Table of Contents

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:

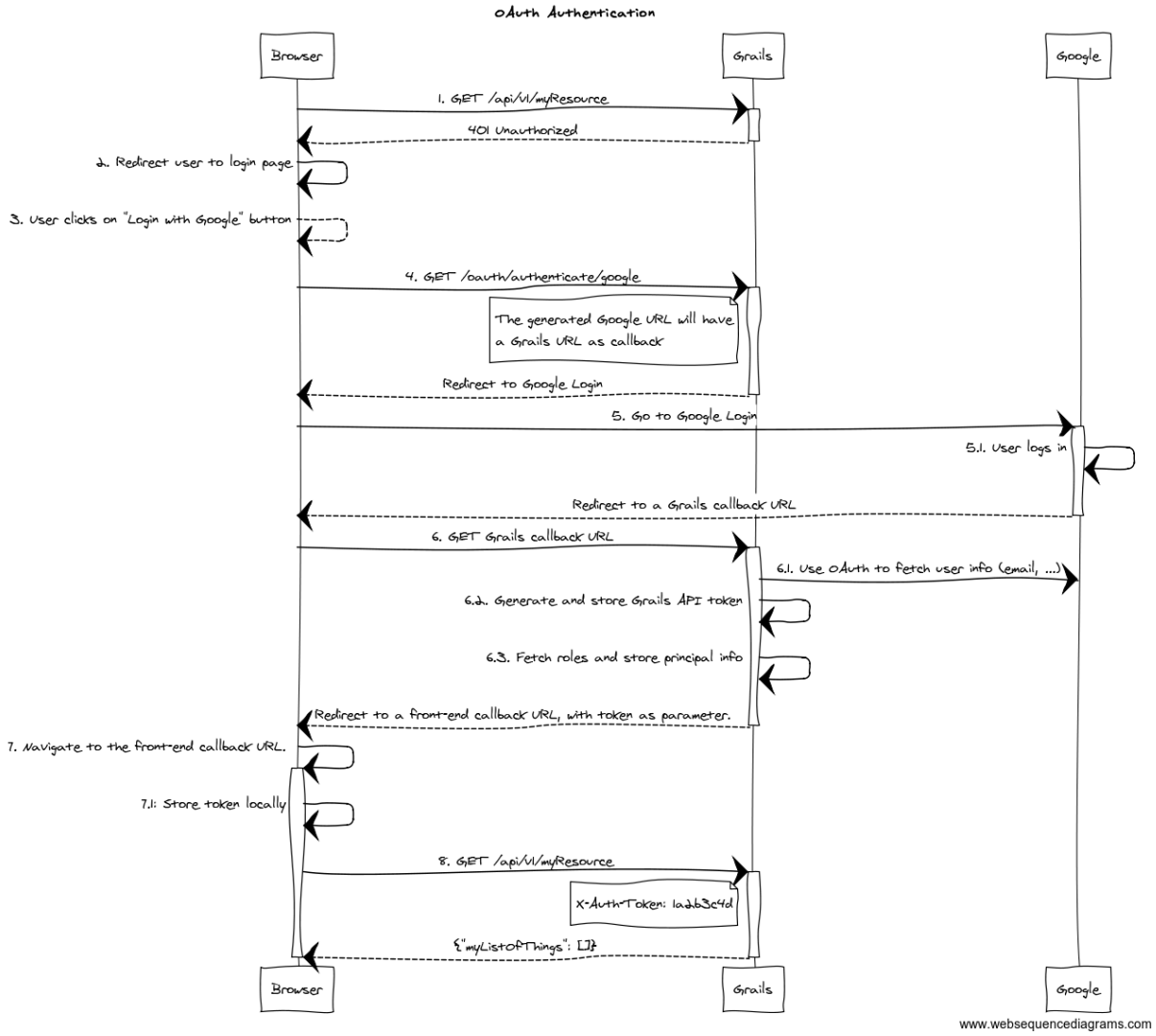

Figure 1. OAuth delegation protocol

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 1. 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 2. 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%40gmai
l.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 3. 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 4. 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 5. 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 6. 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.