-security-rest:{{VERSION}}', {
    exclude 'cors
}
```

Note that you have to explicitly expose the headers you are using for token validation. Otherwise, the frontend application won't be allowed to read them:

```
cors.headers = ['Access-Control-Allow-Headers': 'Content-Type, Authorization']
```

Be also aware of the default behaviour of the CORS plugin for `Access-Control-Allow-Origin` is to just echo the `Origin` header sent by the browser. Make sure that in your production environment you properly configure the header.

A.12. Delegating authentication to OAuth providers

This plugin is meant to be used in applications serving a REST API's to pure Javascript clients. The main authentication flow of this plugin is to allow you to authenticate your users against any Spring Security-compatible user directory (like a DB or an LDAP server).

However, there might be situations where you want to delegate the authentication against a third-party provider, like Google or Facebook. Unfortunately, your pure Javascript front-end application cannot request the providers directly using OAuth, because then the access keys will be made public.

So is this plugin's responsibility to provide endpoints so your Grails backend acts as a proxy for your front-end client.

The flow is something like the following:

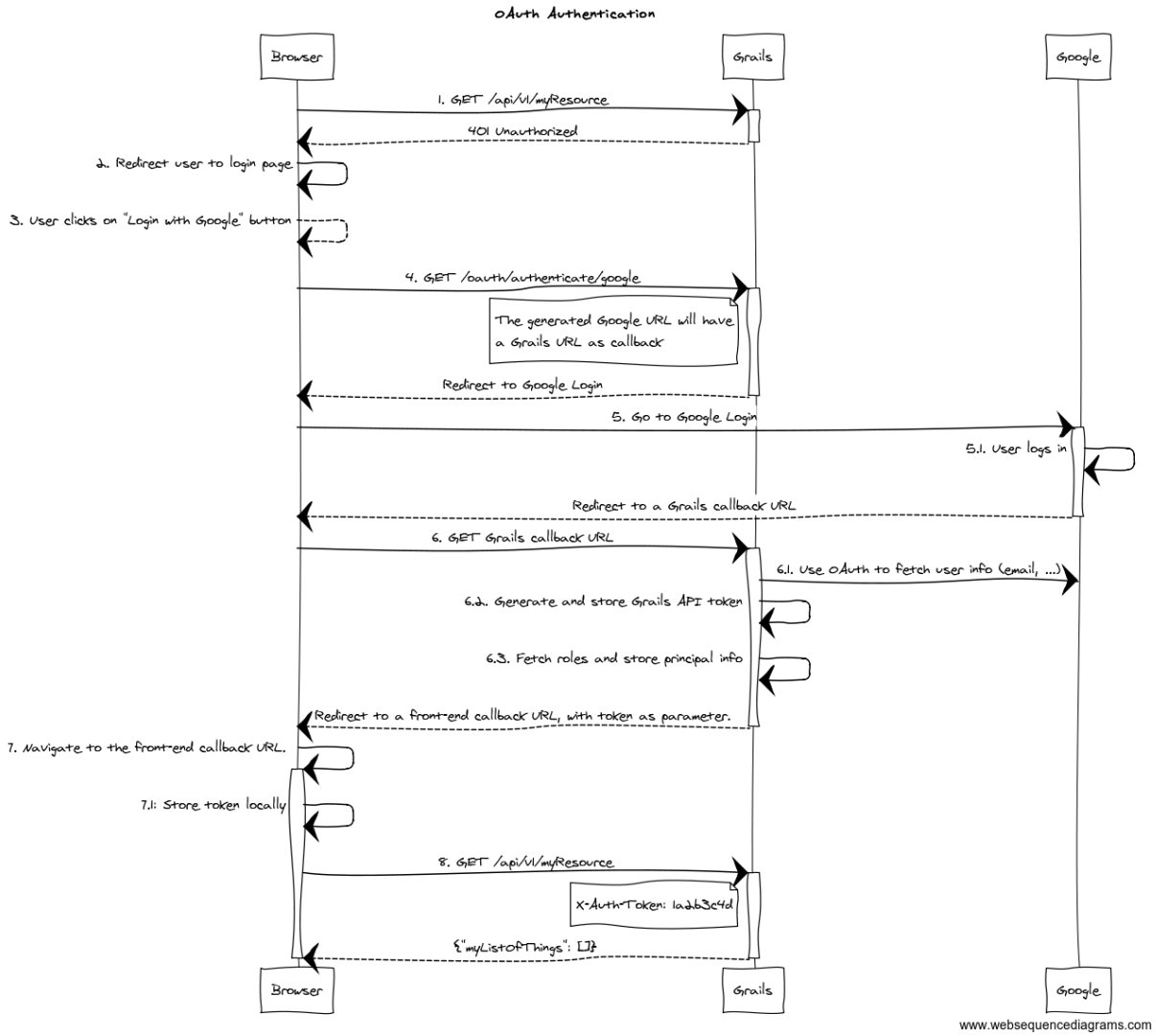

Figure 4. OAuth delegation protocol

1. The client application requests and endpoint that requires authentication, so the server responds with a 401 response (*).
2. The client redirects the user to the login form (*).
3. This time, instead of using username and password, the user clicks on "Login with Google" button.
4. Browser navigates to a Grails URL. Grails will generate a Google Login URL, giving Google a Grails callback URL.
5. Browser navigates to Google Login. User logs in, and Google redirects the browser to the Grails callback URL.

6. Browser navigates to that Grails callback URL. Then, Grails will use OAuth to fetch user information (like email) from Google. Based on that, will generate a REST API token and fetch and store principal information. The response from Grails will be a front-end URL where the token is a parameter.
7. The browser will navigate to that URL, and the Javascript logic will read the token from the URL and store it locally.
8. The client sends again a request to the protected resource, passing the token as an HTTP header (*).

The steps flagged with (*) remain unchanged from the normal flow.

The Grails callback URL mentioned above has this general format: `${grails.serverURL}/oauth/callback/${providerName}`. You will need to configure such URL in your OAuth 2.0 provider.

To support OAuth, this plugin uses Profile & Authentication Client for Java. So you can use any OAuth 2.0 provider they support. This includes at the time of writing:

- Dropbox.
- Facebook.
- GitHub.
- Google.
- LinkedIn.
- Windows Live.
- Wordpress.
- Yahoo.
- Paypal.

Note that OAuth 1.0a providers also work, like Twitter.

The plugin also supports CAS (Central Authentication Service) using the OAuth authentication flow. See CAS Authentication for details.

To start the OAuth authentication flow, from your frontend application, generate a link to `<YOUR_GRAILS_APP>/oauth/authenticate/<provider>`. The user clicking on that link represents step 4 in the previous diagram.

Note that you can define the frontend callback URL in `Config.groovy` under `grails.plugin.springsecurity.rest.oauth.frontendCallbackUrl`. You need to define a closure that will be called with the token value as parameter:

```
grails.plugin.springsecurity.rest.oauth.frontendCallbackUrl = { String tokenValue
-> "http://my.frontend-app.com/welcome.token=${tokenValue}" }
```

You can also define the URL as a `callback` parameter in the original link, eg:

```
http://your-grails-api.com/oauth/authenticate/google?callback=http://your-
frontend-app.com/auth-success.html?token=
```

In this case, the token will be **concatenated** to the end of the URL.

Upon successful OAuth authorisation (after step 6.1 in the above diagram), an OauthUser will be stored in the security context. This is done by a bean named `oauthUserDetailsService`. The default implementation delegates to the configured `userDetailsService` bean, passing the profile ID as the username:

Listing 50. DefaultOauthUserDetailsService

```
/**
 * Builds an {`link OauthUser}. Delegates to the default {`link
UserDetailsService.loadUserByUsername(java.lang.String)}
 * where the username passed is {`link UserProfile.getId()}. If the user is not
found, it will create a new one with
 * the the default roles.
 */
@Slf4j
class DefaultOauthUserDetailsService implements OauthUserDetailsService {

    @Delegate
    UserDetailsService userDetailsService

    UserDetailsChecker preAuthenticationChecks

    OauthUser loadUserByUserProfile(CommonProfile userProfile, Collection
<GrantedAuthority> defaultRoles)
            throws UsernameNotFoundException {
        UserDetails userDetails
        OauthUser oauthUser

        try {
            log.debug "Trying to fetch user details for user profile:
${userProfile}"
            userDetails = userDetailsService.loadUserByUsername userProfile.id

            log.debug "Checking user details with
${preAuthenticationChecks.class.name}"
            preAuthenticationChecks?.check(userDetails)

            Collection<GrantedAuthority> allRoles = userDetails.authorities +
defaultRoles
            oauthUser = new OauthUser(userDetails.username, userDetails.password,
allRoles, userProfile)
        } catch (UsernameNotFoundException unfe) {
            log.debug "User not found. Creating a new one with default roles:
${defaultRoles}"
            oauthUser = new OauthUser(userProfile.id, 'N/A', defaultRoles,
userProfile)
        }

        return oauthUser
    }

}
```

If you want to provide your own implementation, define it in `resources.groovy` with bean name `oauthUserDetailsService`. Make sure you implements the interface `OauthUserDetailsService`

If you want to do any additional post-OAuth authorisation check, you should do it on your `loadUserByUserProfile` implementation. This is useful if you want to allow your corporate users to log into your application using their Gmail account. In this case, you should decide based on OAuth20Profile.getEmail(), for instance:

Listing 51. Custom loadUserByUserProfile implementation

```
OauthUser loadUserByUserProfile(OAuth20Profile userProfile, Collection
<GrantedAuthority> defaultRoles) throws UsernameNotFoundException {
    if (userProfile.email.endsWith('example.org')) {
        return new OauthUser(userProfile.id, 'N/A', defaultRoles, userProfile)
    } else {
        throw new UsernameNotFoundException("User with email ${userProfile.email}
now allowed. Only `example.org accounts are allowed.")
    }
}
```

In case of any OAuth authentication failure, the plugin will redirect back to the frontend application anyway, so it has a chance to render a proper error message and/or offer the user the option to try again. In that case, the token parameter will be empty, and both `error` and `message` params will be appended:

```
http://your-frontend-app.com/auth-
success.html?token=&error=403&message=User+with+email+jimmy%40gmail.com+now
+allowed.+Only+%40example.com+accounts+are+allowed
```

Below are some examples on how to configure it for Google, Facebook and Twitter.

Google

Define the following block in your `Config.groovy`:

Listing 52. Google OAuth sample configuration

```
grails {
    plugin {
        springsecurity {

            rest {

                oauth {

                    frontendCallbackUrl = { String tokenValue ->
"http://my.frontend-app.com/welcome#token=${tokenValue}" }

                    google {

                        client = org.pac4j.oauth.client.Google2Client
                        key = 'xxxx.apps.googleusercontent.com'
                        secret = 'xxx'
                        scope = org.pac4j.oauth.client.Google2Client.Google2Scope
.EMAIL_AND_PROFILE
                        defaultRoles = ['ROLE_USER', 'ROLE_GOOGLE']

                    }
                }
            }
        }
    }
}
```

The `scope` can be from any value of the enum `org.pac4j.oauth.client.Google2Client.Google2Scope`. But if you use the default `OauthUserDetailsService`, you need to use `EMAIL_AND_PROFILE`. That is because the default implementation uses the profile ID as the username, and that is only returned by Google if `EMAIL_AND_PROFILE` scope is used.

Facebook

Define the following block in your `Config.groovy`:

Listing 53. Facebook OAuth sample configuration

```
grails {
    plugin {
        springsecurity {

            rest {

                oauth {

                    frontendCallbackUrl = { String tokenValue ->
"http://my.frontend-app.com/welcome#token=${tokenValue}" }

                    facebook {

                        client = org.pac4j.oauth.client.FacebookClient
                        key = 'xxx'
                        secret = 'yyy'
                        scope = 'email,user_location'
                        fields =
'id,name,first_name,middle_name,last_name,username'
                        defaultRoles = ['ROLE_USER', 'ROLE_FACEBOOK']
                    }
                }
            }
        }
    }
}
```

The `scope` is a comma-separated list, **without blanks**, of Facebook permissions. See the Facebook documentation for more details.

`fields` may contain a comma-separated list, **without blanks**, of user fields.

Both `scope` and `fields` are optional, but it's highly recommendable to fine tune those lists so you don't ask for information you don't need.

Twitter

Define the following block in your `Config.groovy`:

Listing 54. Twitter OAuth sample configuration

```
grails {
    plugin {
        springsecurity {

            rest {

                oauth {

                    frontendCallbackUrl = { String tokenValue ->
"http://my.frontend-app.com/welcome#token=${tokenValue}" }

                    twitter {

                        client = org.pac4j.oauth.client.TwitterClient
                        key = 'xxx'
                        secret = 'yyy'
                        defaultRoles = ['ROLE_USER', 'ROLE_TWITTER']
                    }
                }
            }
        }
    }
}
```

There is no additional configuration for Twitter.

CAS (Central Authentication Service)

Define the following block in your `Config.groovy`:

Listing 55. CAS sample configuration

```
grails {
    plugin {
        springsecurity {

            rest {

                oauth {

                    frontendCallbackUrl = { String tokenValue ->
"http://my.frontend-app.com/welcome#token=${tokenValue}" }

                    cas {

                        client = org.pac4j.cas.client.CasClient
                        casLoginUrl = "https://my.cas-server.com/cas/login"
                    }
                }
            }
        }
    }
}
```

Set `casLoginUrl` to the login URL of your CAS server.

A.13. Debugging

If you need debug information, you can specify the following entries in `Config.groovy`:

Listing 56. Logging setup

```
log4j = {
    ...

    debug  'grails.plugin.springsecurity',
           'grails.app.controllers.grails.plugin.springsecurity',
           'grails.app.services.grails.plugin.springsecurity',
           'org.pac4j',
           'org.springframework.security'

    ...
}
```

A.14. Frequently Asked Questions

Why this token-based implementation? Can't I use HTTP basic authentication?

In theory you can. The only restriction to be truly stateless is to not use HTTP sessions at all. So if you go with basic authentication, you need to transfer the credentials back and forth every time.

Let's think about that. Keep in mind that your frontend is a pure HTML/Javascript application, consuming a REST API from the Grails side. So the first time, the Javascript application will make an API query and will receive a 401 response indicating that authentication is required. Then you present the user a form to enter credentials, you grab them, **encode** them with Base64 and in the next request, you send an HTTP header like `Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`.

Now remember you are doing RESTful application, so the session state is maintained in the client. That means that you would need to store that Base64 encoded string somewhere: cookies? HTML5 local storage? In any case, they are accessible using browser tools. And that's the point: there is a huge security risk because Base64 it's not encryption, just encoding. And it can be easily decoded.

You could argue that someone can access the token in the browser. Yes, but having the token will not allow him to obtain user's credentials. The tokens are just not decodable. And they can be revoked if necessary.

Fortunately for you, a token-based solution is not a magic idea that I only got; it's actually a specification: RFC 6750 - The OAuth 2.0 Authorization Framework: Bearer Token Usage.

Moreover, if you use tokens, you have the chance to implement expiration policies.

A couple of link with further explanations on the token-based flow:

- http://www.jamesward.com/2013/05/13/securing-single-page-apps-and-rest-services
- http://blog.brunoscopelliti.com/authentication-to-a-restful-web-service-in-an-angularjs-web-app

Why can't the API be secured with OAuth?

RFC 6749 - OAuth 2.0 specification does cover this scenario in what they call "public clients":

> Clients incapable of maintaining the confidentiality of their credentials (e.g., clients executing on the device used by the resource owner, such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

The OAuth 2.0 specification supports public clients with the implicit grant. This plugin supports that by default when you delegate the authentication to another OAuth provider. If it's you who are authenticating the users (via DB, LDAP, etc), the token-based flow of this plugin is *OAuth-ish*.

Why you didn't use any of the existing OAuth plugins? Why pac4j?

I'm aware of plugins like OAuth and Spring Security OAuth, but all of them rely on Spring Security Core's way of using HTTP sessions. So not acceptable.

I chose pac4j because:

1. They support major OAuth 2.0 providers out-of-the-box, whereas Scribe does not.
2. It's deadly simple and works just fine.

I'm also aware of a pac4j-spring-security module. See my previous response on HTTP sessions.

Dude, this is awesome. How can I compensate you?

I doubt you can :). You may try giving me free beers the next time you see me in a conference. Or you can just express your gratitude via Twitter.

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.
- [groovy] Groovy programming language.
- [grails] Grails framework.
- [oauth2] Internet Engineering Task Force (IETF) Request for Comments 6749. The OAuth 2.0 Authorization Framework. October 2012.
- [asciidoctor] Asciidoctor project.
- [gradle] Gradle project.
- [spring-security] Spring Security.
- [ss-core] Spring Security Core Grails plugin.