OIO Service Oriented Infrastructure

RASP Library for Java Version 2.0.0

Tutorials

Contents

1 Introduction

The OIOSI RASP Library for Java is a java based toolkit for implementation of RASP business applications.

This distribution is Version 2.0.0

The distribution is part of the OIOSI work for exchanging business documents in a secure and reliable way using the internet. See http://www.digst.dk/Loesninger-og-infrastruktur/NemHandel/For-it-udviklere for more information.

The framework can be downloaded from http://digitaliser.dk/group/405442/resources/type/150019.

The intended audience is developers who want to integrate the RASP framework into their own application.

Before reading this document, please take the time to read the following documents:

- OIOSI RASP Library for Java Release Notes.doc
- OIOSI RASP Library for Java Source - Installation Guide.doc

2 Prerequisites

These tutorials suppose you are working with Eclipse, Tomcat 6 or later and are running Ant from within Eclipse.

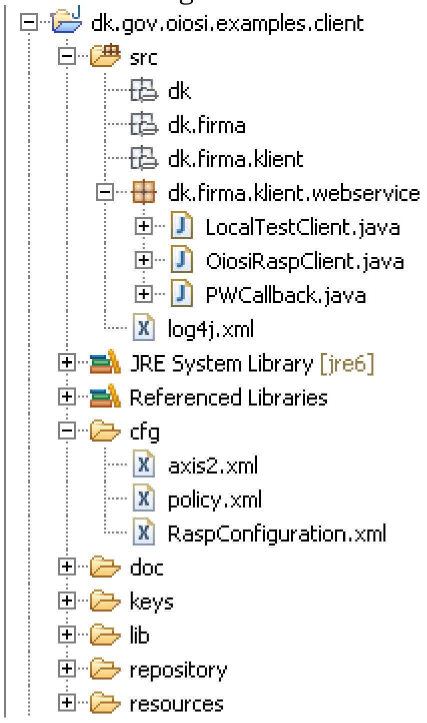
Before starting, make sure you have gone through the installation document for the RASP library.

3 Lesson: Creating a RASP client and service

This tutorial assumes that the examples projects have been imported into Eclipse.

3.1 The example clients

The example client project has the following structure:



```
dk.gov.oiosi.examples.client
  src
    dk
    dk.firma
    dk.firma.klient
    dk.firma.klient.webservice
      LocalTestClient.java
      OiosiRaspClient.java
      PWCallback.java
    log4j.xml
  JRE System Library [jre6]
  Referenced Libraries
  cfg
    axis2.xml
    policy.xml
    RaspConfiguration.xml
  doc
  keys
  lib
  repository
  resources
```

The source folder of the client examples contains two applications, the Local test client, and a full OIOSI RASP client.

Our local test client sends a UBL document to a service deployed locally by simply giving the RASP framework the correct address and server side certificate to use.

The full RASP client however, fetches a test document, finds the endpoint reference within it, looks it up in the UDDI registry, downloads the server side certificate and then sends the document according to the data just found. By default the test document has an EAN that is registered to a RASP test server run by Digitaliseringsstyrelsen.

The full client runs through the following steps

1. Load a document
2. Validate the document
3. Wrap the document in a SOAP message and add the custom RASP headers
4. Make an UDDI lookup to find the service address
5. Use information from the UDDI response to download the server certificate via LDAP
6. Validate the certificate downloaded against an OCSP server
7. Sends it to a service.

All of these steps are explained in further detail as comments within the code.

It is strongly recommended that you read these comments before implementing your own

RASP client application, since they describe what SOAP headers to add, and what validation to make to ensure a truly secure communication.

3.1.1 Mail transport configuration

The mail transport is configured using the following two files:

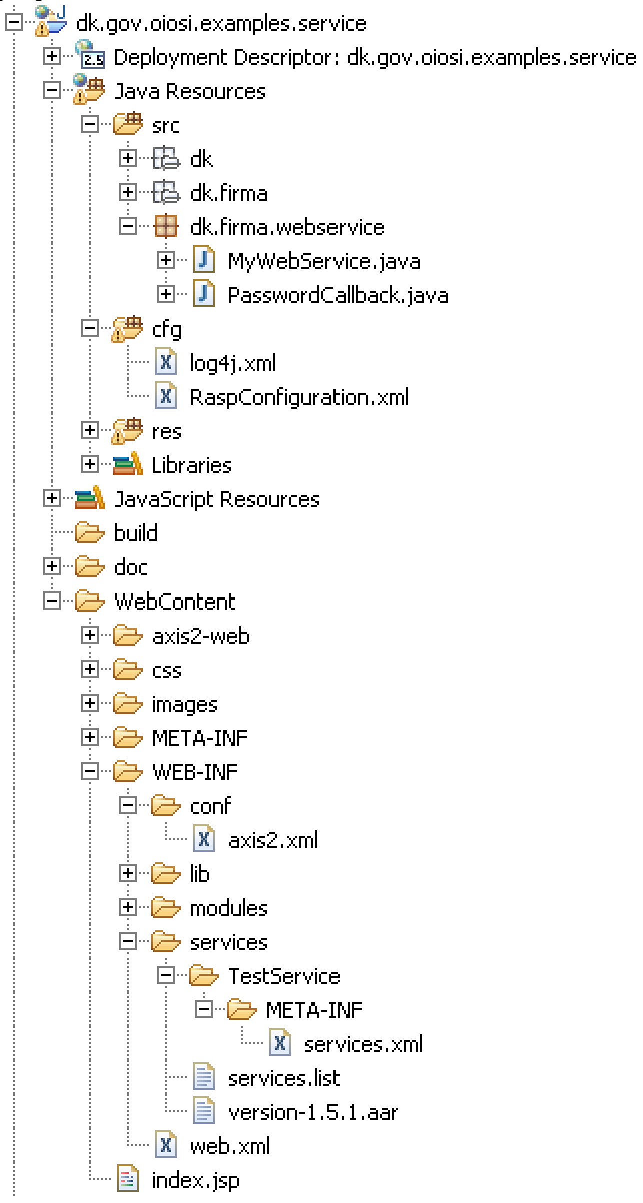
- cfg/axis2.xml
- cfg/RaspConfiguration.xml

axis2.xml contains the configuration of the SMTP server to use. This configuration can be found in the “Mail Transport Sender” configuration segment.

RaspConfiguration.xml contains the configuration of POP3 server to use. This configuration is found in the “EmailTransportUserConfig” configuration section.

3.2 The service example

The example service project structure is shown below:



```
dk.gov.oiosi.examples.service
    Deployment Descriptor: dk.gov.oiosi.examples.service
    Java Resources
        src
            dk
            dk.firma
            dk.firma.webservice
                MyWebService.java
                PasswordCallback.java
        cfg
            log4j.xml
            RaspConfiguration.xml
        res
        Libraries
    JavaScript Resources
    build
    doc
    WebContent
        axis2-web
        css
        images
        META-INF
        WEB-INF
            conf
                axis2.xml
            lib
            modules
            services
                TestService
                    META-INF
                        services.xml
                services.list
                version-1.5.1.aar
            web.xml
        index.jsp
```

The service example is set up as a simple web service, which validates the incoming document and replies with an empty message.

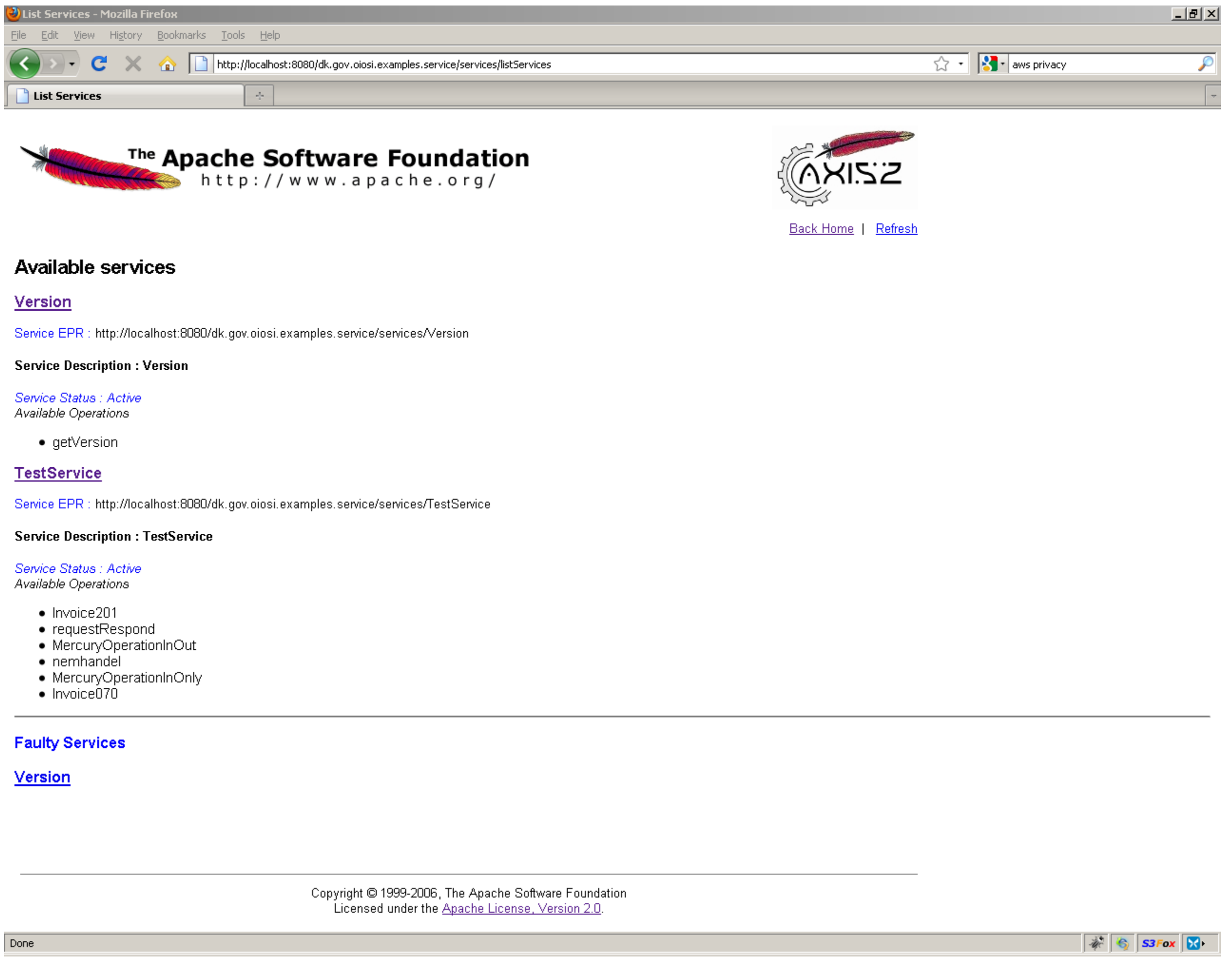
The service can by run directly from eclipse by choosing “Run On Server”, or it can by exported as an WAR file. The WAR file can be deployed on a Tomcat server by copying it into the Tomcat home/webapps folder, and it should automatically be deployed.

To test the service out, enter the following address into your web browser

http://localhost:8080/dk.gov.oiosi.examples.service/

(assuming you have set up Tomcat to run on port 8080, and the webapp name is dk.gov.oiosi.examples.service).

On this page one can click "List Axis services" and a presentation of the operations offered by the service will be shown:



List Services - Mozilla Firefox

http://localhost:8080/dk.gov.oiosi.examples.service/services/listServices

The Apache Software Foundation http://www.apache.org/

Back Home | Refresh

Available services

Version

Service EPR : http://localhost:8080/dk.gov.oiosi.examples.service/services/Version

Service Description : Version

Service Status : Active
Available Operations

- getVersion

TestService

Service EPR : http://localhost:8080/dk.gov.oiosi.examples.service/services/TestService

Service Description : TestService

Service Status : Active
Available Operations

- Invoice201
- requestRespond
- MercuryOperationInOut
- nemhandel
- MercuryOperationInOnly
- Invoice070

Faulty Services

Version

Copyright © 1999-2006, The Apache Software Foundation
Licensed under the Apache License, Version 2.0.

Done

To further test the service, run the LocalTestClient application as described above. The client should send documents to the following address:

http://localhost:8080/dk.gov.oiosi.examples.service/services/TestService/nemhandel

3.2.1 Mail transport configuration

The mail transport is configured using the following two files:

- WebContent/WEB-INF/services/TestService/META-INF/services.xml
- WebContent/WEB-INF/conf/axis2.xml

services.xml contains the POP3 server configuration, while axis2.xml contains the SMTP server configuration just like in the client example.

4 Lesson: Adapting the code for your own use

To use NemHandel you will need a function certificate. A guide (in Danish) for requesting one can be found here
https://www.nets-danid.dk/produkter/oevrige_signaturer/funktionssignatur/bestil_funktionssignatur/

4.1 Creating a Java key store for your NemHandel function certificate

To make your own keystore, you'll need to import the complete certificate chain, which means that for a function certificate store you will also have to import the TDC OCES Root certificate. You can find the root certificate here:

https://www.certifikat.dk/export/sites/dk.certifikat.oc/da/download/rodcertifikater/ocesca.crt

When you have your own certificate and the root, rename the root cert to have a filename ending .cer, and run the following commands:

```
keytool -importcert -keystore keyStore.jks -alias ocesca -file ocesca.cer
keytool –importcert -keystore keyStore.jks –alias func –file functional.cer
```

4.2 Configuring the examples to run on your own key store

Start by running the following command:

keytool –list –keystore keyStore.jks

to list the certificates in your store. Your certificate has been assigned an alias (the name given in the output). Remember this *alias* for later use.

The keystore used is set up in the dk.gov.oiosi.examples.service/WebContent/WEB-INF/services/TestService/META-INF/services.xml and dk.gov.oiosi.examples.client/cfg/policy.xml for the service and client respectively.

In each of these files, find the <ramp:RampartConfig> configuration element. Rampart is the WS-Security implementation we are using, so we will tell it where to find our store by altering

<ramp:user>

to our *alias* from above

<ramp:signatureCrypto>
<ramp:property name="org.apache.ws.security.crypto.merlin.file">

To the path were our key store can be found

```
<ramp:signatureCrypto>
        <ramp:property name="org.apache.ws.security.crypto.merlin.keystore.password">
```

To the password we use for our crypto store.

Whenever Rampart is in need of a private key, it will call the dk.firma.klient.webservice.PWCallback class found in both the client and the service examples. Alter the code in this class to return the password for your private key.