View on GitHub (https://github.com/swisspush)

swisspush

Community Contributions by Developers in the Swiss Postal IT

(/)

OAuth2 in depth: A step-by-step introduction for enterprises

17 October 2016 - Federico Yankelevich

During the last couple of months I have been discussing OAuth2 usage in enterprises with a few friends and developers at our local meetups. People clearly understand the basic concepts behind it. They use it to login with their gmail, twitter or facebook accounts on third party websites. So they have an idea, but … … when speaking about how to implement it inside the company, a lot of misunderstanding occurs. There are so many things you can configure and options to activate, every small detail may have a big impact on the solution.

Usually in today's enterprise there are multiple scenarios when authorization needs to be handled:

- Service accounts used for server-to-server communication
- Mutual authentication via certificates between trusted servers
- SSO across multiple web applications (for example using SAML)
- Mobile apps that want to call private APIs
- B2B integration with partner companies

OAuth2 is a standard protocol that supports all of the above scenarios, but each one requires a specific configuration and usage. Introducing OAuth2. We will be focusing on one scenario at a time. This makes the adoption easier and more robust.

Migrating to OAuth2

Planning to introduce OAuth2 to one application at time, following their natural release cycle, reduces risks and allow better testing. Some services might need to have 2 security mechanisms in parallel during the migration phase.

It requires some time to find the right granularity and conventions to match your business case. A few iterations may be required to find an acceptable landscape and to build a secure enterprise on top of it. This article is just the beginning of a journey.

Motivation of this article

Many enterprises need to run a multitude of different applications that were written during the last 20 years, using many different technologies. This mixture limits the selection of security protocols to those that work everywhere, instead of evaluating them by their quality. Another big risk is that security libraries are rarely updated due to worries that it will be necessary to re-test the whole application.

I have the feeling that some enterprises have not yet introduced OAuth2 because they are concerned that it is too complex and "it does too many things we don't need" (now).

Within this blog post series, I'm focusing on an incremental approach for the introduction of OAuth2 in enterprises. This will start by solving existing problems and then, after some experience with this, look at some of the advanced features available that can fulfil new customer requirements and meet modern demands.

There is a lot of room to improve enterprise security and being able to do it step-by-step is probably the only way to carefully handle the wide impact of touching the security layer.

The goal

The simplest scenario we can start with is the Client Credential grant flow. It is used for server-to-server communication, a common use case in enterprises.

Today, many servers are verifying credentials on LDAP for every request they get. This generates traffic and increases password exposure. With a centralized Authorisation Server managing all the passwords and generating scoped tokens (with limited TTL) and with a Resource Server that can self-validate the token, we would already improve security.

This first step also allows us to:

- Test the OAuth2 infrastructure is working properly with our systems
- Build up knowledge in the team
- Be able to estimate migration costs of existing applications

In another blog post we will then add support for Mobile Applications and B2B solutions.

Note: For a production environment you have to consider how to integrate the Authorisation Server with your existing security infrastructure, and the tools needed to manage all the business processes for access control management. You may also need to consider how to migrate the data.

Hands on

Now, let's move to the coding part!

We've used spring-boot and spring-oauth to make it short and easy to read. Any Spring application can be configured to do the same, also if it is still using the XML configuration.

If you are not using Spring in your applications, there is a little bit more work to integrate OAuth2, and you probably have to integrate some of the Spring facilities yourself.

We considered Spring a valid base for our examples due to the vast adoption in the enterprise world.

Step 1 - Setup base OAuth2 infrastructure:

Using Spring Boot and Spring OAuth2 there are some very nice facility classes that allow us to create the infrastructure very quickly.

Project structure

Create the following projects structure with maven. OAuth2 step-by-step (parent POM) → Authorization Server (child module) → Resource Server (child module)

Instead of building the project from scratch (and check all maven dependencies, etc), we recommend to start from the Step1-InitialSetup branch of our project on GitHub:

```
> git clone https://github.com/exteso/oauth2-step-by-step oauth2-step-by-step
> cd oauth2-step-by-step
> git checkout Step1-InitialSetup
```

Create the Authorisation Server (AS)

Inside the authentication-server module, create a `SpringBoot` server class using the following code snippet:

```
@SpringBootApplication
public class AuthenticationServer {
   private static final Log logger = LogFactory.getLog(AuthenticationServer.class);

   public static void main(String[] args) {
      SpringApplication.run(AuthenticationServer.class, args);
   }

   @RequestMapping("/user")
   public Principal user(Principal user) {
      logger.info("AS /user has been called");
      logger.debug("user info: "+user.toString());
      return user;
   }
}
```

Then create an `OAuth2Config` class:

```
@Configuration
@EnableAuthorizationServer
public class OAuth2Config extends AuthorizationServerConfigurerAdapter {

  @Override
  public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
     clients.inMemory()
           .withClient("service-account-1")
                .secret("service-account-1-secret")
                .authorizedGrantTypes("client_credentials")
                .scopes("resource-server-read", "resource-server-write");
  }
}
```

Note: the current OAuth2Config is working in memory, for a realistic use-case you may need to integrate it with your company LDAP or use JDBC to access persistent data (user/pwd and clients configuration). SpringOAuth documentation covers all the options in details.

Finally configure the web server in the application.yml file:

```
server.contextPath: /auth
logging:
 level:
   org.springframework.security: DEBUG
server:
 port: 8080
```

Create a Resource Server (RS)

Once the Authentication Server is up and running, we want to create a Service that only allows access to authenticated users.

```
@SpringBootApplication
@RestController
@EnableResourceServer
public class ResourceServer {
   public static void main(String[] args) {
      SpringApplication.run(ResourceServer.class, args);
   }

   private String message = "Hello world!";

   @RequestMapping(value = "/", method = RequestMethod.GET)
   public Map<String, String> home() {
      return Collections.singletonMap("message", message);
   }

   @RequestMapping(value = "/", method = RequestMethod.POST)
   public void updateMessage(@RequestBody String message) {
      this.message = message;
   }

   @RequestMapping(value = "/user", method = RequestMethod.GET)
   public Map<String, String> user(Principal user) {
      return Collections.singletonMap("message", "user is: "+user.toString());
   }
}
```

We also need to instruct the Resource Server where it can go to verify that the token is valid. When we use the `@EnableResourceServer` annotation and we configure a `userInfoUri` property, spring-boot (by convention) calls the AS to get the fresh user data at any call (with the token) received by RS.

We just have to add the property in an `application.yml` of the RS file with the following syntax:

```
security:
 oauth2:
   resource:
     userInfoUri: http://localhost:8080/user
```

Time to test:

If you want to jump to the working example, just checkout the Step1-SetupDone branch

```
> git checkout Step1-SetupDone
```

Open a terminal and execute the following commands:

```
//launch Authorization Server on port 8080
> cd authorization-server;mvn spring-boot:run
```

Open another terminal and execute:

```
//launch Resource  Server on port 9090
> cd resource-server;mvn spring-boot:run
```

Then open another terminal to do test calls:

```
//login and get the access token
> curl service-account-1:service-account-1-secret@localhost:8080/auth/oauth/token -d grant_type=client_credentials
//check response contains the access_token
```

```
//save the access token you found in the response in an environment variable named TOKEN
> export TOKEN="7ffe37bd-a520-43b1-9724-18cda6580ed7"
//use the TOKEN when calling the ResourceServer
> curl -H "Authorization: Bearer $TOKEN" -v localhost:9090
//check response contains “Hello world!”
```

Try to call the POST method, changing the message content:

```
//use the TOKEN also when  calling POST on  the ResourceServer
> curl -H "Content-Type: application/json" -H "Authorization: Bearer $TOKEN" -X POST -d    “Bonjour monde” -v localhost:9090
> curl -H "Authorization: Bearer $TOKEN" -v localhost:9090
//check response contains “Bonjour monde”
```

What we have learnt:

- We have setup a super simple OAuth2 infrastructure to use it for our test.
- Users must be authenticated on the AS before they can make a request to the RS.
- The RS (Spring-OAuth2) automagically calls the UserInfo endpoint for each call with a token.
- The User information is also propagated to the RS. You can see it calling:

```
> curl -H "Authorization: Bearer $TOKEN" -v localhost:9090/user
//check response contains all the user information loaded from AS
```

Step 2 - Add user roles and RS access control rules:

Add roles to the user

Specify the roles of a user in AS OAuth2Config, adding 1 line at the end of the `configure(ClientDetailsServiceConfigurer clients)` method:

```
.authorities("ROLE_RS_READ");
```

Add access control rules to the Resource Server config

Modify the `ResourceServer` class, adding the blue lines below. This will check if the user has the proper rights to access each method:

```
@SpringBootApplication
@RestController
@EnableGlobalMethodSecurity(prePostEnabled = true)
@EnableResourceServer
public class ResourceServer {
   public static void main(String[] args) {
       SpringApplication.run(ResourceServer.class, args);
   }
   private String message = "Hello world!";

   @PreAuthorize("hasRole('ROLE_RS_READ')")
   @RequestMapping(value = "/", method = RequestMethod.GET)
   public Map<String, String> home() {
       return Collections.singletonMap("message", message);
   }

   @PreAuthorize("hasRole('ROLE_RS_WRITE')")
   @RequestMapping(value = "/", method = RequestMethod.POST)
   public void updateMessage(@RequestBody String message) {
       this.message = message;
   }

   @PreAuthorize("#oauth2.hasScope('resource-server-read')")
   @RequestMapping(value = "/user", method = RequestMethod.GET)
   public Map<String, String> user(Principal user) {
       return Collections.singletonMap("message", "user is: " + user.toString());
   }
}
```

As you can see with spring-oauth and spring-security we can use 2 different approaches for access control: - by ROLES → `.access("hasRole('ROLE_RS_WRITE')");` - by SCOPES → `.access("#oauth2.hasScope('resource-server-read')");`

Time to test:

If you want to jump to the working example, just checkout the Step2-AccessControl branch

```
> git checkout Step2-AccessControl
```

Open a terminal and execute the following commands:

```
//launch Authorization Server on port 8080
> cd authorization-server;mvn spring-boot:run
```

Open another terminal and execute:

```
//launch Resource  Server on port 9090
> cd resource-server;mvn spring-boot:run
```

Then open another terminal to do test calls:

```
//login and get the access token
> curl service-account-1:service-account-1-secret@localhost:8080/auth/oauth/token -d grant_type=client_credentials
//check response contains the access_token
```

```
//save the access token you found in the response in an environment variable named TOKEN
> export TOKEN="7ffe37bd-a520-43b1-9724-18cda6580ed7"
//call the / resource that is protected with ROLE_RS_READ
> curl -H "Authorization: Bearer $TOKEN" -v localhost:9090
//response contains “Hello world!” because the user has the role ROLE_RS_READ
```

```
//This time the POST call does not work because the user does not have the ROLE_RS_WRITE:
> curl -H "Content-Type: application/json" -H "Authorization: Bearer $TOKEN" -X POST -d 'Bonjour monde' localhost:9090
//returns an error: Access Denied because role ROLE_RS_WRITE was not assigned to the user
```

```
//Due to a known limitation of spring-oauth,  scopes are not loaded in RS:
> curl -H "Authorization: Bearer $TOKEN" -v localhost:9090/user
//returns an error: Insufficient scope because of a known limitation of UserInfoTokenServices in Spring-boot
```

For more information on this issue have a look at https://github.com/spring-projects/spring-boot/issues/5096 (https://github.com/spring-projects/spring-boot/issues/5096).

What we have learnt:

- Different clients can access different methods on RS depending on their roles.
- Client roles (authorities) are correctly passed from the AS to the RS at every call.
- If a token is revoked by AS, although it is still valid, it will not be accepted by RS (because they always call the AS).
- The default spring-oauth behaviour does not propagate OAuth2 scopes to the Principal in RS.
- Due to the OAuth2 specification we must assign some scopes to a client, but then we never use the scopes. What should/could we do with them?

Step 3 - AS returns a JWT with all UserInfo

If we want to reduce the number of calls from each Resource Server service to the AS, then a better solution is to introduce JWT (a token containing all the information inside) instead of using an reference tokens (that are just a reference to data hosted in a secure server).

Reference tokens generate many more calls to the AS to get userInfo, but this allows us to forbid execution as soon as a token gets invalidated. With JWT, if the token has not expired, it can be used by any RS without contacting the AS.

Additionally, since it may contain sensible data, if the token is sent outside a secure network it should be encrypted and signed.

For these reasons, it is always recommended to keep JWT inside a secure network, and when it has to be sent outside, to map it with an reference token (like sessionId in browsers).

Warning:

Never use JWT tokens for session inside a browser! Map it to a sessionid and use the sessionid inside a cookie (as we have done in the last 15 years) to grant proper security of webapps (see http://cryto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/).

Note: For B2B integration security can be increased pretending the servers are always talking through a secure channel (for example requiring Mutual Authentication). But this topic is not covered in this blog post.

Add JWT

Now with just few changes we can add JWT to our application.

First create a certificate with private/public keys with the following command:

```
> keytool -genkeypair -alias jwt -keyalg RSA -dname "CN=jwt, L=Lugano, S=Lugano, C=CH" -keypass mySecretKey -keystore jwt.jks -storepass mySecretKe
y
```

and save it in AS/src/main/resources. In the AS and RS modules pom files, add the dependency:

```
<dependency>
   <groupId>org.springframework.security</groupId>
   <artifactId>spring-security-jwt</artifactId>
</dependency>
```

In the AS module, `OAuth2Config.class` add the following lines:

```
@Autowired
private Environment environment;

@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
   endpoints.tokenStore(tokenStore())
     .tokenEnhancer(jwtTokenEnhancer())
            .authenticationManager(authenticationManager);
}

@Override
public void configure(AuthorizationServerSecurityConfigurer security) throws Exception {
   security.tokenKeyAccess("permitAll()")
    .checkTokenAccess("isAuthenticated()");
}

@Bean
public TokenStore tokenStore() {
   return new JwtTokenStore(jwtTokenEnhancer());
}

@Bean
protected JwtAccessTokenConverter jwtTokenEnhancer() {
   String pwd = environment.getProperty("keystore.password");
   KeyStoreKeyFactory keyStoreKeyFactory = new KeyStoreKeyFactory(
new ClassPathResource("jwt.jks"),
pwd.toCharArray());
   JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
   converter.setKeyPair(keyStoreKeyFactory.getKeyPair("jwt"));
   return converter;
}
```

and add the keystore.password configuration in the `application.yml` file of AS:

```
keystore:
 password: mySecretKey
```

In the RS module, in its `application.yml` file, we can set the endpoint for getting the public key of the certificate used to sign the token:

```
security:
 oauth2:
   resource:
     jwt:
       keyUri: http://localhost:8080/auth/oauth/token_key
```

Note: The configuration of the `userInfoUri` can now be removed from RS because it will not be used anymore.

Time to test:

If you want to jump to the working example, just checkout the Step3-UseJWT branch

```
> git checkout Step3-UseJWT
```

Open a terminal and execute the following commands:

```
//launch Authorization Server on port 8080
> cd authorization-server;mvn spring-boot:run
```

Open another terminal and execute:

```
//launch Resource  Server on port 9090
> cd resource-server;mvn spring-boot:run
```

Then open another terminal to do test calls:

```
//login and get the access token
> curl service-account-1:service-account-1-secret@localhost:8080/auth/oauth/token -d grant_type=client_credentials
//check response contains the NEW access_token, much longer and with scopes appended
```

```
//something like: {"access_token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzY29wZSI6WyJyZXNvdXJjZS1zZXJ2ZXItcmVhZCIsInJlc291cmNlLXNlcnZlci13cml0ZSJ
dLCJleHAiOjE0NzU3NTE0NzYsImF1dGhvcml0aWVzIjpbIlJPTEVfUlNfUkVBRCJdLCJqdGkiOiJkOGI1NTc1MS01YzJkLTRhNjItYmFlMy0yYjM0YTNjMzQ0NDkiLCJjbGllbnRfaWQiOiJzZX
J2aWNlLWFjY291bnQtMSJ9.d5eP533cYORNBt73vbXRSPowOefWvysoBr2lkazhcEjIK6wTRDv9-uO4Bi6CmRW6sBqo8ijiyPHBo596cyZpg6O94vRfI4FnFuqi9qzPc8B6CSeMoWJNf7g6sJUs
K1jrTZBs8_84MBmy2nDxC8DEYkOqwsBvh0FX9wOd3pLTlgl5_sh63D1E2RJsGhskYJb4ql9LZTuBI7KWV0MMYHTZ1QeaOWLMpnbalid5TSERHOsTMKgQNrJTC8ioet_lQJnXTbYIk2VkINyFX80
-RIobN4djlzs8oLEbkHWRT4t_O5vbc56AyvOaQZTPM8_C96VMLIOTuOrzP3rC3t7x7qp90A", "token_type":"bearer","expires_in":43199,"scope":"resource-server-read re
source-server-write","jti":"d8b55751-5c2d-4a62-bae3-2b34a3c34449"}
```

```
//save in an environment variable named TOKEN the access_token part found in the response
> export TOKEN="eyJhbGci….p90A"
```

```
//call the / resource that is protected with ROLE_RS_READ
> curl -H "Authorization: Bearer $TOKEN" -v localhost:9090
//response contains “Hello world!” because the user has the role ROLE_RS_READ
//when RS server is called there is not call to AS /user (check there is no log entry in AS)
```

```
//the POST call does not work because the user does not have the ROLE_RS_WRITE:
> curl -H "Content-Type: application/json" -H "Authorization: Bearer $TOKEN" -X POST -d    “Bonjour monde” -v localhost:9090
//returns an error: Access Denied because user has not the ROLE_RS_WRITE role
```

```
//the call to /user is instead now working because scopes are correctly propagated via JWT
> curl -H "Authorization: Bearer $TOKEN" -v localhost:9090/user
//response contains all the user visible information
```

What we have learnt:

- Different clients can access different methods on RS depending on their scopes.
- Client roles (authorities) and scopes are correctly sent inside JWT.
- RS is able to validate and read JWT without calling AS.

Cleanup

In order to have a more homogenous configuration we can now just use scopes instead of roles: - Configure all RS methods to use `#oauth2.hasScope()` instead of `hasRole()` - Remove `.authorities` from AS `OAuth2Config`.

Known issues:

If the resource server is started while the AS is down, the RS server is started but fails at run-time. See https://github.com/spring-projects/spring-security-oauth/issues/734 (https://github.com/spring-projects/spring-security-oauth/issues/734).

Step 4 - Call RS from a webapp

Great! We have a working environment for server-to-server calls. We now want to test another application (a server or a webserver), that needs to call our RS with a service account (non-personal).

First we quickly setup a WebServer using spring-boot.

Create a new module project called webapp-server (with the same pom of RS, just change the artifactId). Create an App class that makes it start as a WebServer:

```
@SpringBootApplication
@RestController
public class App  {

   @Autowired
   private OAuth2RestTemplate resourceServerProxy;

   public static void main(String[] args) {
       SpringApplication.run(App.class, args);
   }

   @RequestMapping(value = "/api/message", method = RequestMethod.GET)
   public Map<String, String> getMessage() {
       return resourceServerProxy.getForObject("http://localhost:9090", Map.class);
   }

   @RequestMapping(value = "/api/message", method = RequestMethod.POST)
   public void saveMessage(@RequestBody String newMessage) {
       resourceServerProxy.postForLocation("http://localhost:9090", newMessage);
   }

   @Configuration
   public static class OauthClientConfiguration {

       @Bean
       @ConfigurationProperties("resourceServerClient")
       public ClientCredentialsResourceDetails getClientCredentialsResourceDetails() {
           return new ClientCredentialsResourceDetails();
       }

       @Bean
       public OAuth2RestTemplate restTemplate() {
           AccessTokenRequest atr = new DefaultAccessTokenRequest();
           return new OAuth2RestTemplate(getClientCredentialsResourceDetails(),
     new DefaultOAuth2ClientContext(atr));
       }
   }
}
```

Prepare an `application.yml` file with the following content:

```
server:
 port: 9999
logging:
 level:
   org.springframework.security: DEBUG

spring:
 aop:
   proxy-target-class: true

security:
 oauth2:
   resource:
     jwt:
       keyUri: http://localhost:8080/auth/oauth/token_key

resourceServerClient:
 accessTokenUri: http://localhost:8080/auth/oauth/token
 clientId: service-account-1
 clientSecret: service-account-1-secret
```

Add a simple `index.html` to have as a default page to see if the system is working:

```
<!DOCTYPE html>
<html>
   <head>
       <title>My application</title>
   </head>
   <body>
       <h1>Unsecured page</h1>

       <button onclick="loadMessage()">Load message</button>
       <div>Message is: <span id="message"></span></div>

       <div><button onclick="submitNewMessage()">Submit message</button></div>
       <input type="text" id="messageToSubmit">

       <script>
           function loadMessage() {
               fetch('/api/message')
                   .then(r => r.json())
                   .then(json => document.getElementById("message")
.textContent = json.message)
           }

           function submitNewMessage() {
               fetch('/api/message',
{method: 'POST', body:
 JSON.stringify(document.getElementById("messageToSubmit").value),
 credentials: 'same-origin',
 headers: new Headers(
   {'X-Requested-With':'XMLHttpRequest',
    'Content-Type': 'application/json'})
}
)
           }
       </script>
   </body>
</html>
```

To understand how the token expiration works, and its automatic renewal thanks to spring-oauth, let's set it to 60 seconds in OAuth2Config inside AS:
`.accessTokenValiditySeconds(60); // default is 43199 (12h)`

Time to test:

If you want to jump to the working example, just checkout the Step4-CallRSfromWebApp branch

```
> git checkout Step4-CallRSfromWebApp
```

Open a terminal and execute the following commands:

```
//launch Authorization Server on port 8080
> cd authorization-server;mvn spring-boot:run
```

Open another terminal and execute:

```
//launch Resource  Server on port 9090
> cd resource-server;mvn spring-boot:run
```

Open another terminal and execute:

```
//launch Client  Server on port 9999
> cd client-server;mvn spring-boot:run
```

Open your browser at http://localhost:9999 (http://localhost:9999) - Clear the log in the AS console - Click the “Load messages” button. - Hello world! appears in the page. - In the AS console you can see the call to obtain a token. - The RS correctly answers to the request (scope is correct).

In less than 60seconds from the first command: - Clear the log in the AS console. - Click the “Load messages” button. - AS console is still empty (no call to AS).

After more than 60seconds from the first command: - Clear the log in the AS console. - Click the “Load messages” button. - In the AS console you can see the call to obtain a token.

You can also write something in the input box and submit (scope _write works ok!). Then click Load message again and it will appear in the GUI.

Conclusions

Now it is your turn! Take one web application that calls some backend services using a service account.

Try to configure a call to the ResourceServer in our example. If your webapp is already using Spring, it is quite easy to find some examples of how to configure it and add ClientCredential headers to the RestTemplate.

Now you should expose one of your service as a Resource Server. If you have REST services developed with Spring, you just need to add @ResourceServer and configure the `security.oauth2.resource.jwt.keyUri` property to validate the token.

If you have SOAP web services that you want to protect via a JWT token, you have to configure the spring security filter in order to extract JWT and pass it to where your SOAP implementation expects. It is a little bit more work - but then you can copy it for future SOAP projects.

I hope I was able to give you a different perspective on OAuth2. I’ve tried an approach that is less fancy and showing advanced features, but that I hope could help during the introduction process in your company.

Any feedback, correction and opinion on this work is welcome, I will try to improve the article considering all of them.

Attributions

Many websites already describe OAuth2 in good depth, so I haven’t explained it here.

As a reference, I like the simplified explanation of Aaron Parecki (https://aaronparecki.com/2012/07/29/2/oauth2-simplified (https://aaronparecki.com/2012/07/29/2/oauth2-simplified)).

For a more deep dive in OAuth2 I highly recommend all the material produced by Dave Syer. This includes code, samples, articles, slides and videos. It is great stuff! As usual, Spring documentation on the topic is a great companion: http://projects.spring.io/spring-security-oauth/docs/oauth2.html (http://projects.spring.io/spring-security-oauth/docs/oauth2.html)

Last but not least I would like to thank my friend and colleague Sylvain (syjer almost everywhere, but @sy_jer (https://twitter.com/sy_jer) on twitter) for his help and visions while discussing all the OAuth2 scenarios and for his amazing technical skills always able to solve within minutes tricky bugs or errors that usually make me waste a lot of hours. It is easy to be fast and precise with such help. Thanks mate!

-
- security 1 (/categories.html#security-ref)
-
- authentication 1 (/tags.html#authentication-ref)
- ldap 1 (/tags.html#ldap-ref)
- oauth2 1 (/tags.html#oauth2-ref)
- security 2 (/tags.html#security-ref)
- spring 1 (/tags.html#spring-ref)

Recent Articles

OAuth2 in depth: A step-by-step introduction for enterprises (/security/2016/10/17/oauth2-in-depth-introduction-for-enterprises)

Spring Data Rest and JavaScript Client (/web/2016/08/30/spring-data-rest-hybind)

Gateleen - A RESTful Middleware (/api/2016/03/10/gateleen-restful-middleware)

Real-time Docker Stat Graphs in the Terminal (/docker/2016/01/28/realtime-docker-stat-graphs-terminal)

Contributing to Xwalk add mutual authentication support (/mobile/2015/12/17/contributing-to-opensource-project-xwalk)

Own Projects

jminix  (https://github.com/lbovet/jminix)

Contributions

- apikana (https://github.com/swisspush/apikana)
- vertx-rest-storage-editor (https://github.com/swisspush/vertx-rest-storage-editor)
- gateleen (https://github.com/swisspush/gateleen)
- jacoco-gwt-maven-plugin (https://github.com/swisspush/jacoco-gwt-maven-plugin)
- jdeferred (https://github.com/swisspush/jdeferred)
- mod-metrics (https://github.com/swisspush/mod-metrics)
- mod-redis (https://github.com/swisspush/mod-redis)
- platane (https://github.com/swisspush/platane)
- docson (https://github.com/swisspush/docson)
- vertx-cluster-watchdog (https://github.com/swisspush/vertx-cluster-watchdog)
- vertx-log-transformer (https://github.com/swisspush/vertx-log-transformer)
- vertx-redis-client (https://github.com/swisspush/vertx-redis-client)
- vertx-redisques (https://github.com/swisspush/vertx-redisques)
- vertx-rest-mirror (https://github.com/swisspush/vertx-rest-mirror)
- vertx-rest-storage (https://github.com/swisspush/vertx-rest-storage)
- vertx-rest-storage-docker (https://github.com/swisspush/vertx-rest-storage-docker)
- typson (https://github.com/swisspush/typson)

Blog

- Archive (/archive)
- Categories (/categories)
- Pages (/pages)
- Tags (/tags)