Table of Contents

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

Introduction

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

The Groovy programming language

Groovy [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`.

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.

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)

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"
```

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>
```

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.

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.

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!"
```

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!
```

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.

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.

Conclusion

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

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.

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.