

Table of Contents

Introduction	1
Product Backlog	3
Design	13
Development	16
Deployment	17

Introduction

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

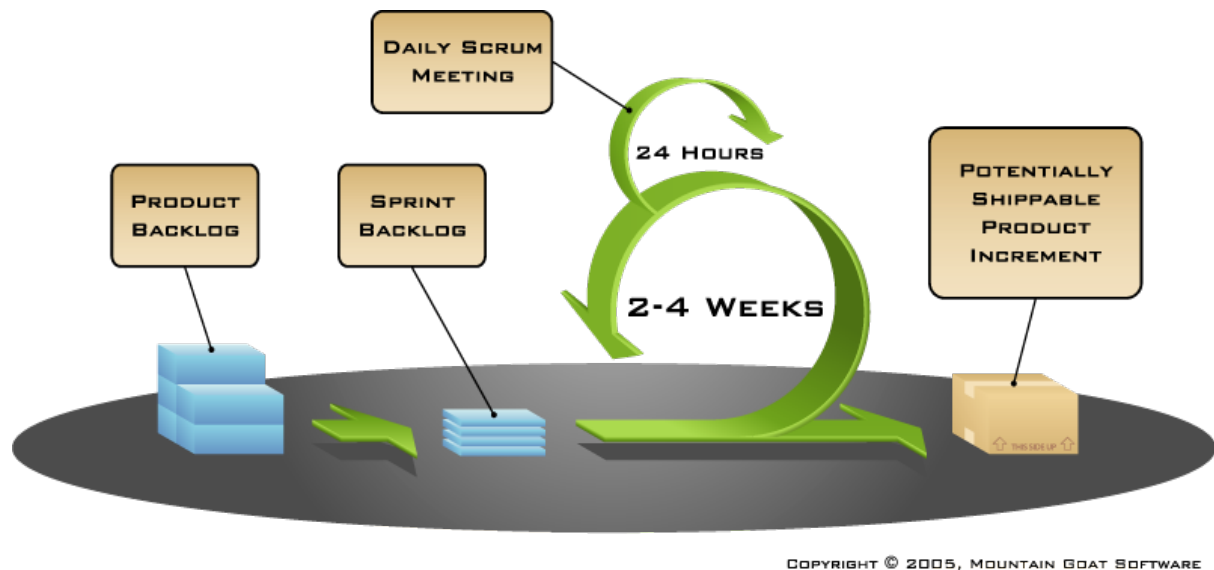


Figure 1. Scrum overview by [Mountain Goat Software](#)

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.

User stories

#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":
"eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJhNTZHQ00ifQ.fUaSWIdZakFX7CyimRIPhuw0sfevgmwL2x
zm5H0TuaqwKx24EafC00TruGKG-1N-
wGCITssnF2LQTqRzQgP0PoLXHfUJ0kkz5rB16LtnRu7cdD1ZUNYXLJtFjQ3IATzoo15tPafRPyStG1Qm7-
1L0VxquhrLxkkpti0F1_VTyTZAq8ltFrnxM4ahJUwS7eriivvdLqmHtnwuXw0kBXEseIyCkiyKk1WDJAcD
_P_gHoQJvSCoXedlr7Pp0n6LEUrRWJ2Hb-
Zyt9dWqWDxm9nyDeEVtEZGcQtpgCGgbXxaUpULIy5nvrBzXSnyT6iXhK1CLqiFVkfH-Y-
DHXdB6Q4sg.uYdpX1835KnlkqC5.gBgSnPWZ0o6FINovJNG7Xx2RuS09QJbU4-_J4EgZQkygt8xE-
HfdYa0mtmJLjGJR1XKoaRsuX1gNjFoCZgqWAon6.Zsrk52dkjSkSVQLXZBQooQ",
  "access_token": "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJhNTZHQ00ifQ.n-
gGe65x0SLSXS3fTG8ZLdXvv6b5_1pDvkcGyCjFy-
vm1VhaBEQL5p3hc6iUcCAcuYrQzGk95LV9dHCv46cNfCiUFHWfbEcd4nqScIxBbc28x09L1mNLnZ0G1rx1
Mx1L0Y_ZPoSxDXpJaHCT28cdZffHLxx2B9ioIClGdLYBAJ50z8VT39-
D0QSomS6QhFqmcPbDsXrsKxs545Pn-TIlu-fSQ4wpIvAxusOKB6CV2EYKqBp1MBRh-
3btE8WksVcX2N3LsrcMhrKxSKi93c06MZh6JzSLWe5b19hvUvBdEuWDrk-
fQgD3ZlmjjoevRWYhv_kslW1PlqUHYmKOQ7csUw.3mvvsFWikEjZzExA.YixjnnzzcPRY_uUpgPv5zq0fs
hv3pUwfrME0AijpsB7u9CmJe94g6f2y_3vqUps-5weKKGZyk3ZtnwEbPVAk9-HZt-
Y27SbZ14JNCFEOLVsMsK8.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 eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJkEYNTZHQQ00ifQ...
```

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 eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkpEYNTZHQ00ifQ...
```

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:

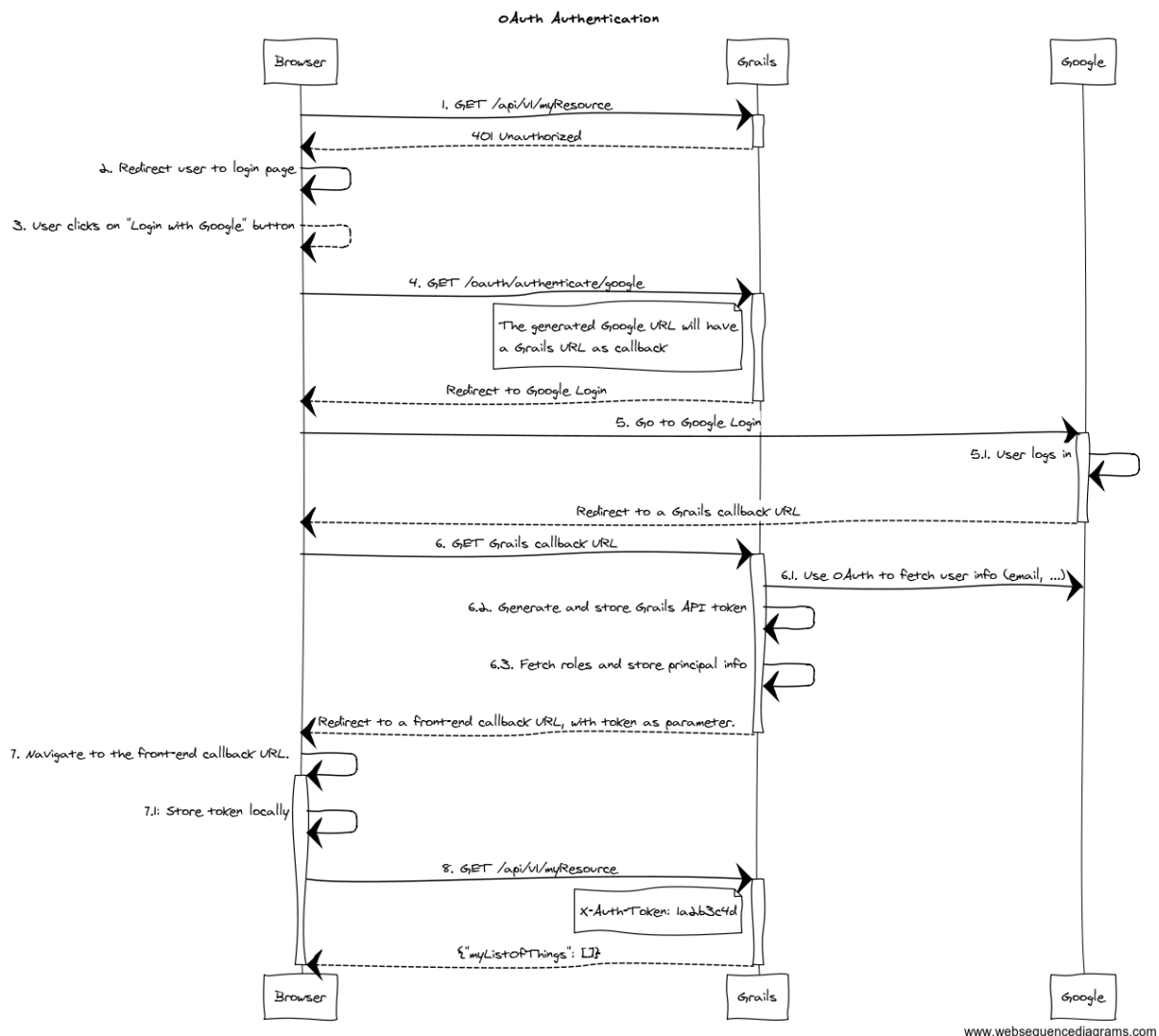


Figure 2. OAuth delegation protocol

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.

Design

As this project is not an application, but a plugin to create other applications, there are no use cases to be listed here. Therefore, below are specified the class diagrams of the different components of the system.

Filters

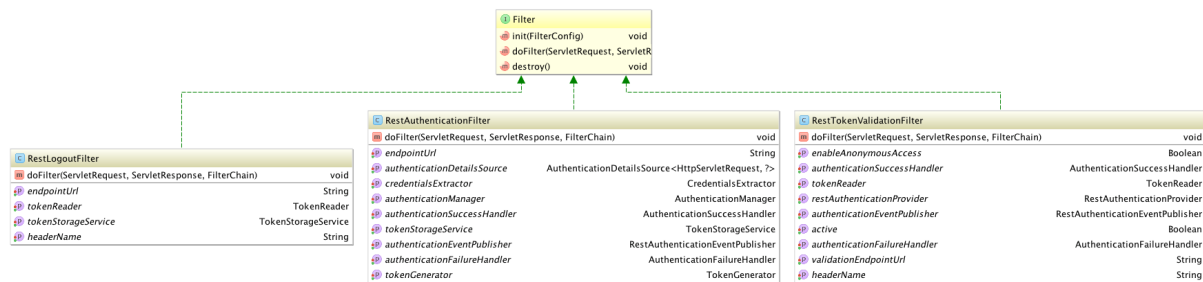


Figure 3. Filters class diagram

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

Events and listeners

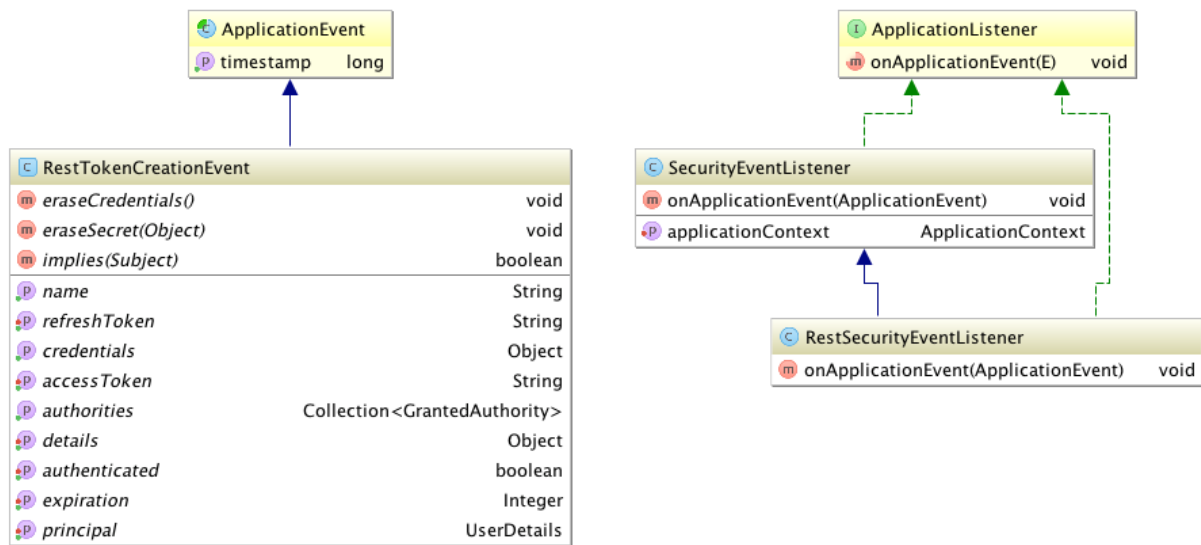


Figure 4. Events and listeners class diagram

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 5. 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.



Development



Deployment