Java Reference Code for

NemLog-in STS Integration

with REST

(System User Scenario)

Status: Version 1.1

Version: 09.03.2018

[Changelog 3](#_Toc508369290)

[1 Introduction 4](#_Toc508369291)

[1.1 Intended audience 4](#_Toc508369292)

[1.2 Prerequisites 5](#_Toc508369293)

[1.3 Apache CXF and Spring Versions 5](#_Toc508369294)

[1.4 Disclaimer 5](#_Toc508369295)

[2 Building a REST-based Web Service Provider with Apache CXF and Spring 6](#_Toc508369296)

[2.1 Hello Service Endpoint 6](#_Toc508369297)

[2.2 Design Choices 6](#_Toc508369298)

[2.3 Security Requirements 6](#_Toc508369299)

[2.4 Registration with NemLog-in 7](#_Toc508369300)

[2.5 Implementation of the Authorization Service 9](#_Toc508369301)

[2.5.1 Shared code 9](#_Toc508369302)

[2.5.2 The Authorization Service code 10](#_Toc508369303)

[2.6 Implementation of the WSP 13](#_Toc508369304)

[2.7 Deployment and Testing 15](#_Toc508369305)

[3 Building a REST-based Web Service Consumer with Apache CXF and Spring 16](#_Toc508369306)

[3.1 Design Choices 16](#_Toc508369307)

[3.2 Security Requirements 16](#_Toc508369308)

[3.3 WSC Implementation Details 16](#_Toc508369309)

[3.4 Using the Client 19](#_Toc508369310)

[4 Example payloads 20](#_Toc508369311)

[5 Summary 21](#_Toc508369312)

[6 Typical Errors 22](#_Toc508369313)

[6.1 Java String Crypto Not Installed 22](#_Toc508369314)

[6.2 A .keystore File in Home Folder 22](#_Toc508369315)

[7 References 23](#_Toc508369316)

# Changelog

01.12.2015 Initial release

09.03.2018 Updated with reference to version 1.0 of OIO IDWS REST specification (no changes from DRAFT 3), and updated certificates in example code

# Introduction

This document is a companion to the Java reference source code for REST that showcase how to use the Apache CXF framework together with the Spring framework to implement

* A REST-based Web Service Provider (WSP) that requires clients to present a token issued by the NemLog-in STS, showing how to perform the following validations
  + Require the client to conform to the OIO IDWS REST Profile [OIO-IDWS-REST]
  + Implements an Authorization Service according to the OIO IDWS REST Profile
* A REST-based Web Service Consumer (WSC) that can
  + Interact with the NemLog-in STS to get a token issued
  + Call the Authorization Service on the above WSP with the issued token to get an Access Token
  + Call the REST service endpoint on the WSP with the Access Token according to the OIO IDWS REST Profile

Note that the NemLog-in STS supports a number of different usage scenarios that differ in how authentication to the STS is performed, including bootstrap token case, local token case and signature case illustrated below:



This reference code only covers the System User Scenario. Please refer to the NemLog-in documentation for descriptions of the other usage scenarios.

This document focuses primarily on the OIO IDWS REST Profile, and both the reference implementation and the documentation is based on the SOAP-based reference code, and the reader is recommended to read the documentation for this reference code for details on how the STS integration is implemented.

## Intended audience

This document is written for developers, and while all configuration and customization of Apache CXF and the Spring framework is dealt with, some experience with Apache CXF and Spring is recommended. The reader is also expected to have experience with Java development in general.

## Prerequisites

The source code uses Apache Maven 3 [MAVEN] as a build tool, and the source code requires at least Java 7 with Strong Crypto [CRYPTO] to compile and run. The reader is expected to have these tools available before using the reference source code. Maven handles all other dependencies.

## Apache CXF and Spring Versions

The reference code is based on Apache CXF 3.0.6 and Spring Boot 1.2.7. Spring Boot is a packaging of the Spring 4 framework, that comes with a sensible default configuration, that requires no further configuration or customization.

The Spring Framework was chosen as the REST framework as it has a very small code-footprint, and thus can easily be replaced by another REST framework if required.

## Disclaimer

The Danish Agency for Digitisation provides the reference code as is and assumes no responsibility for the code by service providers. Service Providers should understand the limitations of the code and deal with these according to their own needs.

# Building a REST-based Web Service Provider with Apache CXF and Spring

This chapter covers all the steps necessary to secure a Spring-based REST service (WSP) using Apache CXF. The reference source code is based on a very simple Hello service, and all parts of the code is clearly packaged and commented, so those sections relevant to dealing with security is easily identified.

Please note that this implementation integrates with the System User Scenario on the NemLog-in STS, which is holder-of-key based. The OIO IDWS REST Profile allowed for both holder-of-key and bearer tokens, but since the main difference between the two cases, is that the holder-of-key scenario requires client certificate validation, the bearer token version of this code is easily created by simply removing certain validation steps.

## Hello Service Endpoint

The Hello service has a single operation, that takes a text-string as input, and returns the corresponding “Hello [input]” text-string.

## Design Choices

The code uses the WSS4J security module from CXF for all security validation, but besides that uses no Apache CXF code. The WSS4J security module is also used internally by the CXF framework in the SOAP-based reference code, and this design choice ensures that the token validation performed in the REST reference code, will be identical to the validations performed in the SOAP reference code.

The choice of the REST framework for the REST service was taken to ensure the smallest code-footprint, allowing for easy substitution of the REST framework with another if this is required.

## Security Requirements

The service must be configured to enforce validation of the client request according to the OIO IDWS REST Profile.

This requires that the service is configured to require Client Authorization based on 2-way SSL. The Tomcat container used by the reference code contains the following configuration, which ensures that 2-way SSL is required for all endpoints

server.ssl.key-store=classpath:ssl-server.p12

server.ssl.key-store-password=Test1234

server.ssl.keyStoreType=PKCS12

server.ssl.keyAlias=1

server.ssl.client-auth=need

server.ssl.trust-store=classpath:ssl-trust.jks

server.ssl.trust-store-password=Test1234

server.ssl.trust-store-type=JKS

The configured keystore ssl-server.p12 contains a self-signed SSL certificate, which in a production environment should be replaced with a valid SSL certificate issued by a trusted CA.

The configured keystore ssl-trust.jks is a Tomcat specific requirement, as Tomcat will not allow a client to negotiate an SSL connection if it presents a client certificate that is not trusted by Tomcat. The keystore contains the OCES CA certificate from Nets test environment. In a production environment, using a reverse proxy like Apache Httpd, this is not required, as the Httpd server will handle the SSL negotiation, and Httpd, unlike Tomcat, can be configured to allow any client certificate to establish the SSL connection.

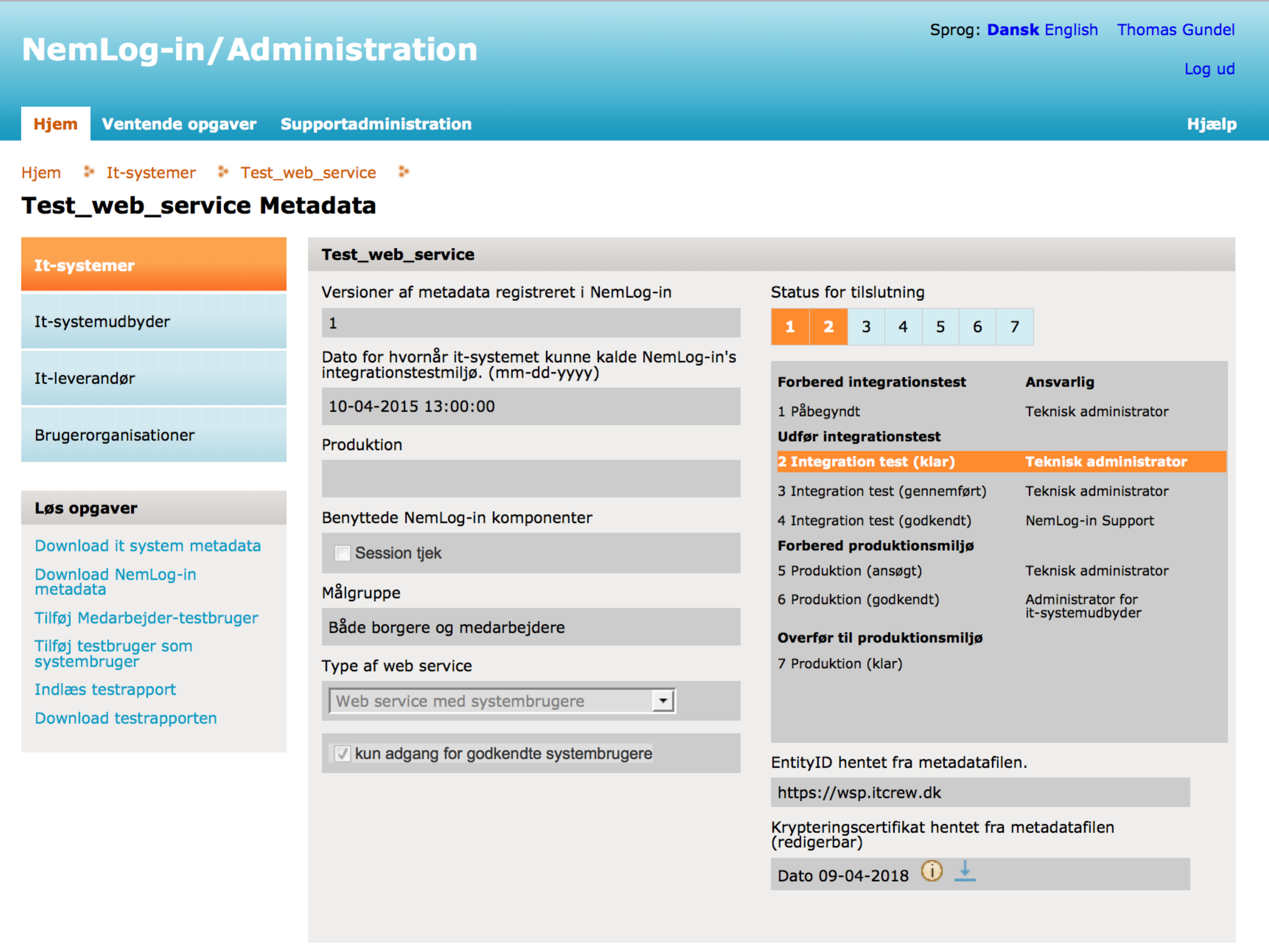
Finally the WSP must be configured to trust the STS. The STS certificate is stored in the sts-trust.jks keystore, and contains the STS certificate from the NemLog-in test environment. This trust store should be replaced with one containing the production STS certificate when used in the production environment.

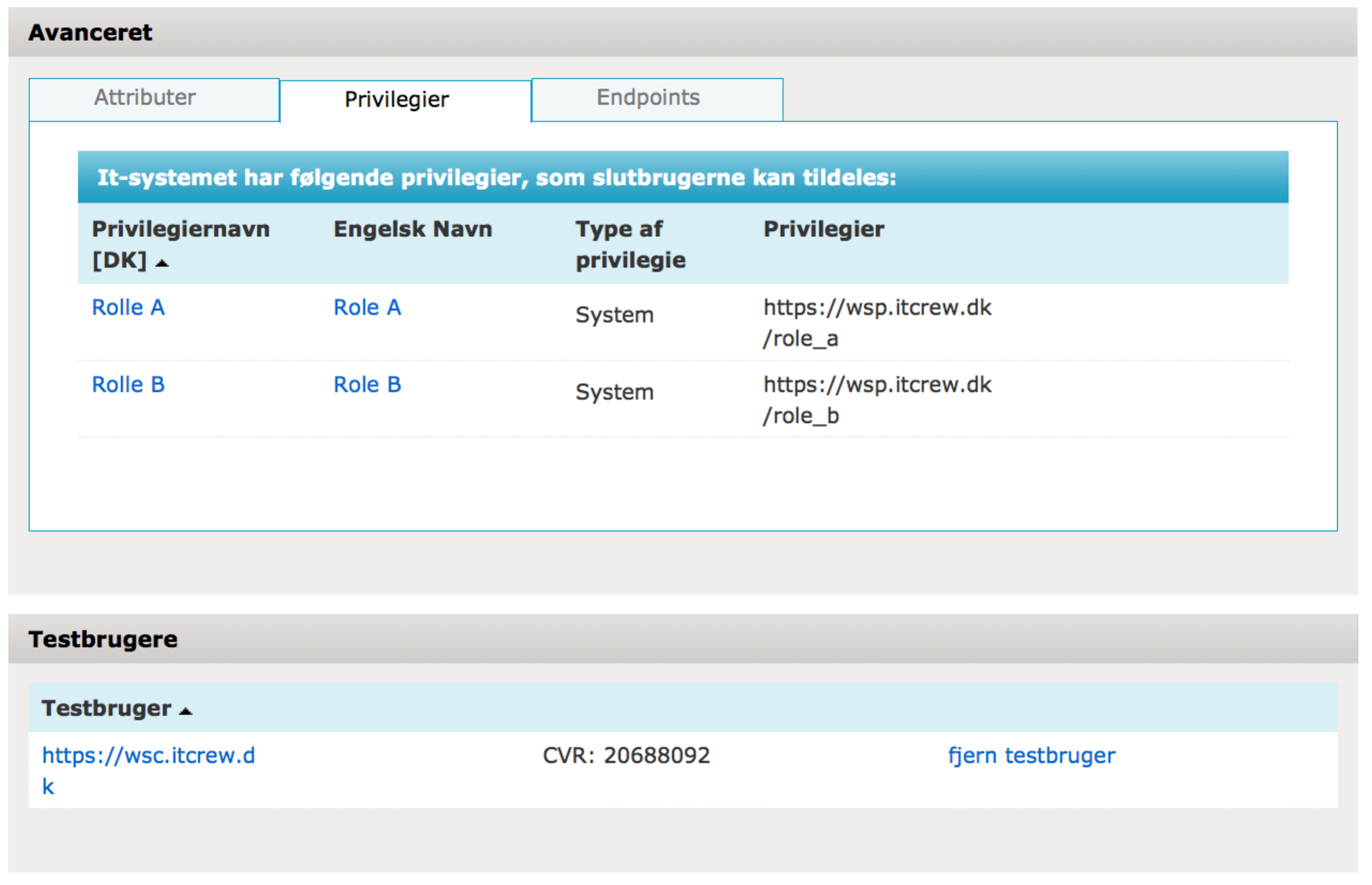
## Registration with NemLog-in

The example WSC and WSP have been registered with NemLog-in[[1]](#footnote-2) to make the example work:

* The client (WSC) is registered as a system user with EntityID “https://wsc.itcrew.dk”.
* The WSP is registered as a web service with the EntityID “<https://wsp.itcrew.dk>” with two privileges (roles). The client has been granted these two privileges by the administrator of the WSP such they will appear in tokens issued by the STS to the WSP.

The process of registering with NemLog-in is not covered in this guide, but a detailed process description with associated screenshots will be provided on Digitaliser.dk in the group[[2]](#footnote-3): A screen shot showing the registration details for the WSP is shown below:





## Implementation of the Authorization Service

The OIO IDWS REST Profile allows for the Authorization Service to either be an integrated part of the WSP, or a stand-alone component. The reference code uses the integrated approach, as this allows for the simplest implementation.

All the code for the Authorization Service is inside the package ***service.auth***, and can easily be extracted into a separate component. If this approach is chosen, the current memory-based AccessTokenCache implementation must be exchanged with some persistent storage implementation, for instance an SQL storage mechanism.

### Shared code

The Authorization Service shares a bit of code with the WSP.

It shares a helper class called CertificateHelper, which has a single method for extracting the clients SSL certificate from the request, which is used by both the WSP and Authorization Service as part of the token validation.

The code shown below finds the first non-CA certificate presented by the client, and returns that to either the WSP or Authorization Service, depending on who needs it.

@Component

public class CertificateHelper **{**

public X509Certificate extractCertificate**(**HttpServletRequest request**)** **{**

X509Certificate**[]** certificates **=** **(**X509Certificate**[])**

request**.**getAttribute**(**"javax.servlet.request.X509Certificate"**);**

**if** **(**certificates **!=** **null** **&&** certificates**.**length **>** 0**)** **{**

**for** **(**X509Certificate certificate **:** certificates**)** **{**

**if** **(**certificate**.**getBasicConstraints**()** **>** 0**)** **{**

**continue;**

**}**

**return** certificate**;**

**}**

**}**

**return** **null;**

**}**

**}**

It also shares the configuration class (SecurityConfig.java), which is a standard Java configuration class. The Authorization Service uses most of the configured elements, and those are shown in the code excerpt below

@Configuration

public class SecurityConfig **{**

@Value**(**"classpath:sts-trust.jks"**)**

private Resource keystore**;**

@Autowired

private CertificateHelper certificateHelper**;**

@Bean**(**name **=** "accessTokenCache"**)**

public Map**<**String**,** SamlAssertionWrapper**>** expiringMap**()** **{**

**return** ExpiringMap**.**builder**().**expiration**(**60**,** TimeUnit**.**MINUTES**).**build**();**

**}**

@Bean**(**name **=** "cert"**)**

public X509Certificate cert**() {**

KeyStore ks **=** KeyStore**.**getInstance**(**"JKS"**);**

ks**.**load**(**keystore**.**getInputStream**(),** "Test1234"**.**toCharArray**());**

**return** **(**X509Certificate**)** ks**.**getCertificate**(**ks**.**aliases**()**

**.**nextElement**());**

**}**

@Bean

public KeyStore keyStore**() {**

KeyStore ks **=** KeyStore**.**getInstance**(**"JKS"**);**

ks**.**load**(new** FileInputStream**(**"src/main/resources/service.jks"**),**

"Test1234"**.**toCharArray**());**

**return** ks**;**

**}**

**}**

Note that the configuration class contains an ExpiringMap instance, which is the caching mechanism used by the Authorization Service to keep track of issued AccessTokens.

The cache is implemented using the ExpiringMap class, which is available as an Apache licensed module, distributed as a Maven artefact.

Note that the cache is configured to expire AccessTokens after 60 minutes. This is the maximum allowed validity period for an AccessToken according to the OIO IDWS REST Profile.

### The Authorization Service code

The Authorization Service is implemented as two classes, the first (AuthorizationService.java) is the actual Authorization Service, and the second (SamlDecoder.java) is a helper class, capable of decoding and decrypting the incoming token.

If we look at the SamlDecoder class first, it has the following structure

@Component

public class SamlDecoder **{**

@Autowired

private KeyStore keyStore**;**

public SamlAssertionWrapper decodeAsertion**(**String samlToken**)** **{**

**...**

**}**

private static Element decodeToken**(**String token**)** **{**

**...**

**}**

private static Document decryptElementDOM**(**Document doc**,** Key rsaKey**)** **{**

**...**

**}**

**}**

The class is annotated as a @Component, which allows the Spring framework to inject the KeyStore into the class. The KeyStore contains the WSP’s private key, allowing the Authorization Service to decrypt the incoming STS issued SAML token.

The public method decodeAssertion first calls the decodeToken class to Base64 decode the token, and the construct an EncryptedData element from the resulting data. Secondly it calls the decryptElementDOM method to decrypt the EncryptedData element, resulting in an Assertion element, which is wrapped in the wss4j helper class SamlAssertionWrapper.

The code content of the three methods is basic XML parsing using the Apache xmlsec library, and is left out of the documentation for brevity.

The code for the Authorization Service is found in the AuthorizationService.java class, and it is shown in completeness below. Note that the class is annotated as a @RestController, which tells the Spring framework that this class exposes a REST service, which must be made available to callers.

The method authService() is bound to /auth by the @RequestMapping annotation, and together with the @RestController annotation, this is all the configuration that Spring requires to expose this method as a REST service endpoint.

The authService() method uses the above SamlDecoder class to decode and decrypt the input, and then it calls the validateAssertion method within this class, to perform the full set of validations required. Note that it uses the wss4j SamlAssertionWrapper class for this purpose, which is the same class that Apache CXF uses to perform validation of SAML Assertions.

Note that a single additional validation has been added, which compares the client certificate used by the WSC to establish the SSL connection, with the certificate found in the SubjectConfirmation element found in the SAML Assertion. This validation is only needed for holder-of-key tokens, and if this reference code is used in Bearer token scenarios, this validation should be removed.

Finally, the Authorization Service issued an AccessToken, based on the UUID class, which is a 122-bit value. Note that the exact amount of entropy within the UUID value depends on the quality of the RNG used by Java. By default, it uses the SecureRandom class, which is a pretty decent CRNG. Finally, the AccessToken is stored in the AccessTokenCache together with the SamlAssertionWrapper instance. This allows the WSP to later access token SAML token when presented with an AccessToken.

@RestController

public class AuthorizationService **{**

private static final Logger logger **=**

Logger**.**getLogger**(**AuthorizationService**.**class**);**

private static final String allowedAudience **=** "https://wsp.itcrew.dk"**;**

@Resource**(**name **=** "accessTokenCache"**)**

private Map**<**String**,** SamlAssertionWrapper**>** accessTokenCache**;**

@Autowired

private SamlDecoder samlDecoder**;**

@Autowired

private CertificateHelper certificateHelper**;**

@Autowired

@Qualifier**(**"cert"**)**

private X509Certificate stsCertificate**;**

@PostConstruct

void init**()** **{**

org**.**apache**.**xml**.**security**.**Init**.**init**();**

**}**

@RequestMapping**(**value **=** "/auth"**,** method **=** RequestMethod**.**POST**,**

consumes **=** "application/x-www-form-urlencoded"**)**

public ResponseEntity**<**AccessToken**>** authService**(**

@RequestParam**(**"saml-token"**)** String samlToken**,**

HttpServletRequest request**,** HttpServletResponse response**)** **{**

SamlAssertionWrapper samlAssertionWrapper **=** **null;**

**try** **{**

samlAssertionWrapper **=** samlDecoder**.**decodeAsertion**(**samlToken**);**

X509Certificate clientCertificate **=**

certificateHelper**.**extractCertificate**(**request**);**

validateAssertion**(**samlAssertionWrapper**,** clientCertificate**);**

**}**

**catch** **(**Exception ex**)** **{**

logger**.**error**(**"Failed to validate token"**,** ex**);**

**try** **{**

response**.**setHeader**(**"WWW-Authenticate"**,**

"error=\"invalid\_token\", error\_description=\"" **+**

ex**.**getMessage**()** **+** "\""**);**

response**.**sendError**(**HttpStatus**.**UNAUTHORIZED**.**value**());**

**}**

**catch** **(**IOException e**)** **{**

logger**.**error**(**"Failed to send error to client"**,** e**);**

**}**

**}**

AccessToken accessToken **=** **new** AccessToken**();**

accessToken**.**setToken**(**UUID**.**randomUUID**().**toString**());**

accessToken**.**setTokenType**(**"holder-of-key"**);**

accessToken**.**setExpiresIn**(**3600**);**

accessTokenCache**.**put**(**accessToken**.**getToken**(),** samlAssertionWrapper**);**

HttpHeaders responseHeaders **=** **new** HttpHeaders**();**

responseHeaders**.**add**(**"Pragma"**,** "no-cache"**);**

responseHeaders**.**add**(**"Cache-Control"**,** "no-store"**);**

responseHeaders**.**add**(**"Content-Type"**,** "application/json"**);**

**return** **new** ResponseEntity**<>(**accessToken**,**

responseHeaders**,** HttpStatus**.**OK**);**

**}**

private void validateAssertion**(**SamlAssertionWrapper samlAssertionWrapper**,**

X509Certificate clientCertificate**)** **{**

// validate that the token is well-formed

samlAssertionWrapper**.**validateAssertion**(true);**

// validate that the token has not expired (defaults to 8 hours

// time-to-live if no expire timestamp is present in the token)

samlAssertionWrapper**.**checkConditions**(**8 **\*** 60 **\*** 60**);**

// validate that the token has been signed by an STS that we trust

List**<**X509Certificate**>** certList **=** **new** ArrayList**<>();**

certList**.**add**(**stsCertificate**);**

SAMLKeyInfo samlKeyInfo **=** **new** SAMLKeyInfo**(**certList**.**toArray**(**

**new** X509Certificate**[**0**]));**

samlAssertionWrapper**.**verifySignature**(**samlKeyInfo**);**

// parse the token to make the SubjectKeyInfo element available

// (samlAssertionWrapper.getSubjectKeyInfo() will return null

// otherwise)

WSDocInfo docInfo **=** **new** WSDocInfo**(**

samlAssertionWrapper**.**getElement**().**getOwnerDocument**());**

RequestData data **=** **new** RequestData**();**

data**.**setWssConfig**(**WSSConfig**.**getNewInstance**());**

samlAssertionWrapper**.**parseSubject**(**

**new** WSSSAMLKeyInfoProcessor**(**data**,** docInfo**),** **null,** **null);**

// extract the client-certificate from the SSL connection and the

// KeyInfo element from the presented token

SAMLKeyInfo subjectKeyInfo **=** samlAssertionWrapper**.**getSubjectKeyInfo**();**

// since this is a holder-of-key token, we compare the public key from

// the certificate found in the SujectConfirmation element

// with the public key of the certificate used by the client for

// negotiating 2-way SSL

boolean match **=** Arrays**.**equals**(**

clientCertificate**.**getPublicKey**().**getEncoded**(),**

subjectKeyInfo**.**getCerts**()[**0**].**getPublicKey**().**getEncoded**());**

**if** **(!**match**)** **{**

**throw** **new** Exception**(**"ssl cert does not match token certificate"**);**

**}**

// validate that this token is indeed intended for this

// service (and not some other service)

List**<**AudienceRestriction**>** audienceRestrictions **=**

samlAssertionWrapper**.**getSaml2**()**

**.**getConditions**()**

**.**getAudienceRestrictions**();**

boolean found **=** **false;**

**for** **(**AudienceRestriction audienceRestriction **:** audienceRestrictions**)** **{**

**for** **(**Audience audience **:** audienceRestriction**.**getAudiences**())** **{**

**if** **(**allowedAudience**.**equals**(**audience**.**getAudienceURI**()))** **{**

found **=** **true;**

**}**

**}**

**}**

**if** **(!**found**)** **{**

**throw** **new** Exception**(**"token audience does not match '" **+**

allowedAudience **+** "'"**);**

**}**

**}**

**}**

## Implementation of the WSP

The reference code for the WSP is located in the same source project as the Authorization Service, the two main packages are ***service.rest*** and ***service.security***. These two packages contains the following classes

├── rest

│   └── HelloWorldService.java

└── security

└── AccessTokenFilter.java

The HelloWorldService class contains only the very simple Hello REST service, which in its completeness looks like this

@RestController

public class HelloWorldService **{**

@RequestMapping**(**"/api/hello"**)**

public String greeting**(**@RequestParam**(**value **=** "name"**)** String name**)** **{**

**return** "Hello " **+** name**;**

**}**

**}**

Note that the REST Service does not deal with security, this is handled by the AccessTokenFilter, which is configured to run as a standard Java Servlet Filter in front of the REST Service endpoint.

The AccessTokenFilter class uses the AccessTokenCache and the CertificateHelper classes to perform the validation of the request. The code looks like this, and basically validates that a token exists in the AccessTokenCache that matches the presented token, and then validates that the presented client SSL certificate matches the one found in the stored Assertion.

public class AccessTokenFilter **implements** Filter **{**

private static final Logger logger **=**

Logger**.**getLogger**(**AccessTokenFilter**.**class**);**

private Map**<**String**,** SamlAssertionWrapper**>** accessTokenCache**;**

private CertificateHelper certificateHelper**;**

public AccessTokenFilter**(**Map**<**String**,** SamlAssertionWrapper**>**

accessTokenCache**,** CertificateHelper certificateHelper**)** **{**

**this.**accessTokenCache **=** accessTokenCache**;**

**this.**certificateHelper **=** certificateHelper**;**

**}**

@Override

public void doFilter**(**ServletRequest request**,** ServletResponse response**,**

FilterChain chain**)** **{**

HttpServletRequest httpRequest **=** **(**HttpServletRequest**)** request**;**

HttpServletResponse httpResponse **=** **(**HttpServletResponse**)** response**;**

**try** **{**

String authorizationHeader **=** httpRequest

**.**getHeader**(**"Authorization"**);**

**if** **(**authorizationHeader **==** **null)** **{**

**throw** **new** Exception**(**"Missing Authorization header"**);**

**}**

**if** **(!**authorizationHeader**.**startsWith**(**"Holder-of-key "**))** **{**

**throw** **new** Exception**(**

"Authorization header does not contain a valid Holder-of-key access token"**);**

**}**

String accessToken **=** authorizationHeader**.**substring**(**

"Holder-of-key "**.**length**());**

**if** **(!**accessTokenCache**.**containsKey**(**accessToken**))** **{**

**throw** **new** Exception**(**"Access token is invalid or expired"**);**

**}**

X509Certificate clientCertificate **=** certificateHelper

**.**extractCertificate**(**httpRequest**);**

SAMLKeyInfo subjectKeyInfo **=** accessTokenCache**.**get**(**accessToken**)**

**.**getSubjectKeyInfo**();**

boolean match **=** Arrays**.**equals**(**

clientCertificate**.**getPublicKey**().**getEncoded**(),**

subjectKeyInfo**.**getCerts**()[**0**].**getPublicKey**().**getEncoded**());**

**if** **(!**match**)** **{**

**throw** **new** Exception**(**"ssl cert does not match token cert"**);**

**}**

chain**.**doFilter**(**httpRequest**,** httpResponse**);**

**}**

**catch** **(**Exception ex**)** **{**

logger**.**error**(**"Failed to validate access token"**,** ex**);**

httpResponse**.**setHeader**(**"WWW-Authenticate"**,**

"error=\"invalid\_token\",

error\_description=\"" **+**

ex**.**getMessage**()** **+** "\""**);**

httpResponse**.**sendError**(**HttpStatus**.**UNAUTHORIZED**.**value**());**

**}**

**}**

**}**

## Deployment and Testing

To build and deploy the service, perform the following commands from the command-line. Make sure to execute the command from the rest-service folder (where the pom.xml file for the rest-service project is located)

$ mvn clean install

Once the project has been compiled, start the service with the following command

$ mvn spring-boot:run

The command uses the Spring Boot plugin for maven to start a Tomcat instance and run the application inside the Tomcat container. The service will deploy two endpoints

<https://localhost:8443/auth>

<https://localhost:8443/api/hello>

Both endpoints are protected, the first requiring an STS-issued token to access, and the second an AccessToken issued by the first endpoint.

Testing that the service works will require a REST-based web service consumer that implements the OIO IDWS REST Profile. The following chapter covers how to build such a service.

# Building a REST-based Web Service Consumer with Apache CXF and Spring

This chapter covers all the steps necessary to build a REST-based Web Service Consumer (client) capable of calling a REST-based web service provider secured as detailed in the previous chapter. This includes calling the STS and getting a token, exchanging this token for an Access Token at the Authorization Service, and using the Access Token to call the actual REST service endpoint.

## Design Choices

As with the service in the previous chapter, the reference code uses Apache CXF for the security implementation and Spring for the REST client implementation. The use of Apache CXF ensures that the client acts in the same way as the SOAP reference code, and the Spring REST Client code is easily replaced by another framework.

## Security Requirements

The client must be configured to follow the security requirements of both the STS and the service it wants to call. It must also ensure that it follows the requirements detailed by the OIO IDWS REST Profile.

The Apache CXF framework deals with accessing the STS, and the code found in this project is a direct copy of the same code found in the SOAP-based reference code. For that reason, this documentation only touches on the STS integration very lightly, and primarily on the way that the REST code uses the code from the SOAP-based project.

As the OIO IDWS REST Profile requires that the WSC uses 2-way SSL when connecting to the WSP, and that the same client certificate that is used against the STS is used against the WSP, the reference code configured the Apache HttpClient to use the same keystore for SSL as the code uses for accessing the STS.

The project must be configured to trust the SSL certificate used by the WSP, as well as the certificate used by the STS to issue tokens. These certificates are located in the trust.jks keystore found in the project.

Finally the STS must be aware of the client, so it can issue token to the client. This is outside the scope of this document, and the reference source for this client has already been registered with the STS under the following EntityId: <https://wsc.itcrew.dk>

## WSC Implementation Details

The WSC reuses parts of the code from the SOAP reference code. The classes and code reused are the following

DigstSTSClient.java

STSAdressingInterceptor

ClientCallbackHandler

cxf.xml

sts.wsdl

These files have been copied verbatim from the SOAP reference code, and are not documented here. They are all used for calling the STS and the SOAP reference code contains the full documentation for these classes.

The class TokenFetcher found in this reference code wraps the above classes, and exposes a single public method called getAccessToken, which is responsible for getting an AccessToken from the Authorization Service found in the WSP project.

As a side-effect of getting an AccessToken, the TokenFetcher must get a token from the STS, as shown in the code below

@Component

public class TokenFetcher **{**

private ExpiringMap**<**String**,** AccessToken**>** accessTokenCache **=**

ExpiringMap**.**builder**().**expiration**(**55**,** TimeUnit**.**MINUTES**).**build**();**

private ExpiringMap**<**String**,** SecurityToken**>** samlTokenCache **=**

ExpiringMap**.**builder**().**expiration**(**7**,** TimeUnit**.**HOURS**).**build**();**

@Autowired

private STSClient stsClient**;**

@Autowired

private RestTemplate restTemplate**;**

public AccessToken getAccessToken**(**String audience**)** **throws** Exception **{**

AccessToken accessToken **=** accessTokenCache**.**get**(**audience**);**

**if** **(**accessToken **==** **null)** **{**

SecurityToken samlToken **=** samlTokenCache**.**get**(**audience**);**

**if** **(**samlToken **==** **null)** **{**

samlToken **=** stsClient**.**requestSecurityToken**(**audience**);**

samlTokenCache**.**put**(**audience**,** samlToken**);**

**}**

String encodedToken **=** "saml-token=" **+**

TokenEncoder**.**encode**(**samlToken**);**

HttpHeaders headers **=** **new** HttpHeaders**();**

headers**.**add**(**"Content-Type"**,**

"application/x-www-form-urlencoded;charset=UTF-8"**);**

ResponseEntity**<**AccessToken**>** authorizationServiceResponse **=**

restTemplate**.**exchange**(**"https://localhost:8443/auth"**,**

HttpMethod**.**POST**,**

**new** HttpEntity**<>(**encodedToken**,** headers**),** AccessToken**.**class**);**

accessToken **=** authorizationServiceResponse**.**getBody**();**

accessTokenCache**.**put**(**audience**,** accessToken**);**

**}**

**return** accessToken**;**

**}**

**}**

Note that like the WSP, the WSC uses the ExpiringMap class to great effect, caching both the tokens it gets from the STS as well as the AccessTokens from the AuthorizationService. The logic behind the getAccessToken method ensures that both the STS and AuthorizationService is only called when no valid token exists in the cache.

The TokenEncoder class used by the above code is just a simple helper class, that will convert a token to a base64/url-encoded string.

As all requests towards both the Authorization Service and the WSP requires that the WSC presents a client certificate when negotiating the SSL connection, the following class sets up the correct configuration of the Apache HttpClient component used for performing REST class.

@Configuration

public class ClientConfig **{**

@Value**(**"classpath:client.jks"**)**

private Resource keystore**;**

@Bean

public ClientHttpRequestFactory httpComponentsClientHttpRequestFactory**() {**

SSLContextBuilder builder **=** **new** SSLContextBuilder**();**

builder**.**loadTrustMaterial**(null,** **new** TrustSelfSignedStrategy**());**

InputStream inputStream **=** keystore**.**getInputStream**();**

KeyStore ks **=** KeyStore**.**getInstance**(**"JKS"**);**

ks**.**load**(**inputStream**,** "Test1234"**.**toCharArray**());**

builder**.**loadKeyMaterial**(**ks**,** "Test1234"**.**toCharArray**());**

SSLConnectionSocketFactory sslsf **=**

**new** SSLConnectionSocketFactory**(**builder**.**build**());**

CloseableHttpClient httpClient **=** HttpClients**.**custom**()**

**.**setSSLHostnameVerifier**(new** NoopHostnameVerifier**())**

**.**setSSLSocketFactory**(**sslsf**).**build**();**

HttpComponentsClientHttpRequestFactory requestFactory **=**

**new** HttpComponentsClientHttpRequestFactory**();**

requestFactory**.**setHttpClient**(**httpClient**);**

**return** **new** BufferingClientHttpRequestFactory**(**requestFactory**);**

**}**

@Bean

public RestTemplate restTemplate**()** **throws** Exception**{**

**return** **new** RestTemplate**(**httpComponentsClientHttpRequestFactory**());**

**}**

**}**

The configured RestTemplate acts as the REST client, and it is configured to use the configured httpClient factory to construct http-requests, which in effect will ensure that the configured keystore is used for establishing the 2-way SSL connection.

The configured RestTemplate class, together with the TokenFetcher is used by the applications main class like this

@ComponentScan

@EnableAutoConfiguration

public class Application **implements** CommandLineRunner **{**

private static final Logger logger **=** Logger**.**getLogger**(**Application**.**class**);**

@Autowired

private TokenFetcher tokenFetcher**;**

@Autowired

private RestTemplate restTemplate**;**

public static void main**(**String**[]** args**)** **{**

SpringApplication**.**run**(**Application**.**class**,** args**);**

**}**

public void run**(**String**...** args**)** **throws** Exception **{**

// get the access token

AccessToken accessToken **=** tokenFetcher**.**getAccessToken**(**

"https://wsp.itcrew.dk"**);**

// setup request Authorization header

HttpHeaders headers **=** **new** HttpHeaders**();**

headers**.**add**(**"Authorization"**,**

"Holder-of-key " **+** accessToken**.**getToken**());**

// call service

ResponseEntity**<**String**>** restServicResponse **=** restTemplate**.**exchange**(**

"https://localhost:8443/api/hello?name=John"**,**

HttpMethod**.**GET**,**

**new** HttpEntity**<>(**""**,** headers**),** String**.**class**);**

// should print out "Hello John"

logger**.**info**(**restServicResponse**.**toString**());**

**}**

**}**

## Using the Client

The reference code for the client consists of a single self-contained project, which is located in the folder “rest-client”. The project can be compiled using Maven with the following command. Make sure that the command is issued from the directory that contains the pom.xml for this project

$ mvn clean install

This will compile the client project as a command-line application. The client can be executed by using the following Maven command

$ mvn spring-boot:run

This will execute the client, the implementation being located in the Application.java class.

The class first gets an AccessToken (which is cached for later use), and then uses the AccessToken to call the actual REST service.

It is assumed that the rest-service from the previous chapter is running.

Also note that tracing is enabled on the client project, so the console will contain the full request and response payloads towards both the service and the STS.

# Example payloads

The reference code has trace logging enabled on the client, so by running the reference code, it is possibly to recreate example payloads mentioned in this chapter.

Full traces can be found in the “traces” folder inside the “doc” folder.

The following files are available for inspection

├── 1-WSC-TO-STS.XML

├── 2-STS-TO-WSC.XML

├── 3-WSC-TO-AS.TXT

├── 4-AS-TO-WSC.TXT

├── 5-WSC-TO-WSP.TXT

└── 6-WSP-TO-WSC.TXT

# Summary

This document has covered the steps needed to implement both an Authorization Service as well as a WSP that follows the OIO IDWS REST Profile, and a client that can use both the Authorization Service and the WSP REST service.

The document is bundled with reference code that has implemented these steps on a very simple web service, and the reference code can potentially be used as a template when creating a new web service from scratch, or simply as inspiration when modifying an existing web service.

# Typical Errors

## Java String Crypto Not Installed

Getting an exception of the following type is usually a strong indication that the Unlimited Strength Jurisdiction Policy Files [CRYPTO] is not installed correctly

java.security.InvalidKeyException: Illegal key size

## A .keystore File in Home Folder

When running the reference code from the command-line (or Eclipse), the following exception might be thrown

java.security.UnrecoverableKeyException: Password must not be null

This is because Apache CXF will look for a .keystore file (note the . in the beginning of the filename) in the executing users home folder when setting up the SSL connection. To get rid of the error, remove the keystore from the home folder.

# References

[MAVEN] Apache Maven Build Tool v 3.x

<https://maven.apache.org/download.cgi>

[CRYPTO] Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files

**Java 7**

<http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html>

**Java 8** <http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html>

[OIO-IDWS-REST] Version 1.0 of the specification is located in the Documentation folder of the reference code distribution.

1. https://administration.nemlog-in.dk [↑](#footnote-ref-2)
2. https://digitaliser.dk/group/2848479 [↑](#footnote-ref-3)