Table of Contents

This chapter contains a detailed description of the analysis performed, the user stories extracted and implemented, as well as some notes about the development process.

Introduction

The methodology used to develop this project is **Scrum**.

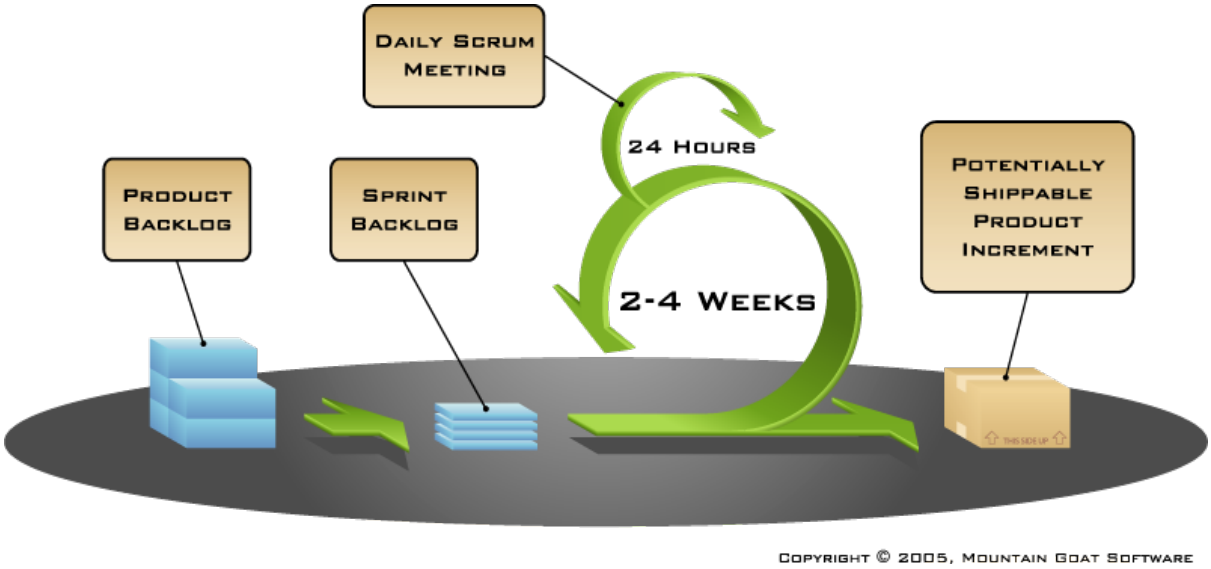


Figure 1. Scrum overview by Mountain Goat Software

Scrum is an agile process most commonly used for product development, especially software development. Scrum is a project management framework that is applicable to any project with aggressive deadlines, complex requirements and a degree of uniqueness. In Scrum, projects move forward via a series of iterations called sprints. Each sprint is typically two to four weeks long.

Scrum defines the following concepts:

- **Scrum team**: A typical scrum team has between five and nine people, but Scrum projects can easily scale into the hundreds. However, Scrum can easily be used by one-person teams and often is. This team does not include any of the traditional software engineering roles such as programmer, designer, tester or architect. Everyone on the project works together to complete the set of work they have collectively committed to complete within a sprint. Scrum teams develop a deep form of camaraderie and a feeling that “we’re all in this together.”
- **Product owner**: The product owner is the project’s key stakeholder and represents users, customers and others in the process. The product owner is often someone from product management or marketing, a key stakeholder or a key user.
- **Scrum Master**: The Scrum Master is responsible for making sure the team is as productive as possible. The Scrum Master does this by helping the team use the Scrum process, by removing impediments to progress, by protecting the team from outside, and so on.
- **Product backlog**: The product backlog is a prioritized features list containing every desired feature or change to the product. Note: The term “backlog” can get confusing because it’s used for two different things. To clarify, the product backlog is a list of desired features for the product. The sprint backlog is a list of tasks to be completed in a sprint.
- **Sprint planning meeting**: At the start of each sprint, a sprint planning meeting is held, during which the product owner presents the top items on the product backlog to the team. The Scrum team selects the work they can complete during the coming sprint. That work is then moved from the product backlog to a sprint backlog, which is the list of tasks needed to complete the product backlog items the team has committed to complete in the sprint.

- **Daily Scrum**: Each day during the sprint, a brief meeting called the daily scrum is conducted. This meeting helps set the context for each day's work and helps the team stay on track. All team members are required to attend the daily scrum.
- **Sprint review meeting**: At the end of each sprint, the team demonstrates the completed functionality at a sprint review meeting, during which, the team shows what they accomplished during the sprint. Typically, this takes the form of a demonstration of the new features, but in an informal way; for example, PowerPoint slides are not allowed. The meeting must not become a task in itself nor a distraction from the process.
- **Sprint retrospective**: Also at the end of each sprint, the team conducts a sprint retrospective, which is a meeting during which the team (including its ScrumMaster and product owner) reflect on how well Scrum is working for them and what changes they may wish to make for it to work even better.

Product Backlog

The software requirements are defined through Scrum's **user stories**.

User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system. They typically follow a simple template:

> As a <type of user>, I want <some goal> so that <some reason>.

User stories are often written on index cards or sticky notes, stored in a shoe box, and arranged on walls or tables to facilitate planning and discussion. As such, they strongly shift the focus from writing about features to discussing them. In fact, these discussions are more important than whatever text is written.

#1. As a user, I want to authenticate, so that I can access protected resources

The authentication endpoint must enable a client application to login using credentials such as username and password. It should be RESTful, i.e.: it should consume and produce JSON over HTTP. An example request is like:

Listing 1. User Story 1: authentication endpoint.

```
POST /login HTTP/1.1
Host: api.example.com
Content-Type: application/json
{
    "username": "admin",
    "password": "secret"
}
```

If the authentication is successful, the response should be like:

Listing 2. User Story 1: successful authentication response

```
200 OK HTTP/1.1
Content-Type: application/json
{
   "username": "admin",
   "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"
}
```

If the authentication fails, the response should be a `401 Unauthorized` HTTP status code.

If the HTTP method in the request is not `POST`, the response must be a `405 Method Not Allowed` HTTP status code.

If the request is invalid, the response shall be a `400 Bad Request` HTTP status code.

#2. As a user, I want to logout, so that my token is no longer valid

Logout is only possible with stateful token storages.

There must be a logout endpoint to allow users remove their token from the token storage. An example request would be like this:

Listing 3. User Story 2: logout endpoint

```
POST /logout HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ...
```

Note that the token is passed in the `Authorization` HTTP Header, as specified in the RFC 6750 [rfc6750].

If the token is found and removed, the response should be a `200 OK` HTTP status code. Otherwise, it will be a `404 Not Found`.

If the HTTP method in the request is not `POST`, the response must be a `405 Method Not Allowed` HTTP status code.

If the request is invalid, the response shall be a `400 Bad Request` HTTP status code.

#3. As a developer, I want to configure how access tokens are generated

Token generation should be configurable by users. There are 2 scenarios:

1. If using JWT, generated tokens will conform to the RFC 6750 [rfc6750] specification, and therefore it won't be possible to override it. In this case, token will have a fixed expiration time, that must be also configurable. The tokens will expire after the indicated period, regardless of having being accessed or not.
2. Otherwise, tokens will be a set of random alphanumeric characters of 32 char length, stored in any token storage (see next user story). In this case, tokens will have a configurable expiration time, that might be refreshed on every access, depending on the underlying storage support for such action.

The plugin must provide at least one implementation for each scenario.

#4. As a developer, I want to store tokens on multiple storages, including the client itself

If using JWT, tokens will be stored on the client. Assuming the client is a front-end application using a Javascript MVC framework like Angular JS [angular-js], the place to store such token would be HTML 5's `localStorage`.

Otherwise, the plugin must enable the developer to configure a stateful storage to store the tokens into. The plugin must provide at least one implementation for storing tokens in a relational database, and in an in-memory storage solution.

#5. As a user, I want to validate whether my access token is still valid or not

Token validation filter

The plugin must provide a Spring Security Filter (based on Java EE Servlet filters) that will validate the token received on every request. If the validation is successful, the plugin will restore the user principal into the Spring Security's `SecurityContext`.

This will enable developers to transparently secure their endpoints using Spring Security utils like the `@Secured` annotation or the `SpringSecurityService`:

Listing 4. User Story 5: securing a controller action

```
import grails.converters.JSON

class UserController {
    SpringSecurityService springSecurityService

    @Secured
    def me() {
        render [username: springSecurityService.principal.username] as JSON
    }
}
```

The above action might be accessed like this:

Listing 5. User Story 5: accessing a protected endpoint

```
GET /user/me HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ...
```

Assuming the token is valid, the response might be like this:

Listing 6. User Story 5: successful protected endpoint response

```
200 OK HTTP/1.1
Content-Type: application/json
{ "username": "raul.gonzalez" }
```

If the token is invalid, the response should be a `401 Unauthorized` HTTP status code, indicating that the request is not sufficiently authenticated.

Token validation endpoint

There must be a token validation endpoint available for users to check whether a token is still valid or not.

Example usage:

Listing 7. User Story 5: token validation endpoint request

```
POST /validate HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ...
```

If the token is valid, the response should be like:

Listing 8. User Story 5: token validation endpoint response

```
200 OK HTTP/1.1
Content-Type: application/json
{
   "username": "admin",
   "roles": [
      "ROLE_ADMIN",
      "ROLE_USER"
   ],
   "expires_in": 3600,
   "token_type": "Bearer",
   "refresh_token": "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ...",
   "access_token": "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ..."
}
```

#6. As a developer, I want to expose CORS headers in the responses, so that the system can be used from separated front-end applications

Cross-Origin Resource Sharing [cors] is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the resource originated.

In the context of separated front-ends and backends, it might be possible that the front-end is running on a different domain (`www.example.com`) than the backend (`api.example.com`).

When CORS request is needed, the browser will issue first a so-called pre-flight request, asking the server for its CORS capabilities:

Listing 9. User Story 6: pre-flight request in CORS.

```
OPTIONS /user/me HTTP/1.1
Origin: http://www.example.com
Access-Control-Request-Method: GET
Host: api.example.com
Accept-Language: en-GB
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

If the server has CORS enabled, it will indicated allowed origins and HTTP methods in the response:

Listing 10. User Story 6: pre-flight response in CORS.

```
Access-Control-Allow-Origin: http://www.example.com
Access-Control-Allow-Methods: GET, POST, PUT
```

The browser will then check if the origin domain and the HTTP is allowed, and will make the actual request. If the server disallows any of them, the browser will halt the request, throwing an error in the console.

The plugin must offer support for enabling CORS.

#7. As a user, I want to authenticate using external OAuth providers such as Google or Facebook

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:

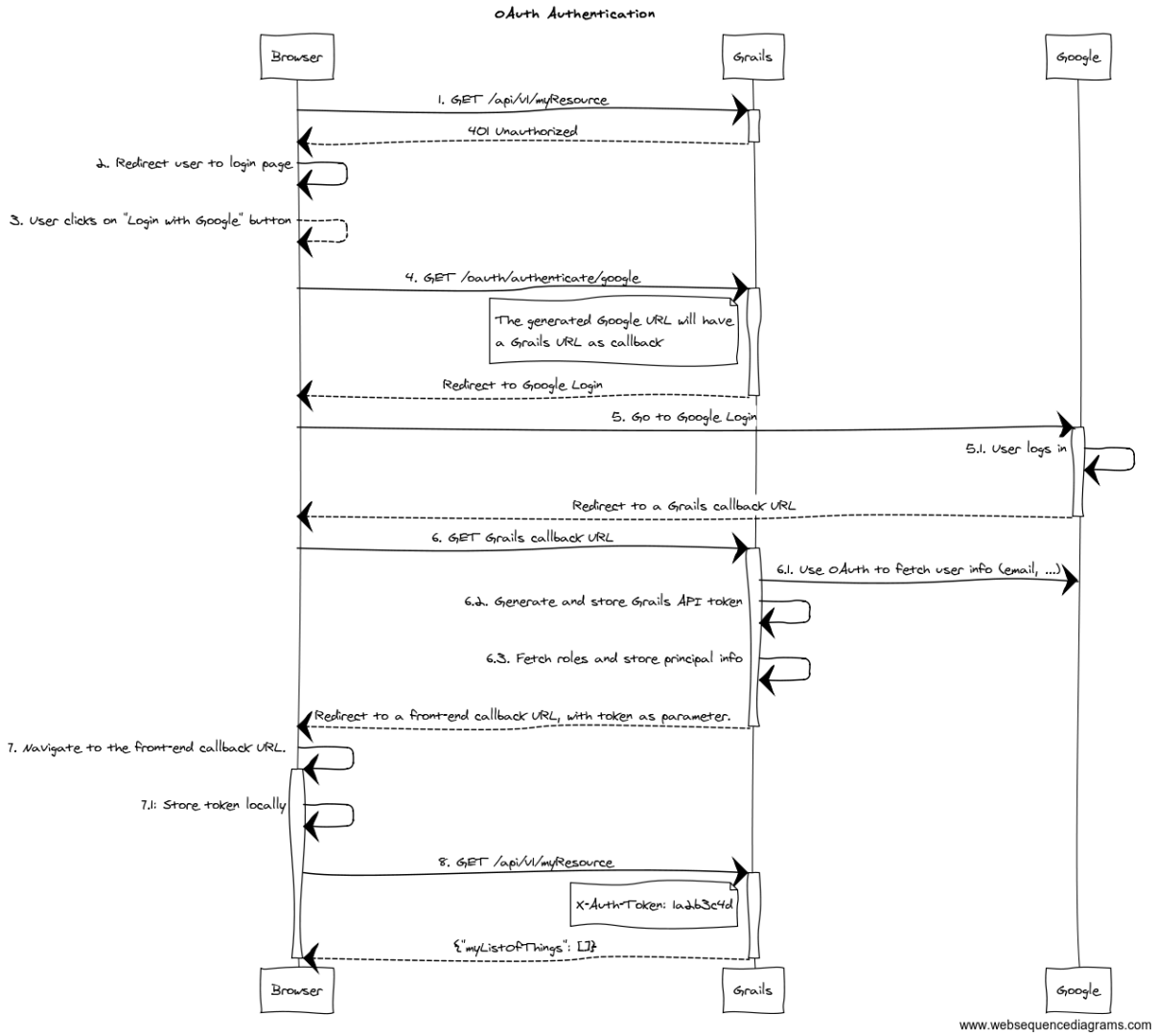


Figure 2. OAuth delegation protocol

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 plugin must provide configurable support to delegate the authentication to, at least, Google, Facebook and Twitter.

Legal framework

The applicable legislation to any software application using this plugin is the Data Protection Act: *"Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal"* [lopd]. The law stablishes a series of protection levels depending on how sensitive is the data stored.

Basically, any data subject to the most basic security level (*"nivel básico"*) must be accessed by uniquely identified users. Organisations have also the obligation to grant confidentiality and integrity of the passwords stored.

In the case of Spring Security, the library offers sufficient mechanism to be compliant with the lay restrictions. However, is the developer's responsibility to ensure that the configuration applied matches the required security restrictions by the law.

Spring Security Core plugin (foundation for the Spring Security REST plugin) uses the **bcrypt** [bcrypt] algorithm to hash passwords. In addition, developers can increase the security of the passwords by adding a salt.

Bcrypt is a much more secure alternative to the message digest approaches since it supports a customizable work level which when increased takes more computation time to hash the users' passwords, but also dramatically increases the cost of brute force attacks. Given how easy it is to use GPUs to crack passwords, developers should definitely consider using bcrypt for new projects and switching to it for existing projects. Note that due to the approach used by bcrypt, developers cannot add an additional salt like with the message digest algorithms.

It can be enabled by using the `bcrypt` value for the algorithm config attribute:

```
grails.plugin.springsecurity.password.algorithm = 'bcrypt'
```

Design

Use cases

In software and systems engineering, a use case is a list of action or event steps, typically defining the interactions between a role (known in UML as an **actor**) and a system, to achieve a goal. The actor can be a human, an external system, or time. In systems engineering, use cases are used at a higher level than within software engineering, often representing missions or stakeholder goals.

In UML, use cases are represented through use cases diagrams. They containg the following key elements:

- **Use cases**. A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse.
- **Actors**. An actor is a person, organization, or external system that plays a role in one or more interactions with your system. Actors are drawn as stick figures.
- **Associations**. Associations between actors and use cases are indicated in use case diagrams by solid lines. An association exists whenever an actor is involved with an interaction described by a use case. Associations are modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line. The arrowhead is often used to indicating the direction of the initial invocation of the relationship or to indicate the primary actor within the use case. The arrowheads are typically confused with data flow and as a result some people avoid their use.

Example:

[use case example] | *use-case-example.png*

Figure 3. Use case example

User use cases

[use cases user] | *use-cases-user.png*

Figure 4. User use cases

Developer use cases

[use cases developer] | *use-cases-developer.png*

Figure 5. Developer use cases

Class diagrams

Below are included some class diagrams of the most important parts of the system.

Filters

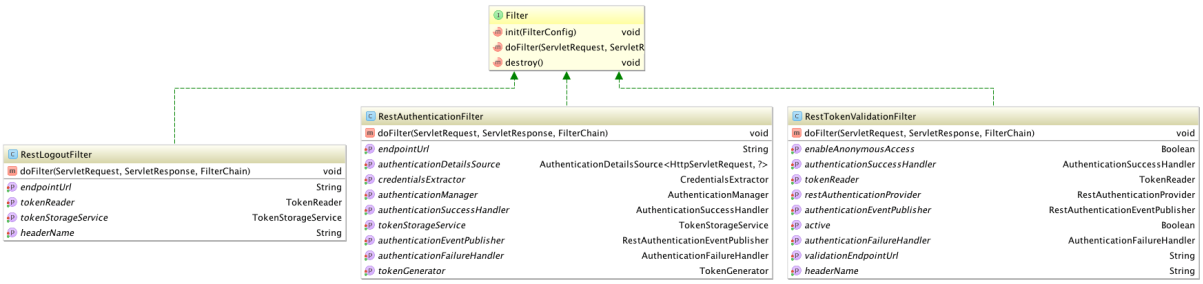


Figure 6. Filters class diagram

The provided filters, as specified in the user stories, implement the interface `javax.servlet.Filter`, where the important method is `doFilter(...)`.

Events and listeners

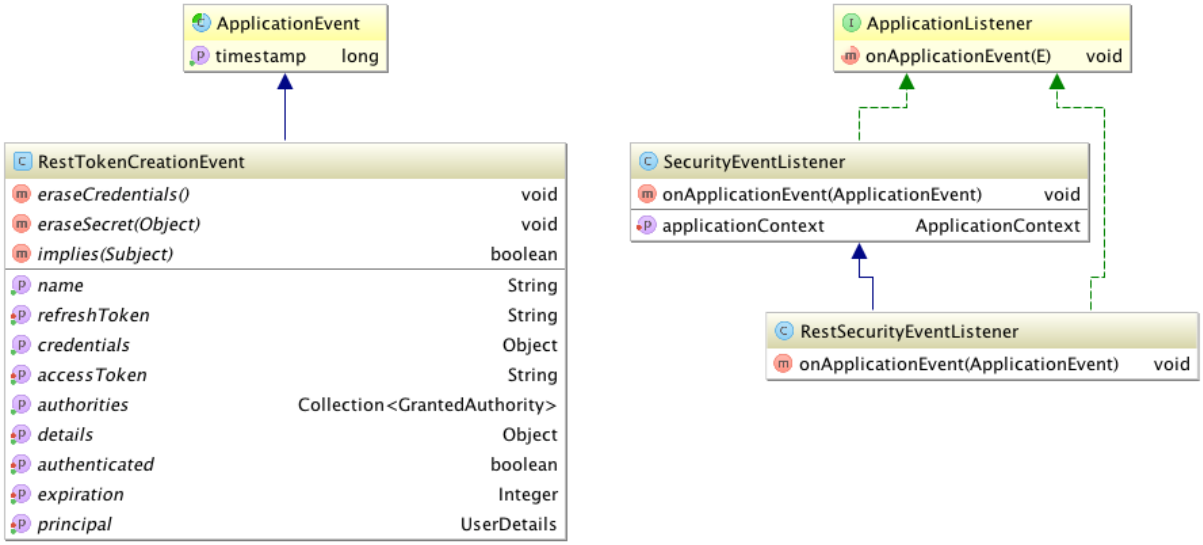


Figure 7. Events and listeners class diagram

Spring framework provides an eventing system, allowing application to register custom listeners. Spring Security implements such interface, defining a generic `SecurityEventListener`. This plugin provides a concrete `RestSecurityEventListener`, that handles `RestTokenCreationEvent`'s.

Providers and handlers

Figure 8. Providers and handlers class diagram

`AuthenticationProvider` is a Spring Security interface to define pluggable authentication mechanisms. This plugin provides one `RestAuthenticationProvider` to authenticate requests based on an access token.

Depending on the result of the authentication, it might be delegated to either `RestAuthenticationSuccessHandler` or to `RestAuthenticationFailureHandler`.

Credentials extraction

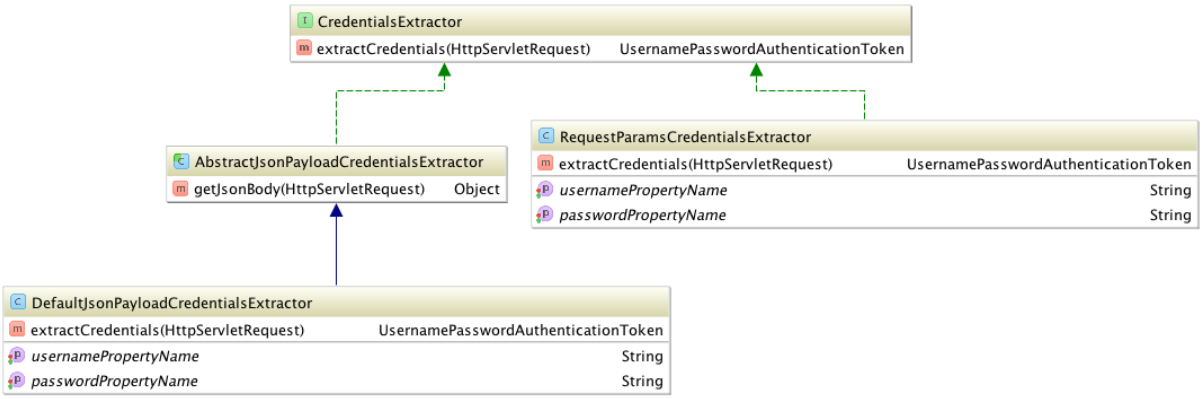


Figure 9. Credentials extraction class diagram

In the authentication filter (login), credentials can be extracted from different sources. Mainly, from a JSON payload and from request parameters. However, the plugin is flexible enough to allow the users to plug in their own implementation.

Token generation

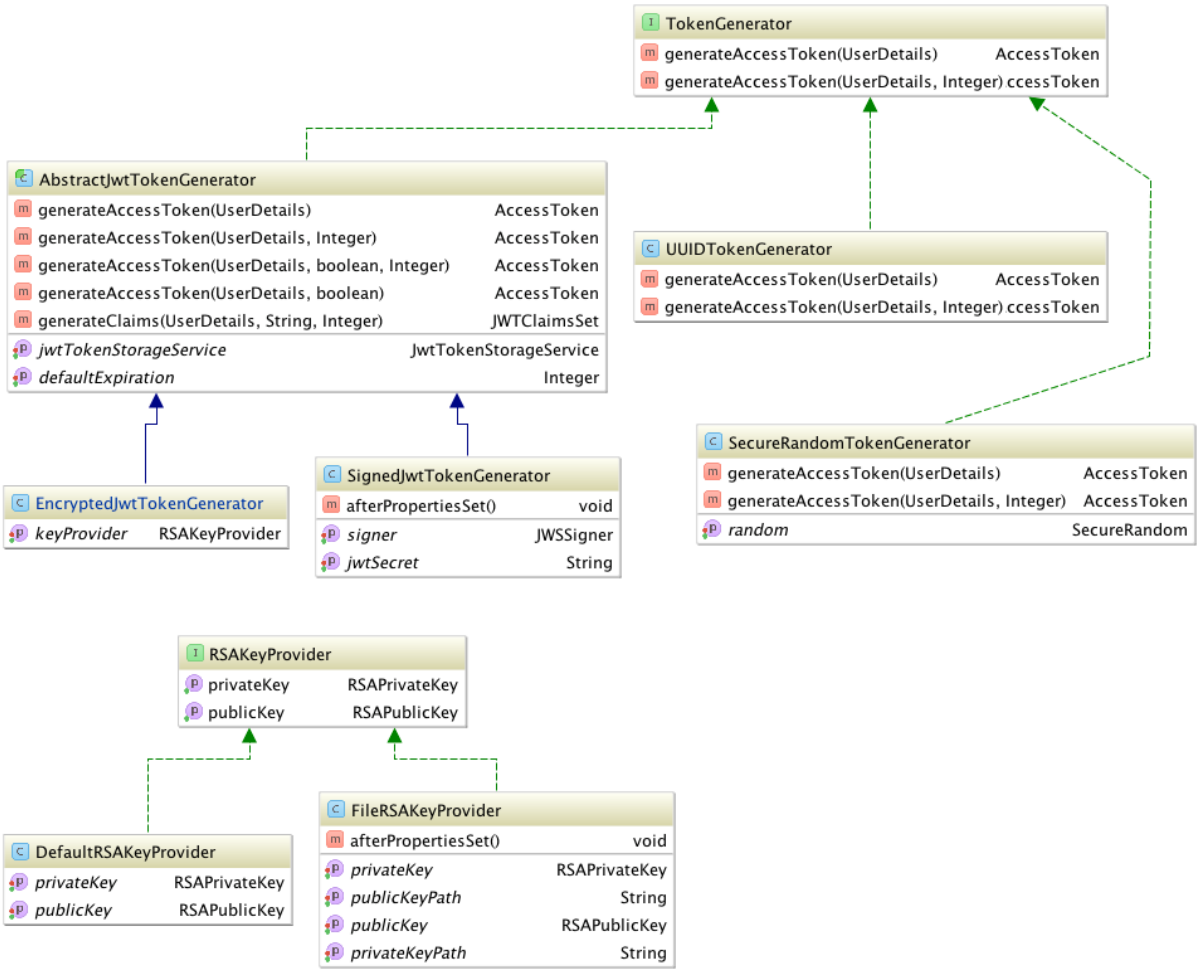


Figure 10. Token generation class diagram

As specified in the user stories, token generation is also pluggable. The plugin offers implementation for both stateless tokens (JWT) and non-stateless. In the case of JWT, there are also 2 options: signed and encrypted tokens.

Token storage

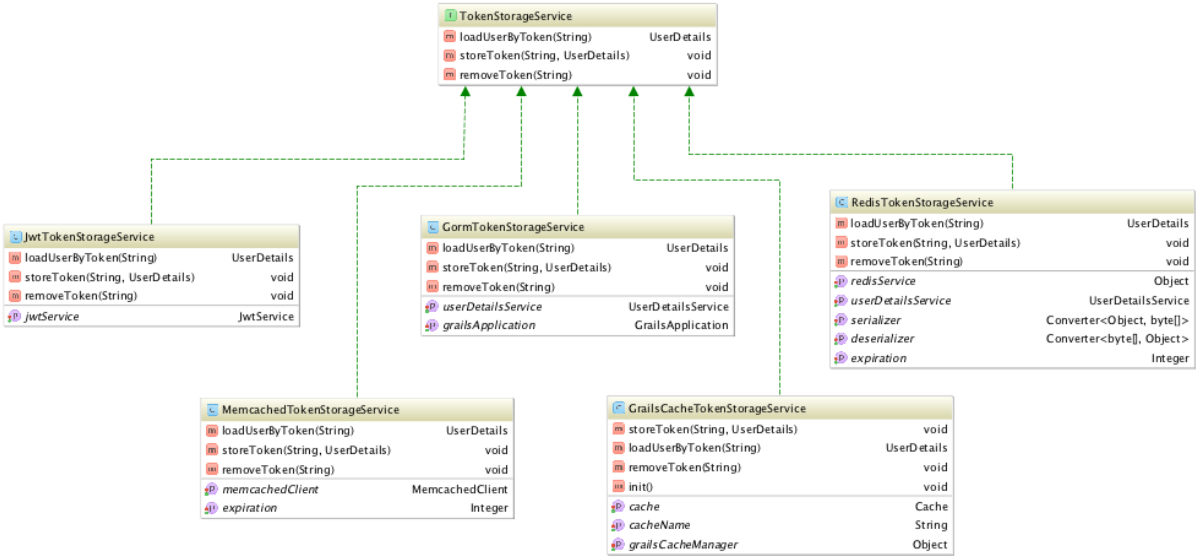


Figure 11. Token storage class diagram

Here we can see the different implementations for token storage:

- **Stateless**: JWT.
- **Stateful**: Memcached, GORM (database), Grails cache plugin (which uses Spring Cache abstraction) and Redis.

Development

Creating the plugin to its current state has taken 2 years and 37 releases. It has been an enormous effort.

The first stages of the development were invested in learning the internals of Spring Security. I had previous experience with it as a user, and in fact, is not the fist plugin I have created on top of it (in 2011 I wrote a plugin for Spring Security SAML). Spring Security is a very well designed yet complex library, with a lot of custom concepts and interfaces, such as `Authentication`, `AuthenticationProvider`, `Filter` and so on. It took me a while to figure out the best way to achieve my goals.

My initial approach was to strip down many things I didn't need nor want from Spring Security, such as the form-based authentication or the dependency on the HTTP Session. That resulted in a plugin that was perfectly fitted for being used from a separated front-end application. As a consequence, it effectively disabled old Spring Security ways of authentication.

However, it turned out that users were including it in applications where both traditional and REST authentication was used. Thus, I revisited my approach and allowed the users to use my features on a per filter-chain basis (see Chapter A.4 in the appendix).

Testing

Considering that it's a plugin that people use in their applications, stability and robustness has always been a key factor. And they use it for one of the most critical aspects in an application: security.

I have invested a significantly amount of time of the development in writing tests, at all levels: unit, integration and functional. Nothing can prove the absence of all bugs, true. However, it's certainly possible to prove the absence of many common classes of bugs.

It's important to note that, in a HTTP-based security system you need to test using an HTTP client. If you mock such interfaces, you have the risk of not testing the system under real conditions, and this is not acceptable for a security component.

The functional testing part of the project is composed of 3 full Grails applications with the plugin installed that perform hundreds of HTTP requests and specifies assertions on the results. Not only there are a considerable amount of tests across the project, but also they all run automatically on every push to GitHub, so I can ensure that no regressions are introduced by having an automated Continuous Integration system running on Travis CI [travis].

Tests are written using Spock [spock] and Geb [geb]. This is how a functional test look like:

Listing 11. Sample functional test

```
@IgnoreIf({ !SpringSecurityUtils.securityConfig.rest.oauth.facebook ||
!System.getenv('FB_PASSWORD') })
class FacebookSpec extends GebReportingSpec {

    void "it can sign users in with Facebook"() {
        when: "a user clicks on a 'Sign In with Facebook' button"
        go "/jwt/oauth/authenticate/facebook"

        then: "its redirected to Facebook Sign In page"
        FacebookSignInPage facebookSignInPage = at FacebookSignInPage

        when: "credentials are entered"
        facebookSignInPage.login 'open_pmazedy_user@tfbnw.net',
System.getenv('FB_PASSWORD')

        then: "is redirected to the frontend callback URL, with a token"
        FrontendCallbackPage frontendCallbackPage = at FrontendCallbackPage
        frontendCallbackPage.jsUrl.contains("token")
    }

}
```

The above test executes over a running testing Grails application with the plugin installed, opens a session in the headless browser Phantom JS [phantom-js], and interacts with the web as if it were a real user, filling forms, clicking buttons, etc.

Figure 12. Unit test report

Continuous integration and automated release

The continuous integration pipeline is a complex process, totally automated. It runs all the suite of tests across all the testing applications whenever something is pushed to GitHub. This ensures that the branches are stable all the time.

Current status of the build can be seen at https://travis-ci.org/alvarosanchez/grails-spring-security-rest:

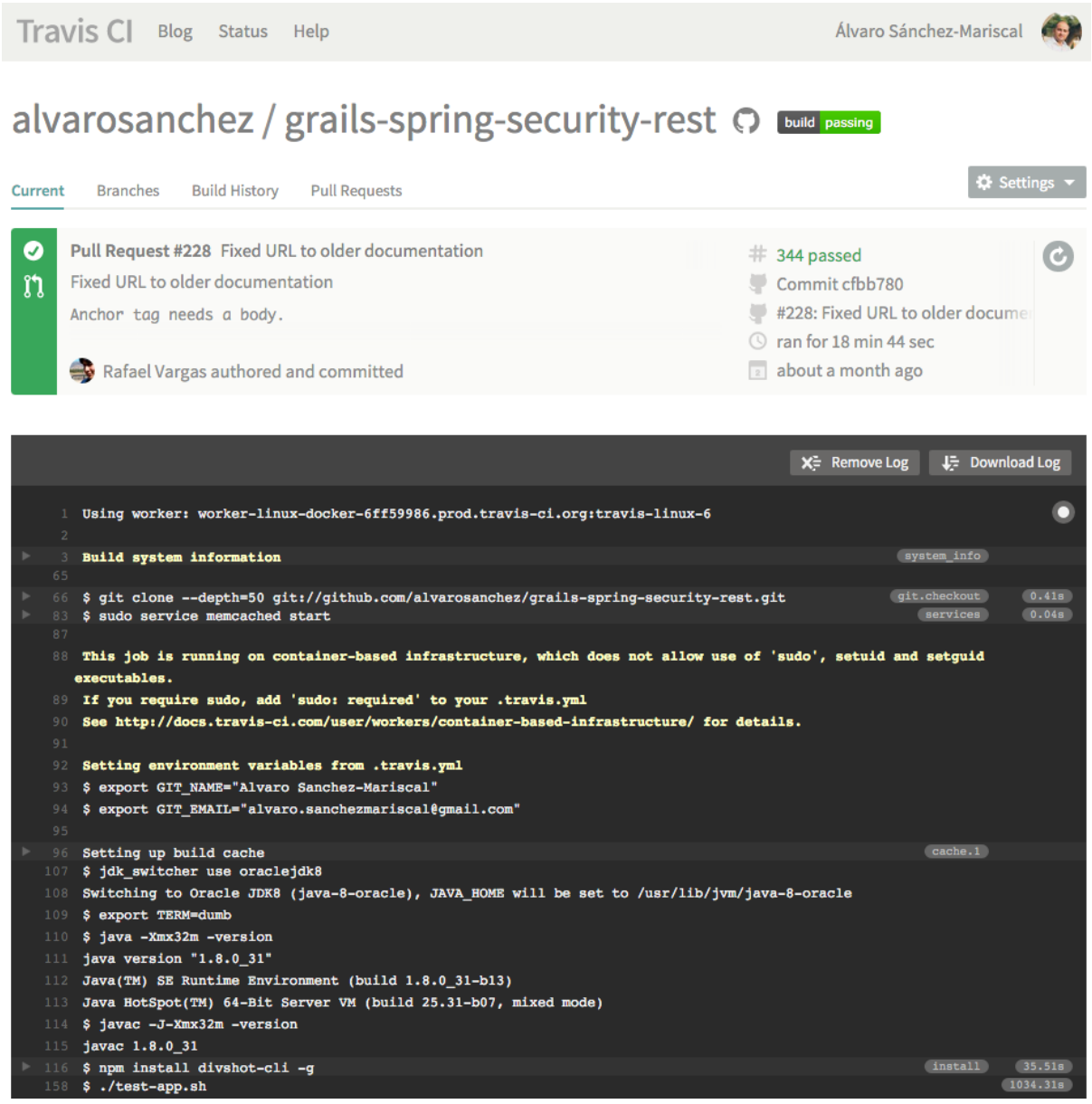
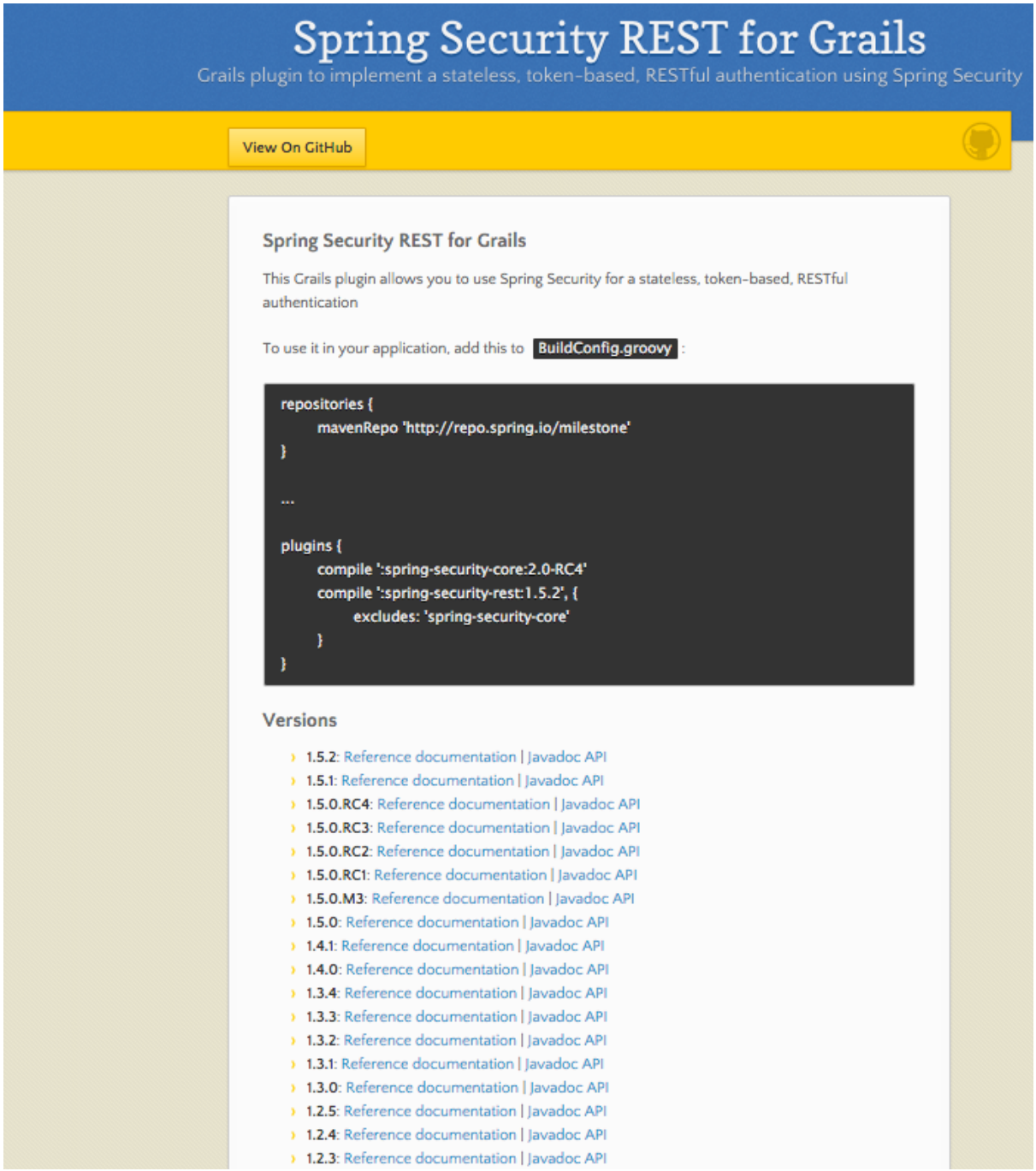


Figure 13. Travis build status

Installation and usage

Documentation has been one of the key factors contributing to the success of this plugin. It's comprehensive, covers all features, has configuration examples and it's also automatically published online on every release. Its publishing is also automated and versioned: http://alvarosanchez.github.io/grails-spring-security-rest



For reference, it has been included in the appendix of this document. Please refer to that section for more information.