

OIO Service Oriented Infrastructure

OIOSI RASP Library for .NET

Version 1.3.0

Tutorials

Contents

1	Introduction	3
2	Overview of Lessons	3
3	Prerequisites	3
3.1	Resources	3
3.2	Sample Documents	3
4	Lesson: Configuration - App.Config	4
4.1	Configuring timeouts	7
5	Lesson: dk.gov.oios.communication	8
5.1	Request	8
5.2	The dk.gov.oios.test.request example project	10
5.3	Extended request	11
5.3.1	The UDDI lookup	15
5.4	Listener	16
6	Lesson: dk.gov.oios.uddi.IUddiLookupClien	18
7	Lesson: dk.gov.oios.extension.wcf	20
7.1	Email transport layer	22
7.2	Schema and schematron interceptors	22
7.3	Signature validation proof interceptor	22
7.4	XSLT Transformation interceptor	23
7.5	Ubiquitous properties interceptor	23
7.6	SenderPartyIdentifier and ReceiverPartyIdentifier headers	23
8	Lesson: Enabling tracing	24
8.1	Changing the trace settings	24
9	Lesson: Configuration – RaspConfiguration.xml	25
10	Lesson: Setting up an IIS hosted RASP service	26
10.1.1	Tips for solving common issues when hosting in IIS	30

1 Introduction

The purpose of this document is to guide developers in creating an application using the RASP library. The main components will be explained here, and example code showing how to set them up will be given.

This document represents a high-level view of the main communication components. More documentation is found

- As comments in the code
- As stand-alone test applications
- As NUnit tests
- By seeing code used as part of other components

Please visit <http://www.itst.dk/arkitektur-og-standarder/infrastruktur-og-felles-losninger/nemhandel> for more information on the OIOSI project.

2 Overview of Lessons

The basics of the RASP Library will be explained through different lessons, each describing one specific aspect of the API.

The lessons are

- Configuring RASP and Windows Communication Foundation
- Sending documents
- Extended Requests (including UDDI, LDAP and OCSP lookups)
- Receiving documents
- The interceptors
- Hosting a HTTP service in IIS

3 Prerequisites

3.1 Resources

All common resources can be found in the resource project `src/dk.gov.oiosi.resource`, and is added (by link) to other project that need the resources.

3.2 Sample Documents

There are several sample documents used in the tests of the rasp library. They are located in the `src\dk.gov.oiosi.raspProfile\Resources\Documents` directory.

4 Lesson: Configuration - App.Config

Each test project has an App.Config application configuration file, and so should every project that uses the RASP Library.

Most of the Windows Communication Foundation settings are done in App.Config, and this section will briefly cover each important section in the configuration.

It is recommended that the main sections described here are copied from one of the test projects, since the RASP Library is dependent on default settings (first and foremost the OiosiHttpEndpoint, OiosiEmailEndpoint, OiosiHttpBinding and OiosiEmailBinding).

App.Config sample 1

```
<system.serviceModel>

  <client>
    <!-- The HTTP endpoint-->
    <endpoint name="OiosiHttpEndpoint"
      address="http://... "
      binding="customBinding"
      bindingConfiguration="OiosiHttpBinding"
      contract=" dk.gov.oiosi.Rasp.Communication.Proxy.IClientProxyContract "
      behaviorConfiguration="OiosiEndpointCertificateBehavior">
      <identity>
        <dns value=" IT- og Telestyrelsen - OIOSI test
          virksomhedssignatur" />
      </identity>
    </endpoint>
  </client>
```

This sample shows a client endpoint, “OiosiHttpEndpoint”, using the attributes

- **Address** sets the remote endpoint address, to which requests will be sent
- **Binding/bindingConfiguration** sets the binding of this particular endpoint. In the binding (which is always a customBinding as App.Config sample 1 shows) we will describe the stack setup of the endpoint. More on this in later sections.
- **Contract** represents service contract implemented by the endpoint, describing what operations are supported and what SOAP actions they expect (and return). RASP endpoints should implement `dk.gov.oiosi.Rasp.Communication.Proxy.IRaspProxyContract`
- **behaviorConfiguration** defines the endpoint behaviour. The RASP Library uses behaviours to select certificates for sending and receiving.
- **Dns identity** overrides the “dns identity”, to allow clients to communicate with endpoints whose server certificates have domains that do not match the actual server DNS domain. This attribute only needs to be set when no certificate is configured programmatically,

App.Config sample 2

```
<behaviors>
  <endpointBehaviors>
    <behavior name="OiosiEndpointCertificateBehavior">
      <clientCredentials>
        <clientCertificate storeLocation="CurrentUser"
                           storeName="My"
                           x509FindType="FindBySerialNumber"
                           findValue=" 40 36 9f 5e"/>
      </clientCredentials>
      <serviceCertificate>
        <authentication revocationMode="NoCheck"/>
        <defaultCertificate storeLocation="LocalMachine"
                           storeName="Root"
                           x509FindType="FindBySerialNumber"
                           findValue="40 36 9f 5e" />
      </serviceCertificate>
    </behavior>
  </endpointBehaviors>
</behaviors>
```

App.Config sample 2 shows an endpoint behaviour. The behaviour references a client and a service certificate, and where to find them. As mentioned before, this behaviour is referenced from a client endpoint.

Note that the test applications use test certificates, to run with your own, live, certificates please change this section to refer to them instead. For a hint on how to find the serial number and store location/name of your certificate check the section named “*Importing certificates*” in the document “Rasp Library for .Net Installation”.

App.Config sample 3

```
<services>
  <service
    name="...communication.service.RaspServiceImplementation"
    behaviorConfiguration="OiosiServiceCertificateBehavior">

    <endpoint contract="...IServiceContract"
              address="mailto:server@oiositest.dk"
              binding="customBinding"
              bindingConfiguration="OiosiEmailBinding"
              >

    <identity>
      <dns value="IT- og Telestyrelsen - OIOSI test
                virksomhedssignatur " />
    </identity>

    </endpoint>
  </service>
```

App.Config sample 3 shows a service. Just like the client endpoint, the service has a behaviour, describing what certificate it uses (only this time it is a <serviceBehavior>) and a dns identity. Furthermore it defines an endpoint, in very much the same way as the client endpoint was defined. **The endpoint address should always be a mail address**, formatted <mailto:a@b.com> though, since the RASP Library only supports hosting email services. Also, most likely **the service and client endpoint should have different (but similar) bindings**, so that mail account settings don't collide (making the server snatch the client's mails and the other way around).

App.Config sample 4

```
<bindings>
  <customBinding>

    <!-- The OIOSI RASP mail binding -->
    <binding name="OiosiParamBinding"
      closeTimeout="00:01:30"
      openTimeout="00:01:30"
      receiveTimeout="00:01:30"
      sendTimeout="00:01:30">

      <clientSchemaValidationInterceptor />
      <clientSchematronValidationInterceptor />

      <reliableSession/>

      <ubiquitousProperties/>

      <clientSignatureValidationProofInterceptor />

      <security
messageSecurityVersion="WSSecurity10WSTrustFebruary2005WSSe
cureConversationFebruary2005WSecurityPolicy11BasicSecurity
Profile10"

        defaultAlgorithmSuite="Default"
        authenticationMode="MutualCertificate"
        requireDerivedKeys="false"
        securityHeaderLayout="Strict"
        includeTimestamp="true"
        keyEntropyMode="CombinedEntropy"
        messageProtectionOrder="SignBeforeEncrypt"
        requireSignatureConfirmation="false"

      />

      <clientPartyIdentifierHeader/>

      <textMessageEncoding messageVersion="Default"
        writeEncoding="utf-8" />

      <emailTransport
        outboxImplementation="dk.gov.oiosi.lesnikowskiMailPro
vider.SmtpOutboxLesnikowski,
dk.gov.oiosi.lesnikowskiMailProvider"

        inboxImplementation="dk.gov.oiosi.lesnikowskiMailProv
ider.Pop3InboxLesnikowski,
dk.gov.oiosi.lesnikowskiMailProvider"

        sendingServerAddress="oiositest.dk"
        receivingServerAddress="oiositest.dk"
        receivingUserName="client"
        receivingPassword="password"
        replyAddress="mailto:client@oiositest.dk" />
    </binding>
  </customBinding>
</bindings>
```

App.Config sample 4 shows an OiosiEmailBinding (the http binding looks the same, only with a httpTransport instead of an OiosiEmailTransport at the end). The binding defines the Windows Communications (WCF) stack that will be used, in the standard RASP case the binding should look as in App.Config sample 4 with a stack looking like

```
[Application layer]
[Xml Schema and Schematron validation]
[WS-Reliable Messaging]
[Ubiquitous properties]
[Signature validation proof]
[WS-Security]
[Sender/ReceiverPartyIdentifier SOAP header]
[Encoding]
[Transport (http or mail)]
```

To be able to use the custom elements, we first need to add their configuration extension at the end of the configuration file, telling WCF where to find the implementation. For example, we add a reference to the email binding implementation as shown in App.Config sample 5.

App.Config sample 5

```
<extensions>
  <bindingElementExtensions>
    <add name="oiosiEmailTransport"
        type="dk.gov.oiosi.extension.wcf.EmailTransport.EmailBindingExtensionElement, dk.gov.oiosi.library" />
  </bindingElementExtensions>
</extensions>
```

4.1 Configuring timeouts

There are several timeout settings in the app.config. Some of the overall timeouts are described here.

- SendTimeout is the overall timeout for a communication, i.e. including all RM messages back and forth between sender and receiver.
- Open- and CloseTimeout concerns the timeout of the creation of a connection from the sender to the receiver, and nothing else.
- ReceiveTimeout is the timeout for a process waiting for a message within a session, before deciding to time out the session.

5 Lesson: dk.gov.oiosi.communication

The communication namespace holds the 3 main classes for communicating with web services using the RASP stack. These are

- Request: Allows the simplest form of request using the RASP stack using either the email or http transport. Transport options (i.e. using RM, security, schema- and schematron validation) can be configured in App.Config as described in the earlier section.
- Listener: Receiving functionality using *the email transport*.

5.1 Request

dk.gov.oiosi.communication.Request is the main class for making RASP service calls.

For a concrete example of how to use the Request class, see the test project *dk.gov.oiosi.test.request*.

The sample below shows how to easily use Request to send an XML document to an http endpoint through the use of the method *GetResponse*. The code should be fairly straight forward.

Code sample 1

```
// Sends an xml document and receives a response
XmlDocument xdoc = new XmlDocument();
Request raspReq = new Request(new Uri("http://myEndpoint"));
Response response;

try{
    raspReq.GetResponse(new OiosiMessage(xdoc), out response);
}
catch(RequestShutdownException e){
    //No need to do anything in particular if one
    //isn't concerned with a nice shutdown
    // If the response variable is set, it's good to use
    // and your message has been acknowledged
}
```

Calls can be made to both http and mail services, and Request automatically detects which type of service is being called by looking at the scheme of the URI given. Http endpoint addresses MUST be formatted <http://address> and mail addresses <mailto:address@host.com>.

Furthermore Request supports asynchronous sending, through the method *BeginGetResponse*. The standard .Net async calling pattern is used.

Code sample 2 demonstrates how to make an async call (without using a callback method) to a mail endpoint.

Code sample 2

```
XmlDocument xdoc = new XmlDocument();
Request raspReq = new Request(
    new Uri("mailto:myMailEndpoint@server.com"));

// Starts the sending
IAsyncResult r = raspReq.BeginGetResponse(
    new OiosiMessage(new XmlDocument()), null);

// Ends the sending
Response response = raspReq.EndGetResponse(r);
```

Code sample 1+2 both give an URI as lone argument to the Rasp constructor, defining what endpoint messages will be sent to. However, Request offers two more constructors, presented in Code sample 3.

Code sample 3

```
// Takes the name of an endpoint in App.Config
public Request(string endpointConfigurationName);

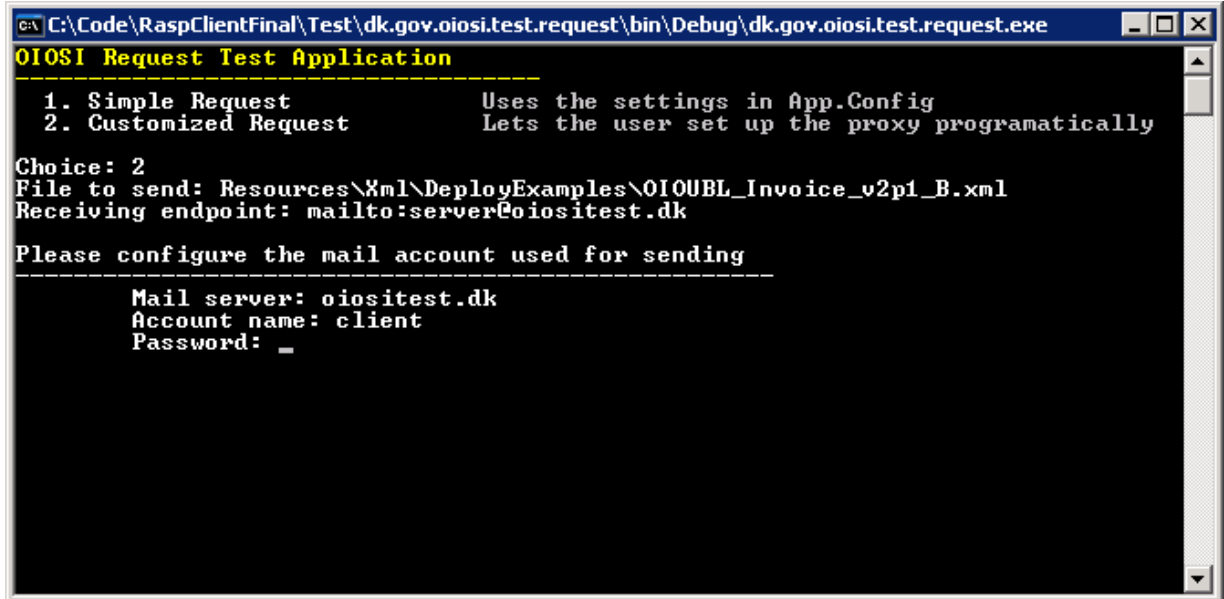
// Takes an endpoint, credentials and a sending policy
public Request(Uri endpointAddress,
    Credentials credentials,
    SendPolicy sendPolicy);
```

The first constructor in Code sample 3 takes the name of an endpoint in the application configuration file. See the file App.Config in the test dk.gov.oiosi.test.request for further reference to how the configured endpoint (Oios IHttpEndpoint) describes the service which we will call.

An endpoint configuration is needed no matter what constructor is used, and unless the first constructor in Code sample 3 is used **“Oios IHttpEndpoint”/“Oios EmailEndpoint” is used as a default, and therefore should always be present in the App-.Config file** when using the RASP Library. See the App.Config section for more information.

The second constructor in Code sample 3 takes an endpoint URI, just like the ones used in Code sample 1+2, but also takes programmatically set certificates (for sending and/or receiving). These will override any certificates given in App.Config. Furthermore a SendPolicy should be given, which can be used to programmatically set up the mail account information when running over the email transport.

5.2 The dk.gov.oiosi.test.request example project

A screenshot of a Windows command prompt window titled "C:\Code\RaspClientFinal\Test\dk.gov.oiosi.test.request\bin\Debug\dk.gov.oiosi.test.request.exe". The application displays a menu with two options: "1. Simple Request" (which uses settings in App.Config) and "2. Customized Request" (which lets the user set up the proxy programmatically). Option 2 is selected, leading to the display of a file path "Resources\Xml\DeployExamples\OIIOUBL_Invoice_v2p1_B.xml" and a receiving endpoint "mailto:server@oiositest.dk". The application then prompts the user to "Please configure the mail account used for sending". The user has entered "Mail server: oiositest.dk", "Account name: client", and "Password: _".

```
C:\Code\RaspClientFinal\Test\dk.gov.oiosi.test.request\bin\Debug\dk.gov.oiosi.test.request.exe
OIOSI Request Test Application
-----
1. Simple Request          Uses the settings in App.Config
2. Customized Request      Lets the user set up the proxy programatically

Choice: 2
File to send: Resources\Xml\DeployExamples\OIIOUBL_Invoice_v2p1_B.xml
Receiving endpoint: mailto:server@oiositest.dk

Please configure the mail account used for sending
-----
Mail server: oiositest.dk
Account name: client
Password: _
```

Figure 1 The Request test application

The library code is distributed with a test project named `dk.gov.oiosi.test.request` that demonstrates what has been learned in this lesson.

5.3 Extended request

This section demonstrates the following in addition to making plain Request:

- Gets identifiers from xml documents (e.g. EAN or OVT numbers) from documents using configurable xpath expressions
- Performs a UDDI lookup using these parameters
- Retrieves an endpoint certificate from LDAP based on information returned from UDDI
- Checks certificate revocation status against OCSP.

Please note that the project has moved into the samples namespace and changed name to “[dk.gov.oiosi.samples.consoleClientExample](#)”.

For a concrete example of how to use the Request class, see the test project `dk.gov.oiosi.test.requestTests`.

Code sample 4 shows how `ExtendedRequest` encapsulates all UDDI, OCSP, LDAP and document searching to send an XML document to an http endpoint through the use of the method `GetResponse`.

Code sample 4

```
// 1. Get client certificate:
X509Certificate2 cert = CertificateLoader.GetCertificateFromStoreWithSSN(
    "CVR:26769388-UID:1172691221366",
    StoreLocation.CurrentUser,
    StoreName.My
);

OcesX509Certificate clientCert = new OcesX509Certificate(cert);

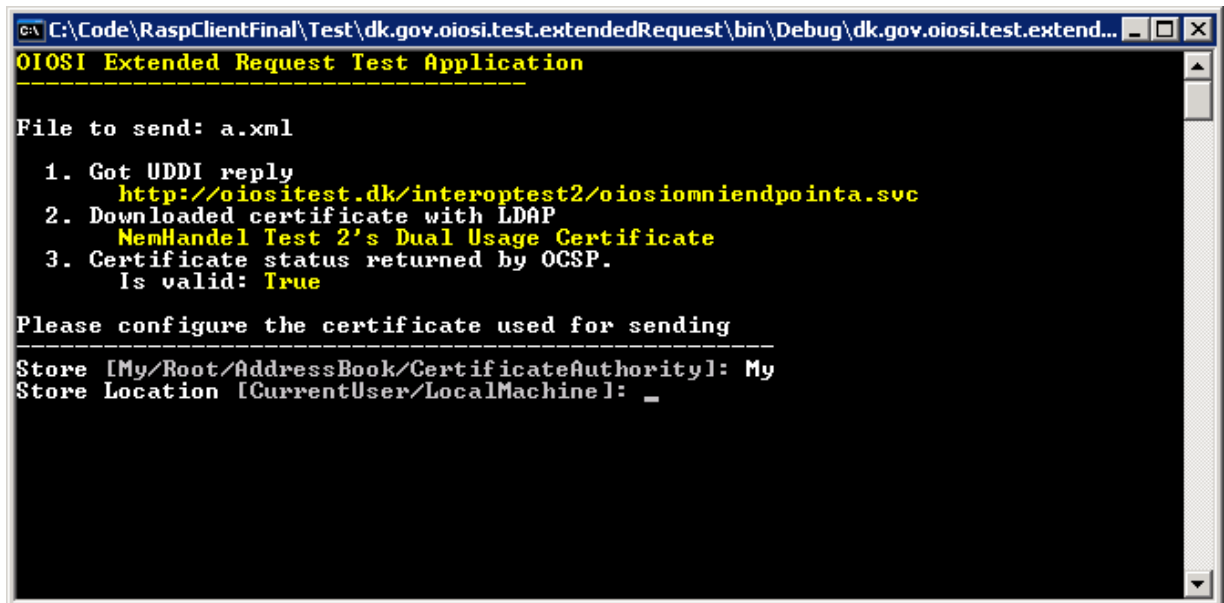
// 2. Define send policy:
SendPolicy sendPolicy = new SendPolicy("*");

// 3. Create request:
ExtendedRequest requestEx = new ExtendedRequest(clientCert, sendPolicy);

// 4. Create test message:
// 4.1 Load a test message from file:
XmlDocument xmlMsg = GetTestMessage();
OiosiMessage msg = new OiosiMessage(xmlMsg);

// 5. Get response:
Response response;
try {
    response = requestEx.GetResponse(msg);
} catch (Exception ex) {
    txtResult.Text += "RequestExtended failed: " + ex.ToString();
    return;
}
```

You can run the ExtendedRequest test sample by running the dk.gov.oiosi.test.extendedRequest project application, see below.



```
C:\Code\RaspClientFinal\Test\dk.gov.oiosi.test.extendedRequest\bin\Debug\dk.gov.oiosi.test.extend...
OIOSI Extended Request Test Application
-----
File to send: a.xml

1. Got UDDI reply
   http://oiositest.dk/interop2/oiosiomniendpointa.svc
2. Downloaded certificate with LDAP
   NemHandel Test 2's Dual Usage Certificate
3. Certificate status returned by OCSP.
   Is valid: True

Please configure the certificate used for sending
-----
Store [My/Root/AddressBook/CertificateAuthority]: My
Store Location [CurrentUser/LocalMachine]: _
```

The LDAP, OCSP and UDDI lookup components have both online and offline implementation of their interfaces, see the RASP client tutorials on how to switch between these.

You can choose which implementation to run by setting the corresponding factory configuration in the configuration file, see below.

You choose the implementation type by setting class + namespace + assembly name. The factory then instantiates this type dynamically. You may supply your own implementation of the OCSP, LDAP and UDDI interfaces.

Code sample 5

```
<ConfigurationSection xsi:type="RevocationLookupFactoryConfig">
  <ImplementationNamespaceClass>
    dk.gov.oiosi.security.revocation.ocsp.OcspLookup
  </ImplementationNamespaceClass>
  <ImplementationAssembly>dk.gov.oiosi.library</ImplementationAssembly>
</ConfigurationSection>
<ConfigurationSection xsi:type="LdapLookupFactoryConfig">
  <ImplementationNamespaceClass>
    dk.gov.oiosi.security.Ldap.LdapCertificateLookup
  </ImplementationNamespaceClass>
  <ImplementationAssembly>
    dk.gov.oiosi.library
  </ImplementationAssembly>
</ConfigurationSection>
<ConfigurationSection xsi:type="UddiLookupClientFactoryConfig">
  <ImplementationNamespaceClass>
    dk.gov.oiosi.uddi.UddiLookupClient
  </ImplementationNamespaceClass>
  <ImplementationAssembly>
    dk.gov.oiosi.library
  </ImplementationAssembly>
</ConfigurationSection>
```

Code sample 5 use OCSF for to check for revocation, while sample 6 use CRL lookup.

Code sample 6

```
<ConfigurationSection xsi:type="RevocationLookupFactoryConfig">
  <ImplementationNamespaceClass>
    dk.gov.oiosi.security.revocation.crl.CrlLookup
  </ImplementationNamespaceClass>
  <ImplementationAssembly>dk.gov.oiosi.library</ImplementationAssembly>
</ConfigurationSection>
<ConfigurationSection xsi:type="LdapLookupFactoryConfig">
  <ImplementationNamespaceClass>
    dk.gov.oiosi.security.Ldap.LdapCertificateLookup
  </ImplementationNamespaceClass>
  <ImplementationAssembly>
    dk.gov.oiosi.library
  </ImplementationAssembly>
</ConfigurationSection>
<ConfigurationSection xsi:type="UddiLookupClientFactoryConfig">
  <ImplementationNamespaceClass>
    dk.gov.oiosi.uddi.UddiLookupClient
  </ImplementationNamespaceClass>
  <ImplementationAssembly>
    dk.gov.oiosi.library
  </ImplementationAssembly>
</ConfigurationSection>
```

The OCSF-server is by default read from the certificate. This can be overridden by having a configuration in the OCSFconfig section :

Code sample 7

```
<ConfigurationSection xsi:type="OcspConfig">
  <DefaultTimeoutMsec>10000</DefaultTimeoutMsec>
  <ServerUrl>http://localhost/</ServerUrl>
</ConfigurationSection>
```

The offline implementations may then be configured differently from the online implementations.

Code sample 8

```
<ConfigurationSection xsi:type="OcspLookupTestConfig">
  <ReturnPositiveResponse>true</ReturnPositiveResponse>
</ConfigurationSection>
<ConfigurationSection xsi:type="LdapCertificateLookupTestConfig">
  <StoreLocation>LocalMachine</StoreLocation>
  <StoreName>Root</StoreName>
</ConfigurationSection>
<ConfigurationSection xsi:type="UddiLookupClientTestConfig">

<EndpointAddress>http://oiositest.dk/interoptest/oiosiOmniEndpointB.s
vc</EndpointAddress>
  <ExpirationDate>2012-06-01T00:00:00</ExpirationDate>
  <CertificateSubjectSerialNumber>
OID.2.5.4.5=CVR:26769388-UID:1172691221366 + CN=IT- og Telestyrelsen
- OIOSI test virksomhedssignatur, O = IT- og Telestyrelsen //
CVR:26769388, C=DK</CertificateSubjectSerialNumber>
  <TermsOfUseUrl>http://test.dk/termsOfUse.html</TermsOfUseUrl>
  <ServiceContactEmail>test@test.com</ServiceContactEmail>
  <Version>1.0.3</Version>
  <NewerVersionReference />
  <HasNewerVersion>false</HasNewerVersion>
</ConfigurationSection>
```

The configuration file also demonstrates how to configure `RaspDocumentType` configuration. This configuration is associated with the following information with the root element of an xml document:

- Validation schemas and schematron xslts
- Xpath expressions for finding e.g. an EAN number or other endpoint key types
- Associate a service and SOAP action with the document type

Most of this configuration points forward to the RASP client which uses this library for automatically sending business messages.

You can see the `RequestExtended` class for an example on how to string the OCSP, LDAP, UDDI, certificate checking and Request components together, either by using configuration or programmatically.

5.3.1 The UDDI lookup

The UDDI lookup of the extended request requires the UDDI connection to be configured in the `app.config` file, where it has its own HTTP binding.

The parameters of the UDDI lookup may additionally use a process definition filter. By setting the `BusinessProcessDefinitionTModel` and `RoleIdentifier` fields of the `LookupParameters` object, the result from the query is filtered using these criteria. If null, they are ignored.

For more information on how to use the `IUddiLookup` interface have a look at section 6.

5.4 Listener

dk.gov.oiosi.communication.Listener is the main class for hosting RASP email services.

Code sample 9

```
// Starts up a service, and adds a callback method for
// receiving incoming messages
static void Main(string[] args) {

    Listener listener = new Listener();

    // Add event handling
    listener.MessageReceive +=
        new MessageEventDelegate(MessageReceived);
    listener.ExceptionThrown +=
        new AsyncExceptionHandler(ExceptionThrown);

    listener.Start();

    // Wait until application is terminated . . .

    listener.Stop();
}

public static void MessageReceived(ListenerRequest msg,
    MessageProcessStatus status) {

    // Do something with the incoming message . . .

}

public static void ExceptionThrown(Exception ex) {
    // Do something with the asynchronous exception
}
```

Code sample 9 shows how to start a standard RASP service up, by reading service information from App.Config. In the configuration file for test project *dk.gov.oiosi.test.testMailService* one can see how a service, a service behavior and a binding are defined, enabling the code in Code sample 9 to run.

Code sample 9 also demonstrates how a *MessageEventDelegate* is used to define a callback method to be called whenever a message is received by the listener. The callback method gets a *ListenerRequest*, which is the incoming message, plus metadata (such as properties set by interceptors), and a status.

Furthermore an *AsyncExceptionHandler* is used to catch exceptions raised by the listener. It is strongly recommended to listen to this event, since it might be the one and only chance one has to discover that the service dies.

Code sample 10

```
public Listener (ListenerIdentity listenerIdentity);
```

Listener also offers a constructor, shown in Code sample 10 which takes a ListenerIdentity. Using this constructor allows one to programmatically set certificates that will be used instead of those given in App.config.

ListenerIdentity also allows one to set the transport binding programmatically (to override mail settings for example) and to make listener run a service of another type than the standard RASP implementation.

6 Lesson: dk.gov.oiosi.uddi.IUddiLookupClient

This section will have a short description on how to use the UddiLookupClient interface and the LookupParameters class for different scenarios.

There are several different kinds of lookup in that the interface IUddiLookupClient supports. The interface only has one method and that is for lookup that take some parameters and returns a list of responses. Se

```
/// <summary>
/// Translate interface for the ARS (Address Resolving Service) client.
/// </summary>
public interface IUddiLookupClient {
    /// <summary>
    /// Translate parametres
    /// </summary>
    /// <param name="parameters"></param>
    /// <returns></returns>
    List<UddiLookupResponse> Lookup(LookupParameters parameters);
}
```

The parameters that are used for the lookup can be different after what is searched for in the UDDI. It seems like there are three major lookups that is needed by suppliers and these are; first one is to find all that a specific identifier supports, second one is to find the endpoint to a specific identifier and specific document type, finally the third one is to find the endpoint to a specific identifier, specific document type and specific profile.

First there is how to make a lookup that get all services for a specific identifier, then you have to use the following constructor:

```
public LookupParameters(
    Identifier identifier,
    List<EndpointAddressTypeCode> acceptedTransportProtocols)
```

Where the identifier is given as the first parameter and the second parameter is what transport protocols the client can support.

Second lookup type takes three parameters; the first parameter is the identifier, the second parameter is the service identifier in the UDDI and the third parameter is the accepted transport protocols. The second parameter is a bit tricky to find but it is defined and found on the UDDI. Se below for the method parameters:

```
public LookupParameters(
    Identifier identifier,
    UddiId serviceId,
    List<EndpointAddressTypeCode> acceptedTransportProtocols)
```

Here the UDDIID on the portType tModel is used as the value in the parameter. The portType can for an example correlate to a document type (invoice) in a process (billing).

Third lookup type takes four parameters where the third parameter is different than the other two lookup constructors. This parameter is a list of UddiId's on the profiles that the service must support. The method looks like the following:

```
public LookupParameters(
    Identifier identifier,
    UddiId serviceId,
```

```
List<UddiId> profileIds,  
List<EndpointAddressTypeCode> acceptedTransportProtocols)
```

The lookup will accept a service as a result if just one of the profiles in the list is supported by it.

There are more constructors but they are not needed to send documents over the RASP protocol, so they are not described here in this document.

7 Lesson: dk.gov.oiosi.extension.wcf

The RASP library comes with several extensions to the .Net 3 Windows Communications Foundation framework, that can all be found under the dk.gov.oiosi.extension.wcf and dk.gov.oiosi.raspProfile.extension.wcf namespaces.

These extensions come in the form of binding elements that are inserted into the communication stack, where they intercept and handle the in- or outgoing message according to their functionality.

The extensions available in version 1.0 are

Transport layers

- The OIOSI Email transport

Interceptors

- The schema interceptor
- The schematron interceptor
- The signature validation proof generator
- The XSLT transformer
- The ubiquitous message property interceptor

Headers

- The party identifier headers

These stack elements are added in the App.Config file, as seen in App.Config sample 4, but only after adding reference to each of the elements configuration extension as seen in App.Config sample 5.

Server and client side interceptors have been implemented in different manners because of the different ways the two handles SOAP messages wherefore one has to make sure the correct interceptor binding element has been selected.

App.Config sample 5

```
<!-- Our binding extension, letting WCF know where our custom WCF components are
implemented -->
<extensions>
  <bindingElementExtensions>
    <add name="emailTransport"
      type="dk.gov.oiosi.extension.wcf.EmailTransport.EmailBindingExtensionElement,
dk.gov.oiosi.library" />

    <!-- Signature validation proof generation -->
    <add name="serverSignatureValidationProofInterceptor"
      type="dk.gov.oiosi.extension.wcf.Interceptor.Security.ServerSignatureValidatio
nProofBindingExtensionElement, dk.gov.oiosi.library" />
    <add name="clientSignatureValidationProofInterceptor"
      type="dk.gov.oiosi.extension.wcf.Interceptor.Security.ClientSignatureValidatio
nProofBindingExtensionElement, dk.gov.oiosi.library" />

    <!-- Schema validation -->
    <add name="serverSchemaValidationInterceptor"
      type="dk.gov.oiosi.extension.wcf.Interceptor.Validation.Schema.ServerSchemaVal
idationBindingExtensionElement, dk.gov.oiosi.library" />
    <add name="clientSchemaValidationInterceptor"
      type="dk.gov.oiosi.extension.wcf.Interceptor.Validation.Schema.ClientSchemaVal
idationBindingExtensionElement, dk.gov.oiosi.library" />

    <!-- Schema validation -->
    <add name="serverSchematronValidationInterceptor"
      type="dk.gov.oiosi.extension.wcf.Interceptor.Validation.Schematron.ServerSchem
atronValidationBindingExtensionElement, dk.gov.oiosi.library" />
    <add name="clientSchematronValidationInterceptor"
      type="dk.gov.oiosi.extension.wcf.Interceptor.Validation.Schematron.ClientSchem
atronValidationBindingExtensionElement, dk.gov.oiosi.library" />

    <!-- Custom RASP headers-->
    <add name="clientPartyIdentifierHeader"
      type="dk.gov.oiosi.raspProfile.extension.wcf.Interceptor.CustomHeader.ClientPa
rtyIdentifierHeaderBindingExtensionElement, dk.gov.oiosi.raspProfile" />
    <add name="serverPartyIdentifierHeader"
      type="dk.gov.oiosi.raspProfile.extension.wcf.Interceptor.CustomHeader.ServerPa
rtyIdentifierHeaderBindingExtensionElement, dk.gov.oiosi.raspProfile" />

    <!-- Adds parameters to ALL messages, including RM messages -->
    <add name="ubiquitousProperties"
      type="dk.gov.oiosi.extension.wcf.Interceptor.UbiquitousProperties.UbiquitousPr
opertiesBindingExtensionElement, dk.gov.oiosi.library" />
  </bindingElementExtensions>

  <behaviorExtensions>
    <!-- Behavior that selects headers to be added for signing -->
    <add name="signCustomHeaders"
      type="dk.gov.oiosi.extension.wcf.Behavior.SignCustomHeadersBehaviorExtens
ionElement, dk.gov.oiosi.library, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=null" />
  </behaviorExtensions>
</extensions>
```

7.1 Email transport layer

App.Config stack usage:

```
<emailTransport
  outboxImplementation="dk.gov.oiosi.lesnikowskiMailProvider.SmtpOutboxLe
  snikowski, dk.gov.oiosi.lesnikowskiMailProvider"
  inboxImplementation="dk.gov.oiosi.lesnikowskiMailProvider.Pop3InboxLesn
  ikowski, dk.gov.oiosi.lesnikowskiMailProvider"
  sendingServerAddress="xxx"
  receivingServerAddress="xxx"
  receivingUserName="xxx"
  receivingPassword="xxx"
  replyAddress="mailto:a@b.com" />
```

The OIOSI Email binding has the following settings

- **outboxImplementation** – the qualified name (“namespace.name, assembly”) of a class that inherits the virtual baseclass Outbox. This implementation handles sending mails. The RASP Library comes with a standard implementation, using the Lesnikowski mail library, as shown in App.Config sample 4.
- **inboxImplementation** – same as above, only implementing Inbox and handling receiving of mails
- **sendingServerAddress/sendingUserName/sendingPassword** – login info for the mailserver used for sending mails. *Only one service or client should use the same binding at a time, to make sure there are no collisions.*
- **receivingServerAddress/receivingUserName/receivingPassword** - login info for the mailserver used for receiving mails *Only one service or client should use the same binding at a time, to make sure there are no collisions.*
- **replyAddress** – the address to which the receiver of messages should reply. In other words, the mailbox that has username *receivingUserName* on server *receivingServerAddress*.
- **pollingInterval** – in seconds, describing how often to poll the mail server when waiting for incoming mails

7.2 Schema and schematron interceptors

The xml validators have the following settings

- **ValidateRequest** - If true it will validate the xml on request. Default is true.
- **ValidateResponse** - If true it will validate the xml on response. Default is true. The current configuration does not return any valid xml so this is disabled.
- **FaultOnRequestValidationException** - If true it will send soap fault to the client if the validation fails. If false the message will continue up the stack and any validation failure is added as a custom property to the message. Default is true.

7.3 Signature validation proof interceptor

The server side signature validation proof interceptor has the following options

- **FaultOnRequestValidationException** - should a SOAP fault be sent on exceptions? Default is true.

The client side signature validation proof interceptor has no options.

Note that the `ServerSignatureProof` and `ClientSignatureProof` interceptors must be located between the RM layer and the security layer as seen in App.Config sample 4.

7.4 XSLT Transformation interceptor

The XSLT transformation transforms the incoming XML, and has the following options

- **FaultOnTransformationException** – should a SOAP fault be sent on exceptions?
- **PropagateOriginalMessage** – The original XML will be added as a message property

Note that the XSLT interceptor must be placed above the `ReliableMessaging` layer.

7.5 Ubiquitous properties interceptor

The ubiquitous properties interceptor adds ubiquitous message properties to all messages that passes it (as opposed to normal WCF Message properties, that will only be added to the payload message).

Ubiquitous properties are added to the `OiosiMessage` before sending, and need to be given a unique string as an identifier, which later stack layers need to be familiar with if they would like to read the property.

```
OiosiMessage msg = new OiosiMessage();  
msg.UbiquitousProperties.Add("MyProperty", new object());
```

Note that the ubiquitous properties interceptor needs to be located under the `ReliableMessaging` layer as seen in App.Config sample 4.

7.6 SenderPartyIdentifier and ReceiverPartyIdentifier headers

In `dk.gov.oiosi.raspProfile.communication.extension.wcf` an additional interceptor can be found, which is used to add the obligatory RASP SOAP headers `<SenderPartyIdentifier>` and `<ReceiverPartyIdentifier>`.

The value of the headers is configured by adding an `PartyIdentifierSettings` object as an ubiquitous property to the message to be sent, as seen in the code below. The name of the ubiquitous property must be the value found in the constant `MessagePropertyKey` on the `PartyIdentifierHeaderSettings` class.

```
OiosiMessage msg = new OiosiMessage();  
  
string key = PartyIdentifierHeaderSettings.MessagePropertyKey;  
PartyIdentifierHeaderSettings partyIdentifierSetting = new  
PartyIdentifierHeaderSettings(senderID, receiverID);  
  
msg.UbiquitousProperties[key] = partyIdentifierSetting;
```

8 Lesson: Enabling tracing

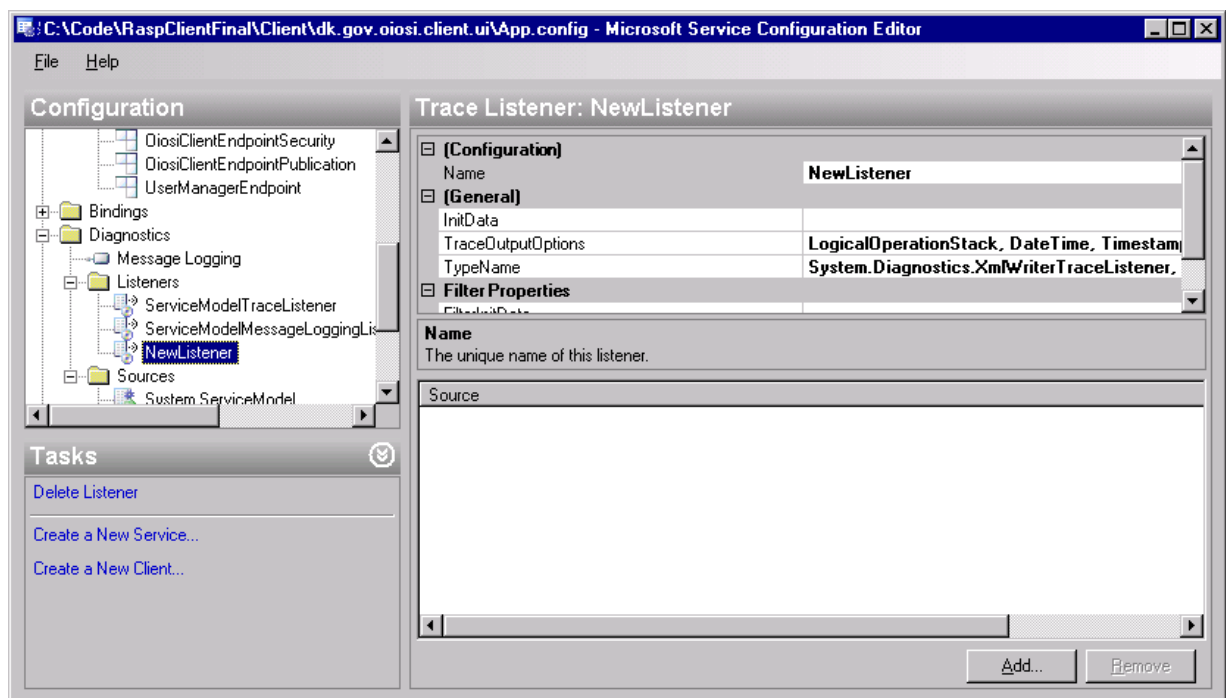
Windows communication enables both Message level logging and internal system traces, both which can be used with the RASP library. Furthermore the RASP library can add it's own internal system logs to the same (or a different) log file as WCF. Beware that when using traces for the library on the Verbose tracing level the trace files will very quickly become very large, so tracing at this level should only be enabled for advanced debugging.

At the very top of all the App.Config files that are distributed with the RASP library source, there should be a section called `<system.diagnostics>`. This section has been disabled by making it an xml comment, and to enable it again just remove the `<!--` and `-->` from before and after it.

When tracing has been enabled two files, App_Traces.svclog and App_Messages.svclog, which should be viewed using the application SvcTraceViewer.exe that comes with the Windows SDK. App_Traces will contain internal logs and App_Messages will contain all the SOAP messages sent and/or received.

8.1 Changing the trace settings

To change the tracing options it is recommended that the App.Config file is edited in the Windows SDK application SvcConfigEditor.exe.



In the configuration editor, under Diagnostics there will be two important sections; Listeners and Sources.

Under listeners you can add new trace listeners that will write to other files than the above mentioned two. Under Sources you can add more sources (for example your own WCF extensions) or change to what Listener the existing Sources will be written (if for example you would like to separate RASP logs from WCF logs). It is also here where you can change on what detail level to log. "Warning" is recommended to not clog the log files.

9 Lesson: Configuration – RaspConfiguration.xml

This section briefly describes how to access the dynamic configuration file, that supplements App.Config, RaspConfiguration.xml and change loaded library versions from live to test versions.

The configuration file can be found here:

```
"<rootDrive>:\Documents and Settings\Administrator\Application  
Data\RaspClient\Configuration\RaspConfiguration.xml"
```

Note that you cannot change the settings while the application is running.

You can choose to use offline test stub versions of the LDAP, OCSP and UDDI libraries. This is suitable for testing in offline environments or to fix some parameters of the test.

You can set this in the factory configuration sections of the config.

To do use test stubs do the following:

- **LDAPLookupFactory** – change the implementation namespace class from “dk.gov.oiosi.security.Ldap.LdapCertificateLookup” to “dk.gov.oiosi.security.Ldap.LdapCertificateLookupTest”.
- **LdapLookupFactoryConfig** – change the implementation namespace class from “dk.gov.oiosi.security.Ldap.LdapCertificateLookup” to “dk.gov.oiosi.security.Ldap.LdapCertificateLookupTest”.
- **RevocationLookupFactoryConfig** – change the implementation namespace class from “dk.gov.oiosi.security.revocation.ocsp.OcspLookup” to “dk.gov.oiosi.security.revocation.ocsp.OcspLookupTest”.

When you use the test stubs, you can configure the behaviour of each of them. You can do this in the following sections:

- LdapCertificateLookupTestConfig: Here you can configure a certificate that the LDAP client should always return.
- OcspLookupTestConfig: Here you can set the response that the OCSP client always will return in response to a question of certificate validity (true/false).
- UddiLookupClientTestConfig: Here you can statically configure an UDDI response, regardless of lookup parameters. Parameters include the endpoint address and certificate subject.

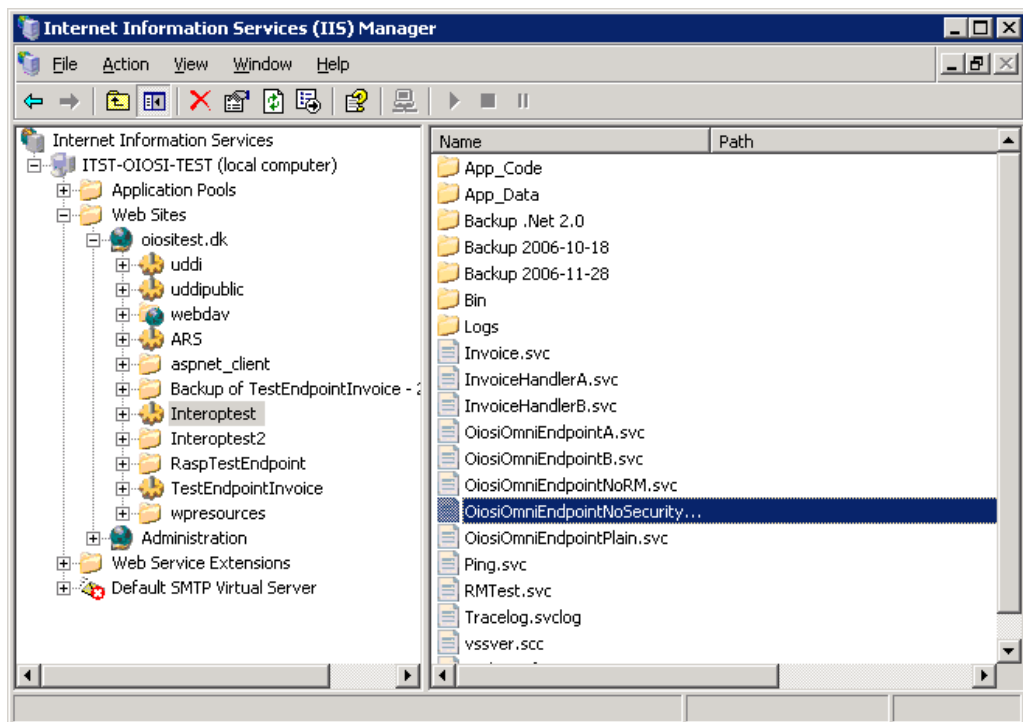
10 Lesson: Setting up an IIS hosted RASP service

The `dk.gov.oiosi.communication.Listener` class only supports hosting mail endpoints, but one might want to host an HTTP endpoint, in which case it is recommended that this endpoint should be hosted by Microsoft Internet Information Services (IIS).

A zip file called `RaspHTTPEndpointExample.zip` is distributed with the RASP Library and can be found under the Resources folder in the `RaspLibrary` Visual Studio project. In this zip archive you will find some code and a `Web.Config` file (which will act as a substitute for the `App.Config` file while hosting our service in IIS).

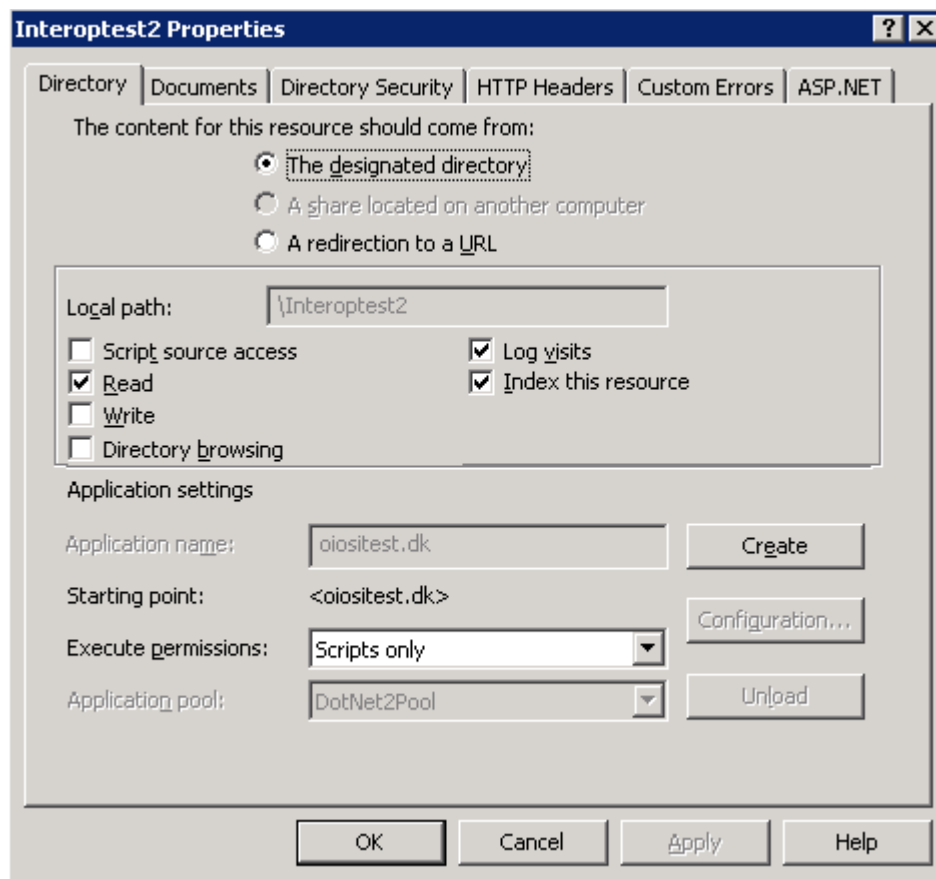
To get started, simply unzip the files into `C:\Inetpub\wwwroot\` (assuming that you have your Windows installation on the C drive).

Then you should open the IIS manager (found under `Start->Control Panel->Administrative Tools`, alternatively as a subsection of `Start->Control Panel->Administrative Tools->Computer Management`).

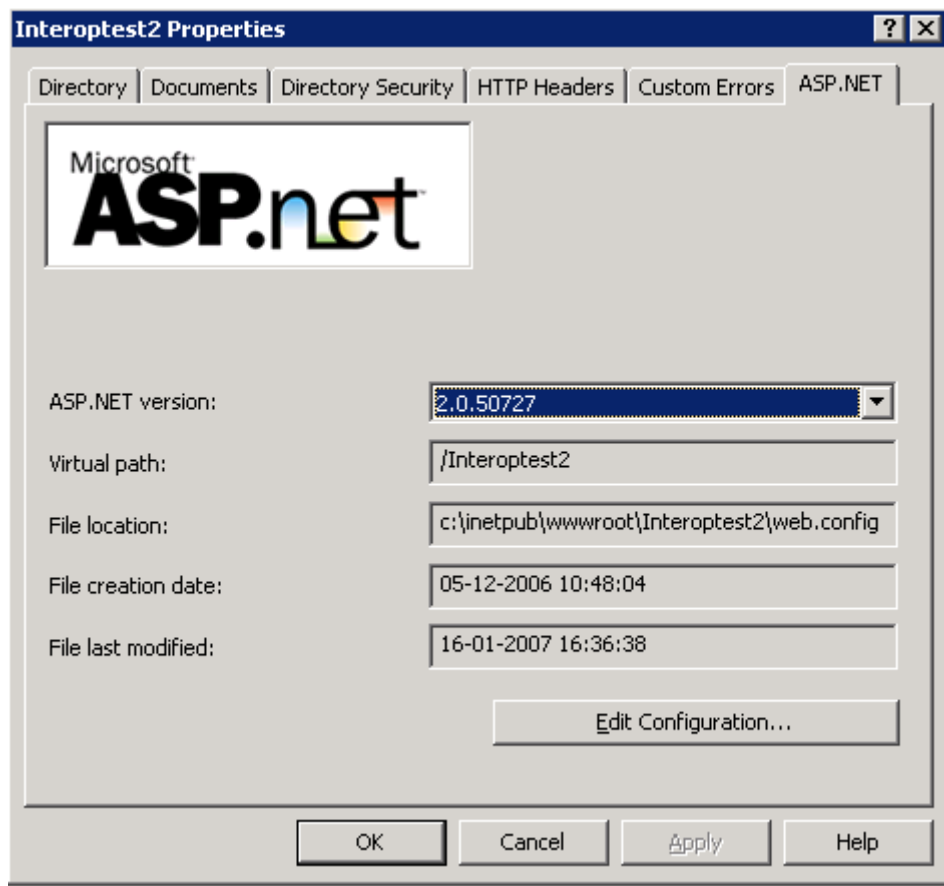


Under `local computer\Web sites` there should be a default web site with the folder `C:\Inetpub\wwwroot\` as its home directory. If there is no web sites please refer to Microsoft help for setting a web site up.

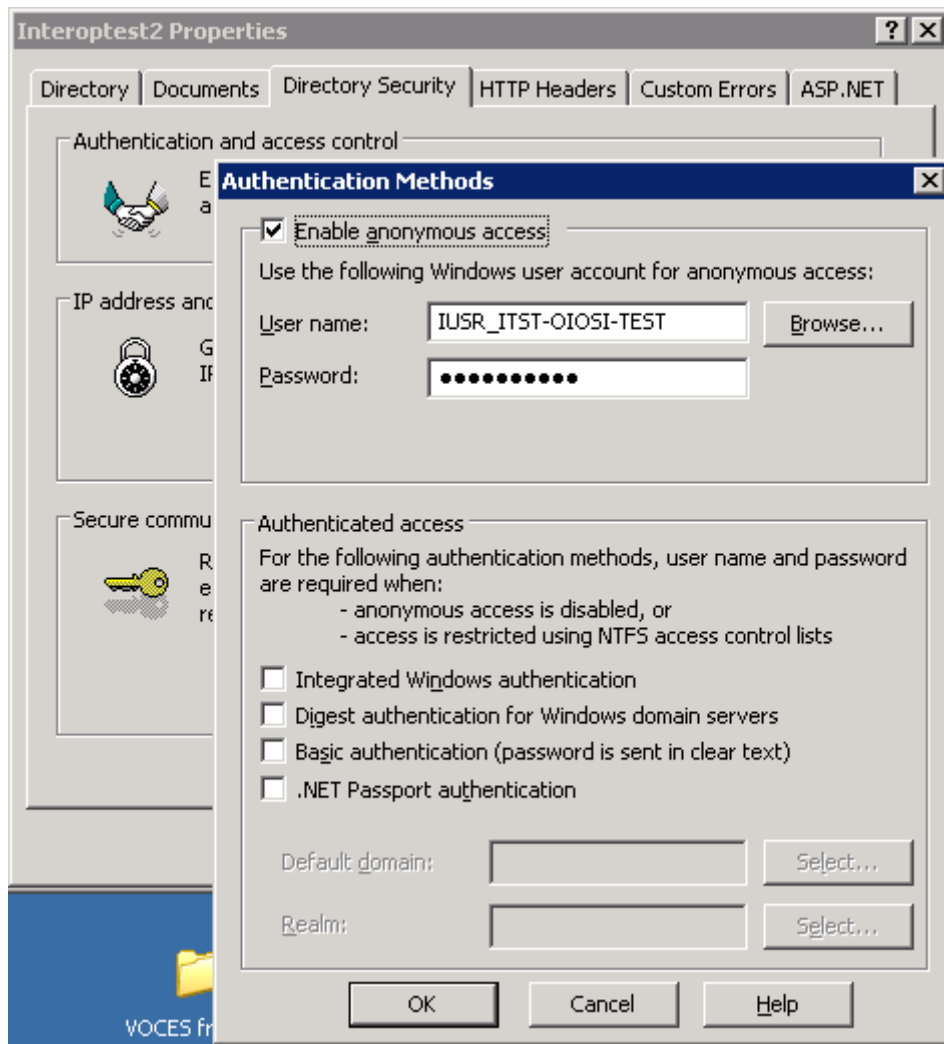
As a sub folder of this web site you should see the folder you just unzipped called `RaspTestEndpoint`. Right click on this folder and select *Properties*.



In the properties window, under the *Directory* tab push the *Create* button.



In the properties window, under the *ASP.NET* tab, select Asp.Net version 2.0



Under the *Directory Security* tab, *Authentication and access control*, make sure anonymous access is enabled.

Now you should be able to see that your service is running by opening

<http://localhost/RaspTestEndpoint/OiosiOmniEndpoint.svc>

in a browser (such as Internet Explorer) .

10.1.1 Tips for solving common issues when hosting in IIS

If you have problems contacting the ISS service from outside, you may try and look into the firewall settings.

If your endpoint is not working, you may try the following:

- Go to the IIS application pool property window
- Select the “Identity” tab.
- Change the account to “local system”

If you do not want to elevate permission on the whole of application pool, you can try the following.

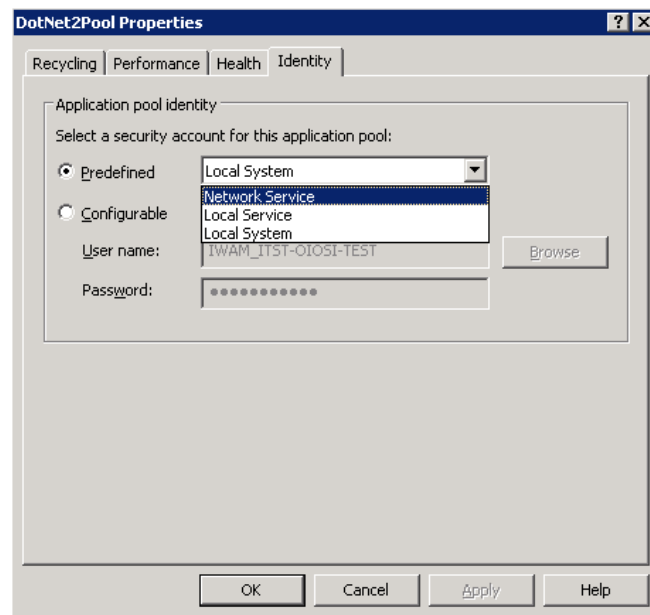
- Creating a new application pool, where all web sites are running on the same .Net version (v. 2.0)
- If that doesn’t work, perhaps the application pool doesn’t have rights to use the certificate. Try:
 - Delete the certificate OiosiTestVOCES.pfx from the MMC window.
 - Run the command line tool WinHttpCertCfg (<http://www.microsoft.com/downloads/details.aspx?familyid=c42e27ac-3409-40e9-8667-c748e422833f&displaylang=en>)
 - Re-install the certificate like:

```
>WinHttpCertCfg.exe -i NemHandelTest2.pfx -c LOCAL_MACHINE\MY -a "NetworkService" -p Test1234
```


(imports the pfx file to the personal store on local machine, for the NetworkService account, using the password Test1234)
 - Grant acces to the certificate for asp.net by running

```
>WinHttpCertCfg.exe -c LOCAL_MACHINE\MY -s "NemHandel Test 2" -g -a "aspnet"
```


(where “NemHandel Test 2” is part of the subject string of the certificate you just imported)
 - In web.config, change the location of the certificate from “Root” to “My” store.



- In the IIS manager, right click on the application pool you're running on (if it is the default, you might want to create a new one) and make sure that under the Identity tab the "Network service" security account is selected