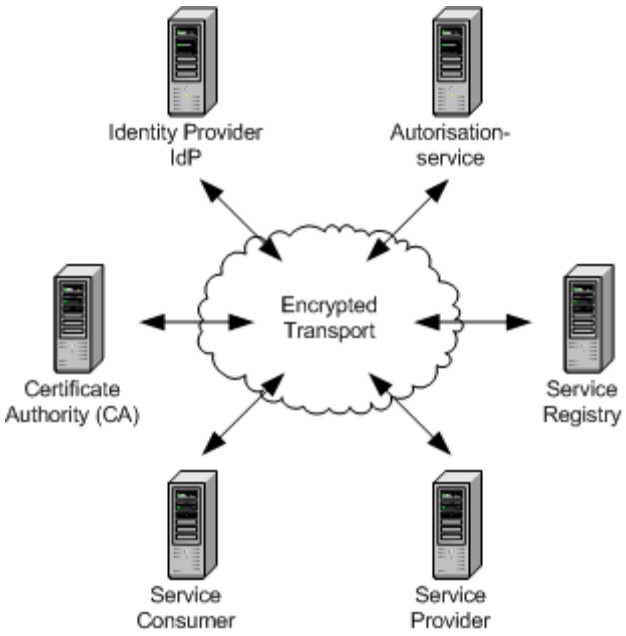
The SOSI Library

Programmers Guide 2.1



1 Introduction

This document is a guide for users of the SOSI library, also known as Seal.Java. The document contains information about how to install and configure the library, and documents details on how to use the library as a service consumer or a service provider.

This document is not a design document and hence will not go into detail about e.g. how XML signature is used in the SOSI envelope format etc. Neither will it cover all the basic concepts of Webservice Single-Sign-On, federations etc. If you need information about the concepts etc. please refer to “Den Gode Webservice” documentation.

The SOSI library is presently implemented as a Java library and the reader must therefore be an intermediate Java programmer. The reader must also have basic knowledge about public key cryptography (signing) and XML.

The most recent version of this document can be found online here:

http://svn.softwareboersen.dk/sosi/trunk/modules/seal/src/site/SOSI%20programmers%20guide.pdf

2 Library concepts

The primary goal for the SOSI library is to encapsulate most of the complex logic in the SOSI concept behind a very simple API. It has been our goal to construct a single point of entry for all programmers (a factory) from which it is possible to acquire simple model objects (POJO's[1]) that are representing the core concepts in the SOSI scheme, e.g. a Message or an ID Card.

Once the programmer has constructed these model objects, it is possible to "serialize" them into XML and vice versa. The de-serialization (from XML to model objects) is also done through the factory. **Error! Reference source not found.** below shows a very simple flow, where a service consumer (e.g. a medication system) is creating a Request message, setting it up, serializing it into XML and sends it to a Service consumer.

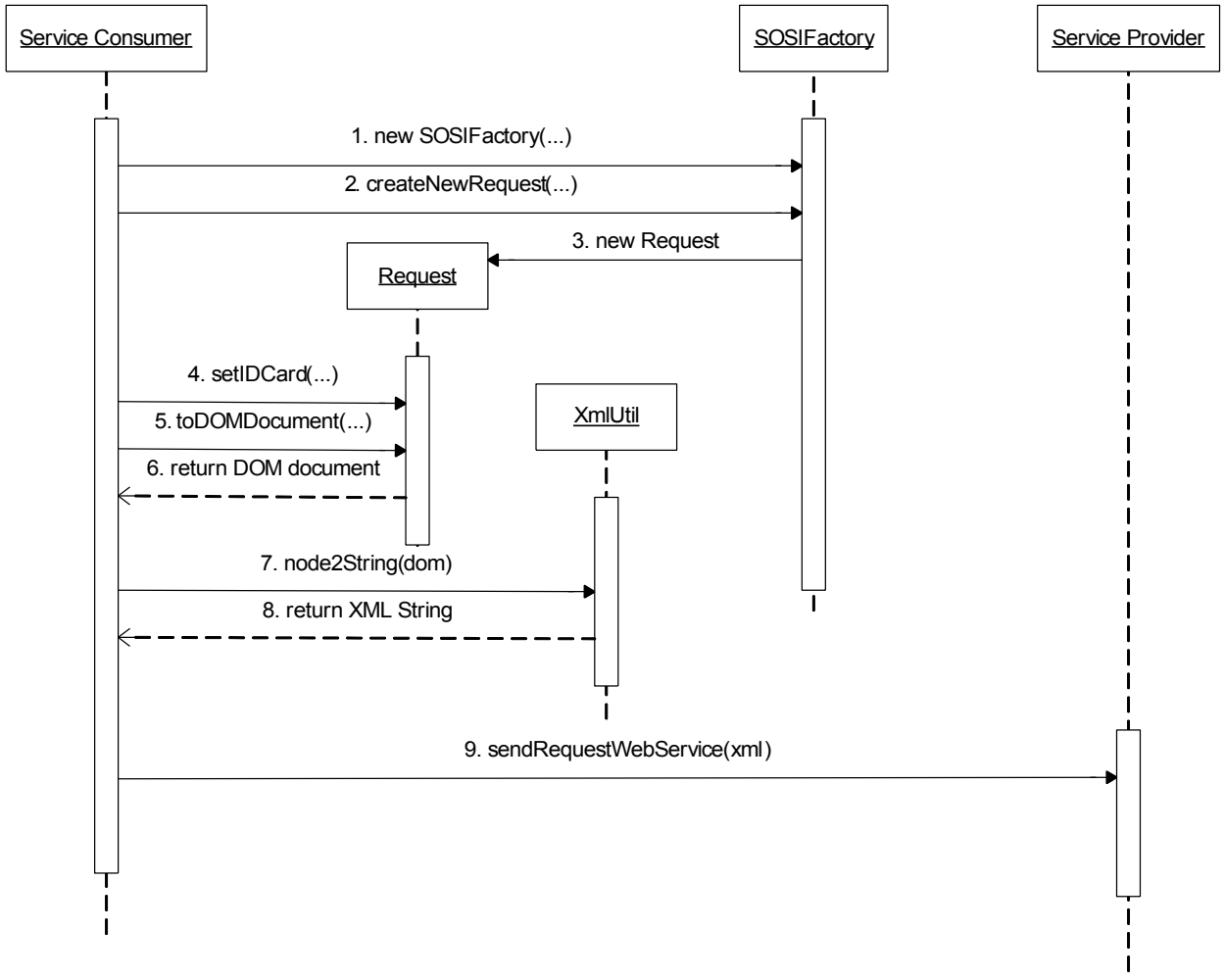


Figure 1 - a simple service consumer usage.

[1] "Plain Old Java Object"

As of version 2.1 Seal.Java now includes the IDWSHFactory. Version 2.1 marks the beginning of IDWSH support. Further versions of Seal.Java will greatly expand the support and workflow using IDWSH.

Seal.Java 2.1 only supports IDWSH identity tokens. The flow for constructing and using Identity tokens is shown in Figure 2.

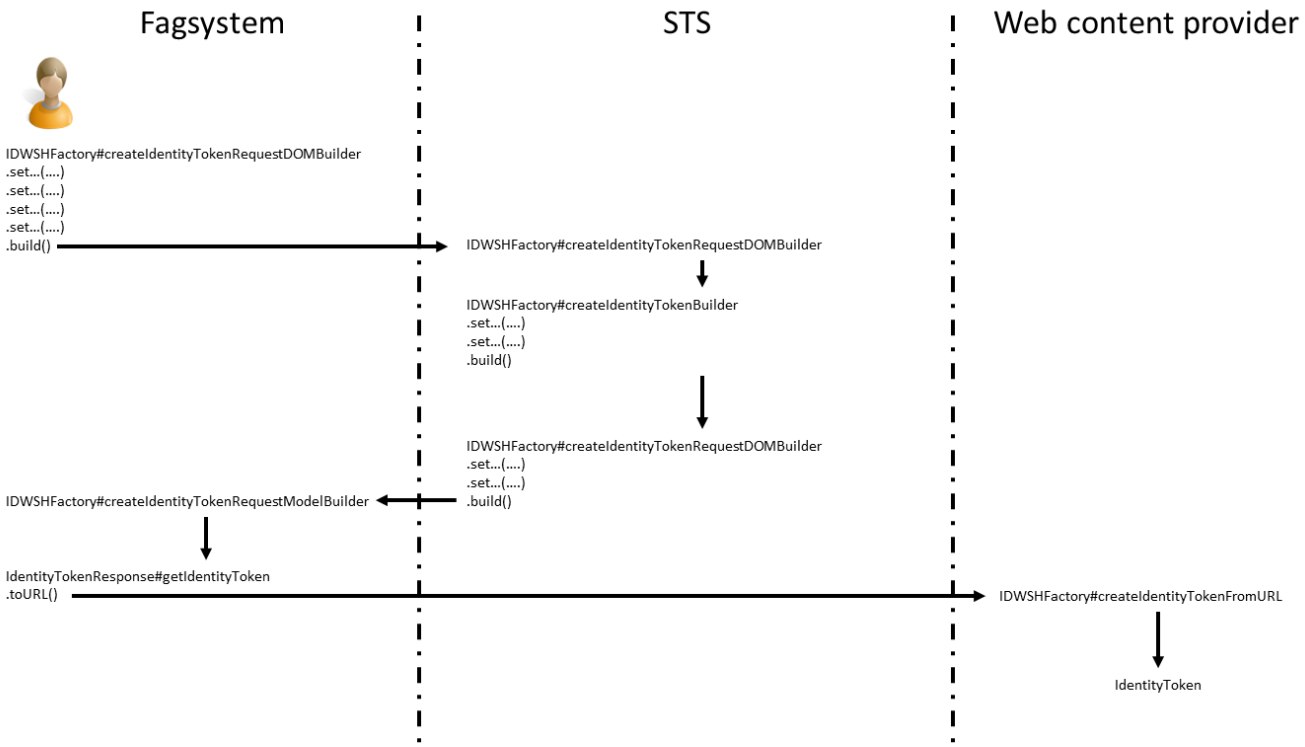


Figure 2 - IDWSH Identity token workflow.

As can be seen from the two examples, the programmer does not at any time produce (or code) any XML, the programmer does not make digests or digital signatures. This kind of complex logic is encapsulated behind elements from the library.

In Seal 2.1.4 the OIOSAMLFactory was introduced which provides functionality to create, parse, sign and validate OIOWS-Trust messages that are used when exchanging OIOSAML assertions that are issued by and IdP to SOSI IDCards.

The core POJO's make up a simple object hierarchy as shown below in Figure 3:

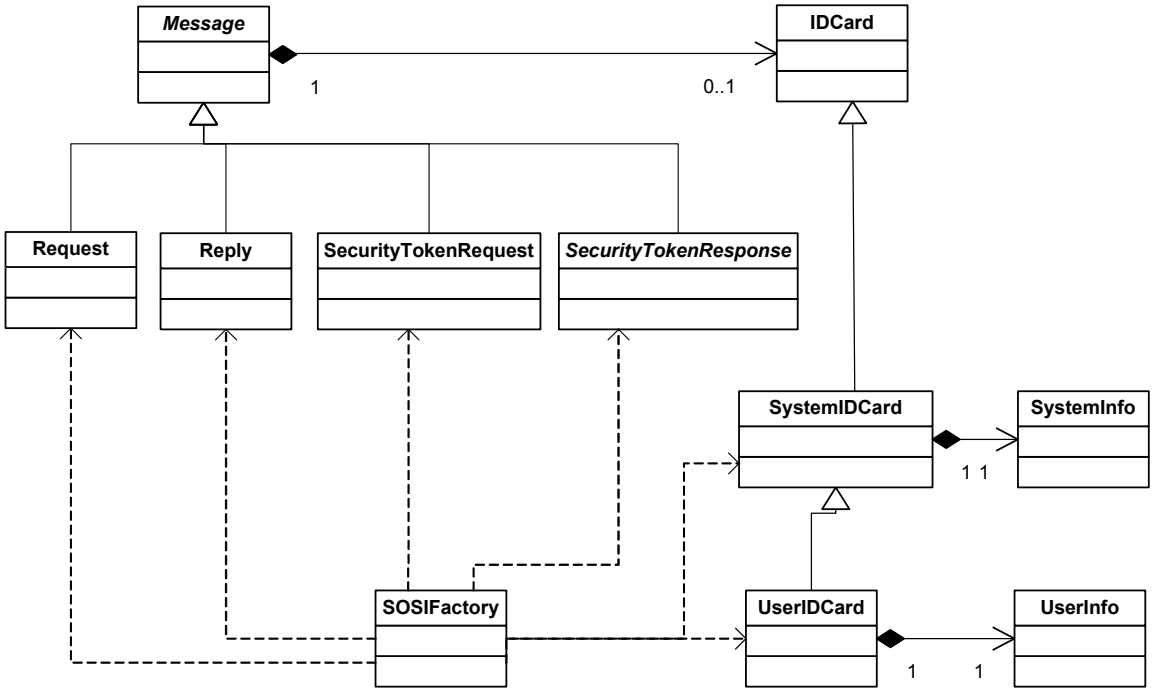


Figure 3 – The POJO class relationships

The class diagram is almost self-explanatory. To the left you find the main abstraction of messages (**Request**, **Reply** etc). **SecurityTokenRequest** and **SecurityTokenResponse** are messages used for establishing federation credentials.

Messages can have an ID-card attached. This ID-card can either be a *system* ID-card or a *user* ID-card, depending on the type of service. If the service requires information about a specific user in order to execute, the ID-card needs to be a **UserIDCard**. An example of such a service could be requesting information about medication for patient from the Danish Medicines Agency. In this case the service provider needs proof of the user's identity. In other cases, e.g. when delivering data *to* the Danish Medicines Agency in nightly batches, it is only necessary to know the identity of the system. In this case a **SystemIDCard** is sufficient.

The entry point for programmers is always the "**SOSIFactory**", the "**IDWSHFactory"** or the **"OIOSAMLFactory"**. From here it is possible to create new request objects, reply objects, ID-cards, IdentityTokens, etc. as well as to de-serialize XML into "copies" of objects from the sender side.

Both factories have a "CredentialVault" associated. The CredentialVault is a simple encapsulation of PKCS[2] elements: a (possibly empty) set of trusted X.509 certificates and zero or one public-private key-pair.
The CredentialVault is passed to the SOSIFactory at construction time.

[2] Public Key Crypto System

The concrete realization of the vault may vary with the environment in which the library is used. In a simple environment, the realization could be a file based CredentialVault that reads a PKCS#12 file and trusted X.509 certificates from the disk[3], or a realization based on a database/cache in a complex J2EE environment. If you have no need of “strong” credentials (i.e. on authentications level 1), you can construct SOSIFactory with EmptyCredentialVault, which is an empty implementation of the CredentialVault interface.

The class relationships are shown below in Figure 4.

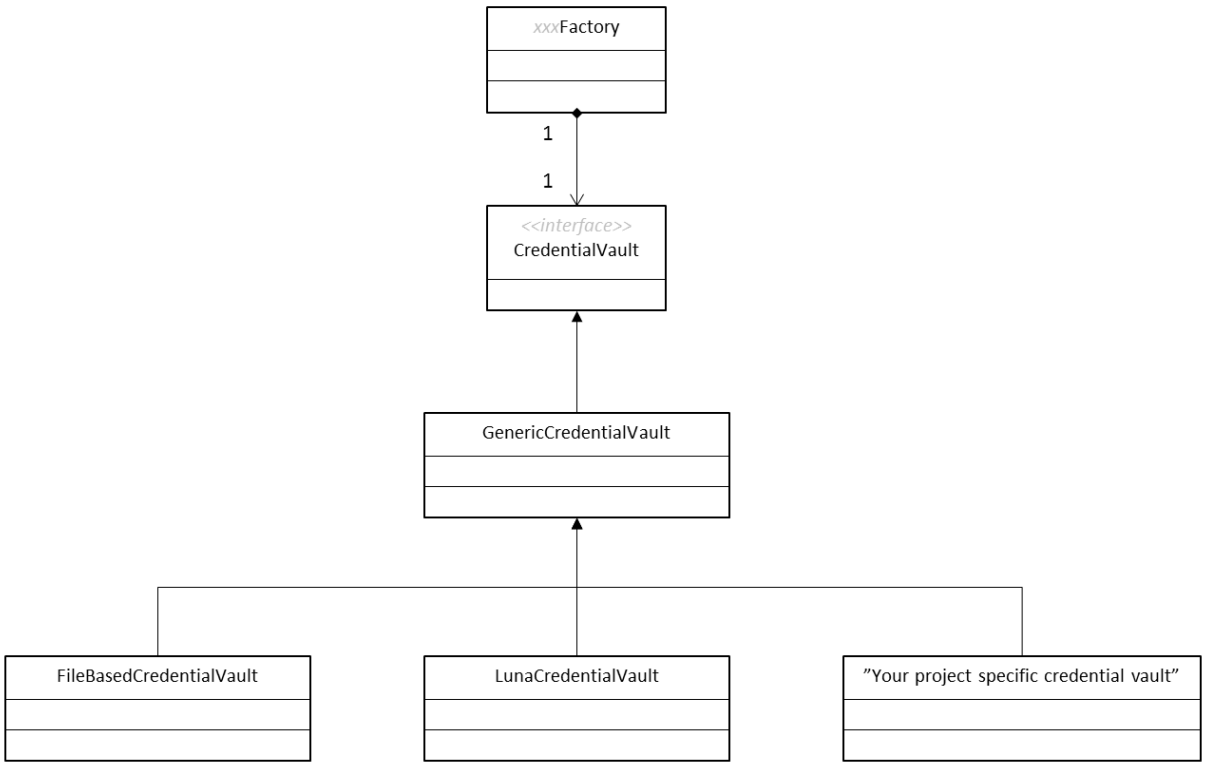


Figure 4 - CredentialVault dependencies

The configuration may further supply the notion of a *Federation* within which the application using the library should operate. The federation is defined in terms of the certification authority issuing the certificates used within the federation, and the identity of the STS. If (and only if) a federation is defined, the library may automatically perform revocation checks when verifying digital signatures on retrieved ID-cards. The library will also be robust against renewal of the STS certificate.

[3] An example of such a file based credential vault can be found in the library

The Federation is the preferred mechanism for establishing trust within SOSI, and users are strongly encouraged to make use of the built-in federations of the SOSI library.

Once the SOSIFactory has been created, it is straightforward to get started. Consider the following code snippet that contains code for constructing a request to send to a service provider:

```
Properties properties = …;
SOSITestFederation testFederation = new SOSITestFederation(properties);
CredentialVault credentialVault = <construct or resolve credentialvault here>;
SOSIFactory factory = new SOSIFactory(testFederation, credentialVault, properties);
Request request = factory.createRequest(
     false, // don't require non-repudiation receipt
     null   // Optional flow-ID (not used here)
);

IDCard idCard = <resolve ID-card here>;

request.setIDCard(idCard);

Element body = <build body DOM element here>;
request.setBody(body);

Document domDocument = request.serialize2DOMDocument();
String xml = XmlUtil.node2String(domDocument, false, true);
<Send xml to Service provider here>
```

The real work for the developer is in the "blue" sections above, i.e.:

- Specifying properties for SOSI, see later for reference on SOSI properties
- Constructing/resolving the credential vault instance
- Constructing/resolving the ID card instance
- Building the body XML
- Sending the xml to the receiver.

In other words; the workload for the developer is greatly reduced compared to a situation where this library was not applicable.

On the "other side" (at the service provider) the library is used as follows:

```
Properties properties = …;
Federation federation = new SOSIFederation(properties);
CredentialVault credentialVault = <construct or resolve credentialvault here>;
SOSIFactory factory = new SOSIFactory(federation, credentialVault, properties);

// This implicitely verifies the STS signature on the ID card etc.
Request request = factory.deserializeRequest(xml);

IDCard idCard = request.getIDCard();

<use ID card attributes for authorization here>

Element body = request.getBody();

<use information in body for business logic here>

Reply reply = factory.createNewReply(
     request.getMessageID(),         // ”In response to” ID
     null                            // Optional flow-ID set to null
);
reply.setIDCard(systemIDCard);

Element replyBody = <build reply body DOM element here>;
reply.setBody(replyBody);

Document domDocument = reply.serialize2DOMDocument();
String replyXML = XmlUtil.node2String(domDocument, false, true);
<Send replyXML to Service provider here>
```

Again the developer more or less only handles things that are related to the business task.

3 Set-up

This chapter describes how to install and set up the SOSI library, i.e. how to download, unpack and configure the library, how to handle dependencies etc.

3.1 Pre-requisites

The library has been tested on Oracle JDK-1.5.0, Oracle SDK-1.6.0 and Oracle SDK-1.7.0

There are the following run-time dependencies:

JDK 1.5.0
Bcmail-jdk15 1.46 Bcprov-jdk15 1.46 Commons-codec 1.5 Commons-httpclient 3.1 Commons-logging 1.1.1 Xalan 2.7.0 Xml security 1.4.5 + some logging library compatible with commons-logging e.g. log4j

Some of the crypto operations (e.g OCES signature verification) need access to crypto algorithms with “unbounded” strength. The JDK’s are shipped with policy files that support strong but not unbounded encryption strength. However Oracle does distribute policy files that allow unbounded encryption strength. You can download or extract **US_export_policy.jar** and **local_policy.jar** from:

Oracle 1.5: http://java.sun.com/javase/downloads/index_jdk5.jsp
Oracle 1.6: http://www.oracle.com/technetwork/java/javase/downloads/index.html
Oracle 1.7: http://www.oracle.com/technetwork/java/javase/downloads/index.html

The files must replace the existing files in $JRE_HOME[4]/lib/security.

3.2 Downloading the library

The library is hosted by the component library facility at Ministry of Science, Technology and Innovation in Denmark. The download area can be found here:
http://www.sosi.dk/twiki/bin/view/ProjectManagement/SOSICompDownload

In the download area you can download a ZIP file containing binaries, test-binaries, sources, documentation, dependency libraries etc.

Seal.Java also contains a number of demos.

[4] Please note that JRE is placed in $JAVA_HOME/jre on Windows platforms.

The demos are useful for development and should not be distributed to production systems. Installing them requires some extra steps in comparison to installing the SEAL component.
The rest of this chapter will concentrate on how to install the SEAL library. Please refer to a later chapter for instructions on how to install the demos.

3.3 A demo installation

Now that the JDK is properly configured you should try following the steps below to make a demo installation. Please consult the SOSI knowledge base (Q/A) if any problems should occur (http://www.sosi.dk/twiki/bin/view/ProjectManagement/SOSIKnowledgeBase):

1. Open the ZIP file and extract to somewhere appropriate on your disk
2. Start a shell (command prompt) and navigate to the bin directory of the installation (sosi/bin)
3. Run the **runtest.sh** (on unix/linux) or **runtests.cmd** (on windows).
4. If you see something like this, your JDK is configured correctly and the library is ready for usage:

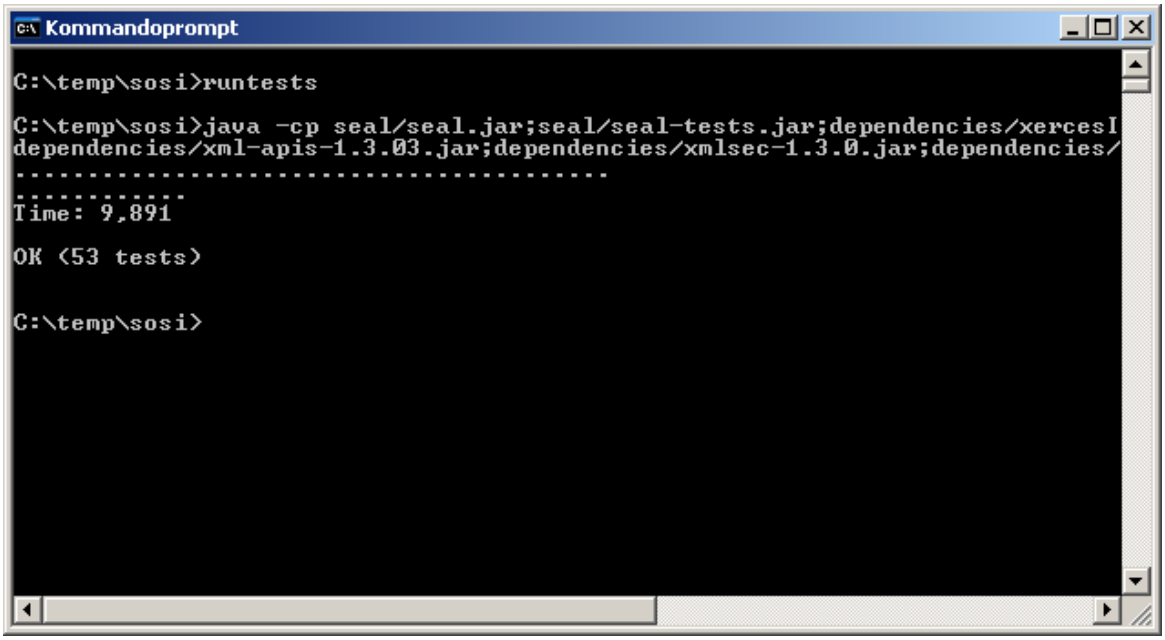


Figure 5 - Expected result from running "runtests.cmd" in a windows shell.

This means that all unittests in the seal module has been executed without any problems.

3.4 Certificates and cryptosystems

The SOSI library does not require a specific JCE crypto provider to run. There are, however, some requirements to the crypto providers used:

- The crypto provider must provide the RSA-SHA1 algorithm.
- If system credentials are stored in #PKCS12 files the crypto provider must be able to read the #PKCS12 format.

- If the built in mechanism for certificate revocation check is used, an external crypto provider supporting “X.509” may be needed.

If you do not have a crypto provider that honors these requirements, one of the most acknowledged free providers is the *Bouncy Castle* crypto provider (http://www.bouncycastle.org/).

Note:
Unfortunately the Seal Tool, a special part of the library used for bootstrapping and renewing credential vaults, uses crypto provider features not standardized by JCE. This results in crypto provider classes being called directly. To decouple this dependency and to enable you to use your favorite crypto provider, the library comes with the `CertificateRequestHandler`, interface and a default BouncyCastle realization. Your own realization can be configured via sosi properties.

As mentioned above the specific way of handling primary keys is not specified in the library. Either you will have to make your own realization of the *CredentialVault* interface or, if your environment allows it, you can use the *FileBasedCredentialVault, RenewableFileBasedCredentialVault* or the *ClasspathCredentialVault*, all of which are supplied with the library. FileBasedCredentialVault reads and writes PKCS entries to the filesystem. RenewableFileBasedCredentialVault is an extension of FileBasedCredentialVault allowing the user to renew VOCES certificates using a webservice, while ClasspathCredentialVault looks for a specific keystore in the classpath. The SOSI library comes with a tool for creating this keystore. Please refer to section 7 for instructions.

If you choose to implement your own credential vault you should take a look at the *FileBasedCredentialVault, RenewableFileBasedCredentialVault, ClasspathCredentialVault,* and *GenericCredentialVault* for inspiration.

A Credential vault can be realized in various ways:

1. Letting a database store the trusted certificates and system credentials.
2. Including the trusted certificates and system credentials in the distribution of the application (EAR, WAR, JAR file).
3. Loading the credentials and certificates once at startup (from a secret file/directory/CD) and storing the credentials and certificates in a cache.
4. If running on an application server the credential vault could integrate to the trust store and credential store on the application server.
5. A “hard coded” class containing the trusted certificates (STS certificate) and system credentials (primary key for this system).

Please note that the use of CredentialVault for storing federation certificates has been deprecated in favor of the Federation mechanism. It is highly recommended to let a CredentialVault store only private certificates and keys when used with federations. As mentioned before the SOSI library also includes the EmptyCredentialVault, which is used when no credential vault is needed. EmptyCredentialVault throws

CredentialVaultException if its methods get called, because this means that you are trying to handle a security level above level 1.

4 How-To (Tutorials)

This chapter describes how to utilize the SOSI-component for communicating using the SOSI protocol. The component supports use cases for the Service Consumer, the STS and the Service Provider, and these are described individually below:

Service Consumer use cases

1. Request an ID-card authentication from a STS
2. Call a Service Provider

STS use cases

3. Issue an ID-card

Service Provider use cases

4. Reply to a service request

Service Consumer use cases (Identity token)

5. Request an Identity token from a STS
6. Call a service provider using an Identity Token

STS use cases (Identity token)

7. Issue an Identity token

Service Provider use cases (Identity token)

8. Retrieve and verify an Identity token.

Service Consumer use cases (OIOSAML assertion to IDCard)

9. Exchange an OIOSAML assertion to an IDCard at a STS

STS use cases (OIOSAML assertion to IDCard)

10. Issue an IDCard based on a OIOSAML assertion

Service Consumer use cases (IDCard to encrypted OIOSAML assertion)

11. Exchange an IDCard to an encrypted OIOSAML assertion at a STS

STS use cases (IDCard to encrypted OIOSAML assertion)

12. Issue an encrypted OIOSAML assertion based on an IDCard

4.1 Setting up properties

The library can be customized with a few properties that are passed to the constructor of the SOSIFactory. Currently the supported properties are:

sosi:validate = {"true", "false"}
Indicates whether or not the DOM parser should validate XMLSchemas for SOSI envelopes. The default value is "true" (i.e. if the property is not specified, the library will validate)

sosi:useDBFCache = {"true", "false"}
Indicates whether or not the DOM parser factory (DocumentBuilderFactory) should be cached or not. The default value is "true" (i.e. if the property is not specified, the factory will get cached).

sosi:issuer = "some String"
The name of the system that is using the library. The value will be inserted into ID-cards when issuing new ID-cards model objects. The default value is "TheSOSILibrary".

sosi:rootschema = "some String"
The root schema file to validate against. The default value is soap.xsd, which is the rootschema for validating the soapheaders. If you want your body validated by the framework you need to define your elements in a new schema and import soap.xsd.
The framework load schema files as resources from the classpath.

sosi:federation.audithandler = "class name"
User provided audit handler to be called, when an audit event arises during a certificate revocation check. Please refer to later section on customizations
The SOSI libray supplies two built-in audithandlers
"dk.sosi.seal.pki.NoAuditEventHandler" which sinks all events, which is the default implementation
"dk.sosi.seal.pki.CommonsLoggingAuditEventHandler" which logs using apache commons logging.

sosi:cryptoprovider.x509 = "provider name"
This enables you to change the provider used for handling x509 certificated. The default value is "BC" (Bouncy Castle).

sosi:cryptoprovider.pkcs12 = "provider name"
This enables you to change the provider used for handling PKCS12. Only used if credentials are stored in pkcs#12 format. The default value is "BC" (Bouncy Castle).

sosi:cryptoprovider.rsa = "provider name"
This enables you to change the provider used for handling RSA. The default value is "BC" (Bouncy Castle).

sosi:cryptoprovider.sha1withrsa = "provider name"
This enables you to change the provider used for handling SHA1withRSA. The default value is "BC" (Bouncy Castle).

As a convenience method dk.sosi.seal.model.SignatureUtil has the following method

```
public static Properties setupCryptoProviderForJVM() {
Properties properties = new Properties();
if("IBM Corporation".equals(System.getProperty("java.vm.vendor"))) {
  properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_PKCS12, "BC");
  properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_X509, "BC");
  properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_RSA, "IBMJCE");
  properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_SHA1WITHRSA, "IBMJCE");
} else { // else SUN
  if ("1.4".equals(System.getProperty("java.specification.version"))) {
    properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_PKCS12, "BC");
  } else { // else 1.5+
    properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_PKCS12, "SunJSSE");
  }

  if ("1.6".equals(System.getProperty("java.specification.version"))) {
    properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_X509, "BC");
  } else if ("1.4".equals(System.getProperty("java.specification.version"))) {
    properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_X509, "BC");
  } else { // else 1.5
    properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_X509, "SUN");
  }

  properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_RSA, "SunRsaSign");
  properties.put(SOSIFactory.PROPERTYNAME_SOSI_CRYPTOPROVIDER_SHA1WITHRSA, "SunRsaSign");
}
return properties;
}
```

As indicated the IBM JVM will need additional security providers (like bouncycastle) if systems credentials are stored in pkcs#12 format or the built in certificate revocation mechanism is used.

For Sun JVM 1.4 external providers are only needed for if credentials are stored in pkcs#12 format. It is recommended to store credentials in java keystore format.

4.2 Sequence Diagrams

Figure 6 illustrates the major steps needed by all participants in requesting an ID-card for authentication at level 3 i.e. with a VOCES system signature. As such, it shows the steps needed for use cases 1 and 3 above:

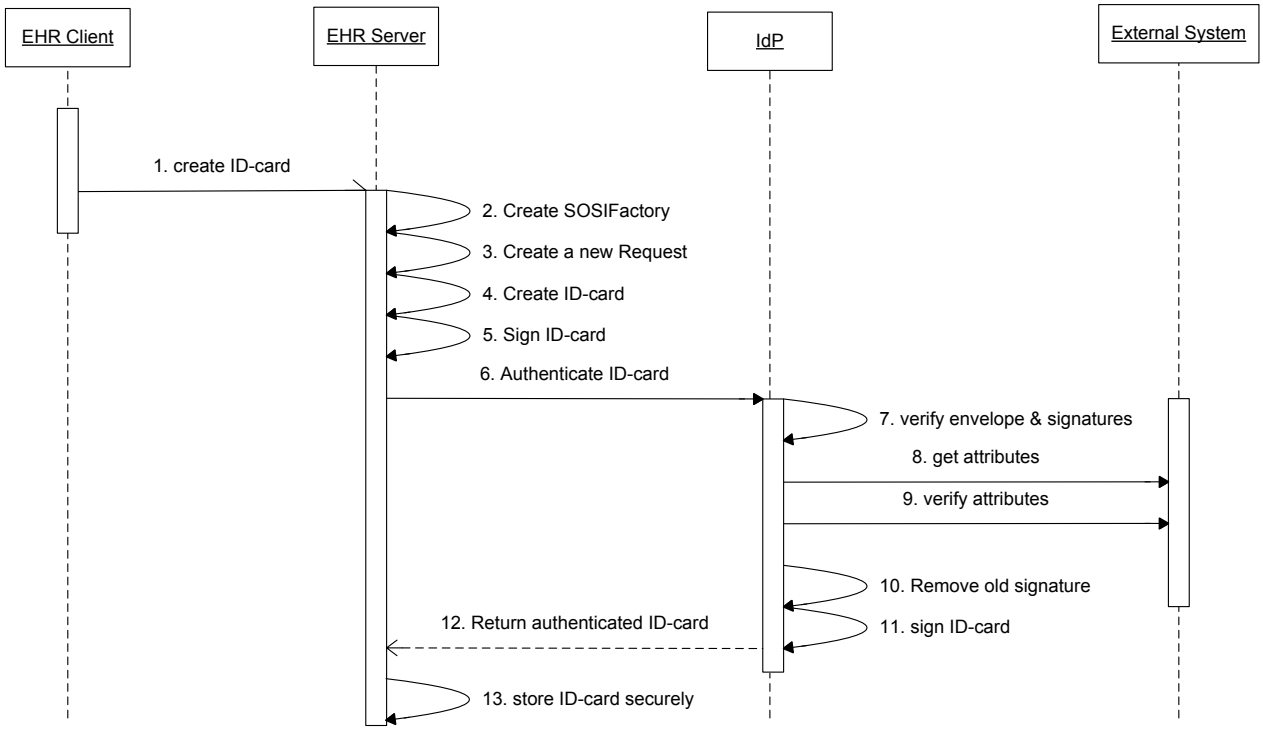


Figure 6 - Sequence diagram for requesting an ID-card.

When an ID-card is to be authenticated at level 4, i.e. with a MOCES signature, the flow is similar, with just a few differences. Figure 7 illustrates this flow for use cases 1 and 3, respectively:

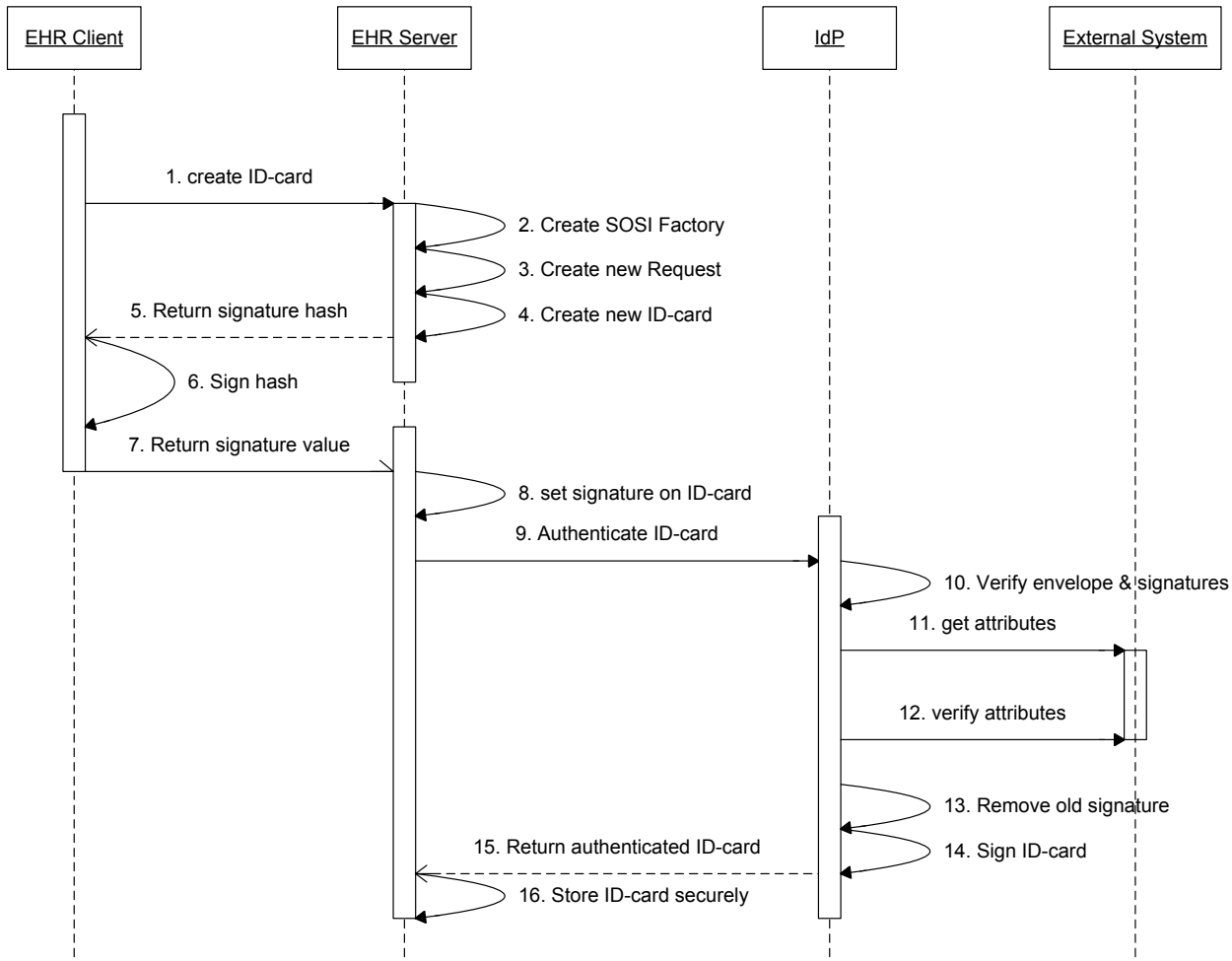


Figure 7 - Sequence diagram for use cases 1 and 3.

Service execution is performed in the same manner regardless of the authentication level on the ID-card.

Figure 8 illustrates a service call with all participants, and hence covers use cases 2 and 4:

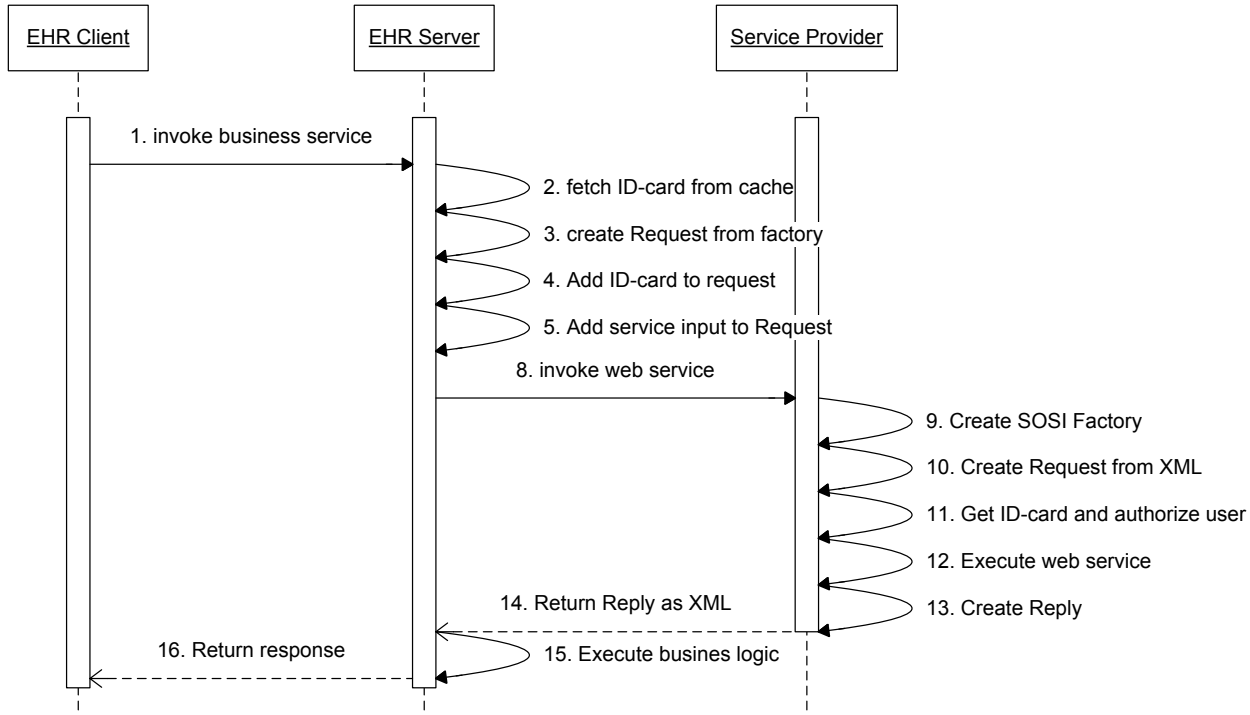


Figure 8 - Sequence diagram illustrating use cases 2 and 4.

4.3 Use case 1: How to authenticate an ID-card at an STS

The steps needed in creating a request for authenticating an ID-card are listed below:

1. Create SOSIFactory
2. Create a Request
3. Create an ID-card
4. Build the XML representation
5. Sign the ID-card
6. Send request to the STS
7. Extract ID-card for use

4.3.1 Create SOSIFactory

All model objects in SOSI are created through the SOSIFactory, which must be initialized before creating model objects:

```
Properties properties = new Properties();
properties.setProperty("sosi:issuer", "XXXX");
// possibly specify other properties here....
Federation federation = new SOSIFederation(properties);
CredentialVault cv = new ClasspathCredentialVault(KEYSTORE_PASSWORD);
SOSIFactory factory = new SOSIFactory(federation, credentialVault, properties);
```

Please note that the actual type of `CredentialVault` may vary depending on the type of environment you need to run the application in. The `ClassPathCredentialVault` may fit perfectly for some situations. In other situations you may have to develop a new `CredentialVault` realization that retrieves keys and certificates from some other persistent store (e.g. a database). If so you should consider subclassing `GenericCredentialVault`.

In this example we instantiate a `ClasspathCredentialVault`. This type of credential vault is a read-only credential vault that reads keys and certificates from a special keystore that must be found on the class path. Use the SEAL tool to generate a JAR file containing this special keystore (see chapter 7).

The SOSI library has 2 built-in Federations

```
Properties properties = new Properties();
properties.setProperty("sosi:issuer", "XXXX");

Federation testFederation = new SOSITestFederation(properties);
// or
SOSIFederation federation = new SOSIFederation(properties);
```

4.3.2 Create a Request

Use the SOSIFactory's createNewSecurityTokenRequest method to create a security token request object.

```
// Create a simple security token request object
SecurityTokenRequest request = factory.createNewSecurityTokenRequest();
```

4.3.3 Create an ID-card

Again use the SOSIFactory for creating the ID-cards. SOSI supports both SystemIDCards and UserIDCards. The difference between the two types of ID-cards is that SystemIDCards only contain information identifying the client system, while UserIDCards also contain information for identifying the client user.

According to the DGWS specification ID-cards must expire after at most 24 hours. However, to prevent problems from server clocks not being synchronized, the ID-cards issued by the SOSI library are displaced 5 minutes so that the "start time" of the ID-Card is "now" minus 5 minutes and the expiration time is "now" plus 23:55.

A SystemIDCard is created by providing information on the name of the system, a CareProvider, the authentication level and the certificate used to sign the ID-card:

```
// Create a system ID-card
IDCard card = factory.createNewSystemIDCard(
     "SOSITEST",
     new CareProvider(SubjectIdentifierTypeValues.SKS_CODE, "1234", "sosi"),
     AuthenticationLevel.VOCES_TRUSTED_SYSTEM,
     factory.getCredentialVault().getSystemCredentialPair().getCertificate()
);
```

To create a UserIDCard the above mentioned information must be complemented by user data, authorization code and whether the ID-card should be signed by a VOCES or a MOCES certificate:

```
// create a new User ID-card
IDCard card = factory.createNewUserIDCard(
    systemName,
    cpr,
    givenName,
    surName,
    email,
    "surgeon",
    UserRole.DOCTOR,
    new CareProvider(CareProvider.CAREPROVIDER_TYPE_CVR, orgCVR, orgName),
    authorizationCode,
    AuthenticationLevel.MOCES_TRUSTED_USER,
    factory.getCredentialVault().getSystemCredentialPair().getCertificate())
  )
);
```

There are several embedded objects in user ID cards that defines the context this user is acting in:

- **SystemName** – the name of the system that this user was using when this ID-card was issued.
- **UserRole** – indicates which role the user has when using this ID-card. Presently only "Doctor" or "Nurse" are valid values. The set of valid user roles will most probably be extended in future releases.
- **CareProvider** – an object representing the organizational unit that the user is acting for.

The **AuthenticationLevel** object defines the level of trust another system can have to this ID-card. Presently three of the five levels in "Den Gode Webservice" are supported:

- **NO_AUTHENTICATION** – the user does not need to present credentials (DGWS level 1).
- **VOCES_TRUSTED_SYSTEM** – the user is implicitly trusted through the system that the user is using (DGWS level 3).
- **MOCES_TRUSTED_USER** – the user will present a digital signature based on a employee certificate (DGWS level 4).

Note: When issuing ID-cards with VOCES authentication level, the digital signature and certificate will automatically be injected into the XML when serializing the message into XML.

The last parameter is the certificate that can be used to validate the signature. The hash is used for correlating an optional signature on the message (DGWS level 5) with the certificate that was used when issuing the ID-card. This enables service providers to check a message signature without having to check the embedded certificate for revocation etc. If NO_AUTHENTICATION is used you can parse a null value to the certificate parameter.

4.3.4 Assign the ID-card to the request

Next assign the ID-card to the request

```
// assign ID-card to request
request.setIDCard(card);
```

4.3.5 Build the XML representation

The SOSIFactory allows for the construction of entire SOAP messages, complete with SOSI specific headers and a custom body. Developers deal with Request and Response objects that represent SOSI specific SOAP messages. When a request or response message is finished, it can be turned into a DOM document[5]:

```
// build request document
Document requestDocument = request.serialize2DOMDocument();
```

4.3.6 Signing the ID-card

As mentioned above, the SOSI library deals with two different types of signature models. When the ID-card is signed with a VOCES signature, the entire process can be handled programmatically with no user involvement. When this is the case, the SOSI library will look for the VOCES key pair in the supplied CredentialVault implementation by doing a `getSystemCredentialPair()`. The actual signature generation is performed at the time the DOM representation of the ID-card is requested i.e. when `serialize2DOMDocument()` is called.

When the ID-card represents a user and is to be signed with a MOCES signature, the process is slightly different. In this case, it is necessary, as per the OCES certificate policy, to prompt the end-user for a password. Therefore Seal cannot generate the signature completely, but will have to rely on some external mechanism to perform the last few steps.

To make a UserIDCard with a MOCES signature, the following steps must be performed:

1) Create a new UserIDCard using `SOSIFactory`
2) Get the bytes to be signed from the `UserIDCard`. These are the bytes from the `<SignedInfo>` element.

[5] Due to limitations in current java.seal implementation, custom tags and attributes in soap header will be stripped during the transformation to and from XML.

3) Use some external mechanism to digest and encrypt the bytes using an RSA key from a MOCES certificate.
4) Inject the signature value into the UserIDCard.

The code sample below illustrates the steps programmatically:

```
// get bytes for signing
byte[] siBytes = request.getIDCard().getBytesForSigning(
    requestDocument,
    factory.getCredentialVault().getSystemCredentialPair().getCertificate()
);

String signature =
  <Send bytes to external signing here. Should return a Base-64 encoded String>

// This will insert the signature into the IDCard.
request.getIDCard().injectSignature(signature);
```

4.3.7 Send request to Security Token Service

Any framework like Apache Axis etc. can be used to call the Security Token Service (STS). The DOM document can be used either directly or in a serialized form. The SOSI library provides the `XmlUtil` class for the purpose of transforming XML documents between DOM and string representations.

Here we show how to call a web service endpoint with a simple URL connection:

```
Writer wout = null;
try {
   URL u = new URL(<server URL>);
   URLConnection uc = u.openConnection();
   HttpURLConnection connection = (HttpURLConnection) uc;
   connection.setDoOutput(true);
   connection.setDoInput(true);
   connection.setRequestMethod("POST");
   connection.setRequestProperty("SOAPAction", SOAP_ACTION);

   OutputStream out = connection.getOutputStream();
   Writer wout = new OutputStreamWriter(out);
   String xml =
      XmlUtil.node2String(
         doc,
         false, // Do not pretty print XML
         true   // Include <?xml version="1.0" encoding="UTF-8"?> in XML
      );

   wout.write(xml);
   wout.flush();
   // Get the response
   InputStream in = connection.getInputStream();
   BufferedReader reader = new BufferedReader(new InputStreamReader(in));
   String line;
   String xmlResponse = "";
   while ((line = reader.readLine()) != null) {
      xmlResponse += line;
   }
   <deserialize reply … see below>
} catch (…) {
   <handle exceptions>
} finally {
   <finish up (close streams etc.)>
}
```

Alternatively, if the service you need to invoke uses SSL, the SOSI library offers a HTTPS helper utility, that also facilitates server trust to be set up programmatically:

```
// setup https helper class.
HttpsConnector helper =
new HttpsConnectorImpl(
       TrustedServerCertificateIssuers.getTrustedServerCertificateIssuers()
);

//you may need to add additional trusted server certificate issuers

String xml =
       XmlUtil.node2String(
          doc,
          false, // Do not pretty print XML
          true      // Include <?xml version="1.0" encoding="UTF-8"?> in XML
    );

String reply = helper.postSOAP(xml, new URL(<server URL>));

<deserialize reply … see below>
```

The SOSI library uses its own server certificate trust mechanism, which does not depend on the global JRE settings. Depending on the issuer of the server certificate used by the service, you may need to pass a different trusted certificate to the constructor above.

4.3.8 Extract ID-card for use

When the response is returned from the STS, the SOSIFactory can be used to deserialize the entire SOAP envelope into a SecurityTokenResponse object. The client can now extract the ID-card using `reply.getIDCard()` and store it for later use with service providers as explained in the next section.

```
// build object representation of the response
SecurityTokenResponse resp = factory.deserializeSecurityTokenResponse(xmlResponse);
if (FlowStatusValues.FLOW_FINALIZED_SUCCESFULLY.equals(resp.getFlowStatus()) {
    ID-card idCard = resp.getIDCard();
} else {
    <Handle SOAP fault here>
}
```

4.4 Use case 2: How to call a Service Provider

The steps for calling a Service Provider are very similar to those required when calling an STS:

1. Create SOSIFactory
2. Create a Request
3. Add the ID-card
4. Build the XML representation
5. Send request to the Service Provider
6. Extract the service reply from the Reply object

Steps 1, 2, 4, and 5 are exactly the same as for the ID-card request and therefore omitted here.

4.4.1 Add the ID-card

ID-cards have a maximum validity of 24 hours[6] and can therefore be cached and reused for a number of requests to different Service Providers. This is in fact the mechanism that provides the single sign on. To verify if an ID-card is valid in time, you can use the `isValidInTime()` method. If this method returns `false`, the user must be re-authenticated and a new ID-card must be issued. Valid ID-cards can be added to requests by calling the `setIDCard(…)` method.

```
// add ID-card to request
if (!idCard.isValidInTime()) {
    idCard = <request ID-card from IdP as described above>
}
request.setIDCard(idCard);
```

4.4.2 Extract the service reply from the Reply object

Handling the reply from a Service Provider is equally simple as getting the ID-card from the STS. The body of the reply is simply extracted by calling `getBody()` on the reply object:

```
// build object representation of reply
Reply reply = factory.deserializeReply(xmlString);
Node body = reply.getBody();

<handle the body here>
```

4.5 Use case 3: How to issue an ID-card (STS functionality)

The STS can also take advantage of the SOSI library. The steps would be:

1. Create SOSIFactory
2. Create the Request from XML
3. Verify the information in the ID-card
4. Sign the ID-card
5. Send Reply

Step one is similar to section 4.3.1 and is therefore omitted here.

[6] Please note that some service providers may choose to reject ID-cards that are older than a certain limit e.g. 8 hours due to a more strict security policy. In case this happens, the client must reissue an ID-card and reauthenticate with the STS.

4.5.1 How to create the Request from XML

The SOAP request is deserialized into seal Request object by the `deserializeRequest()` factory method:

```
// build object representation of request
Request request = factory.deserializeRequest(xmlString);
…
```

The operation above also verifies the signature on the ID-card, and if it is not valid, the `deserializeRequest()` will throw a `SignatureInvalidModelBuildException`.

4.5.2 How to verify the information in the ID-card

Once the request has been deserialized, the ID-card can be retrieved and its attributes verified. The actual verification process is outside the scope of the SOSI library:

```
// verify ID-card information
IDCard idCard = request.getIDCard();
if (idCard instanceof UserIDCard) {
    UserInfo info = ((UserIDCard) idCard).getUserInfo();
    String cpr = info.getCPR();
    String givenName = info.getGivenName();
    String surName = info.getSurName();
    String email = info.getEmail();
    String authorizationCode = info.getAuthorizationCode();
    UserRole role = info.getRole()
    // Verify the user information
    …
}
```

4.5.3 How to sign the ID-card

If all goes well, and the ID-card verifies, a new ID-card must be issued containing much of the data from the original ID-card sent from the requestor and possibly some new attributes resolved from external systems. For this situation the SOSI library supplies a special method, `copyToVOCESSignedIDCard()`, on `SOSIFactory`. This method will copy all attributes from a requested ID card into a new VOCES signed ID card, preserving what is necessary from the original ID card (authentication level, certificate hash code). If the original ID card had no certificate hash code this will be generated. Since the new ID-card is VOCES signed, it will implicitly be signed with the STS key when serialized to DOM:

```
// build object representation of request
IDCard stsSignedIDCard = factory.copyToVOCESSignedIDCard requestedIDCard);

Reply reply = factory.createNewReply(
    request.getMessageID(),
    request.getFlowID(),
    null
);

Reply.setIDCard(stsSignedIDCard);
Document document = reply.serialize2DOMDocument();
```

4.5.4 How to send the reply

The reply object can now be serialized to XML and returned directly following the addition of the STS response body. In SOSI, this would be a SAML Authentication Response message, where the returned ID card is in the SOAP header.

4.6 Use case 4: How to reply to a service request

Replying to a service request is just the more general case of issuing ID-cards. The steps are similar to those described in section 4.5:

1. Create SOSIFactory
2. Create the Request from XML (incl. verification of validity the ID-card)
3. Do stuff
4. Send Reply

The only difference is step 3, where the service does what it is supposed to do (like the STS is supposed to issue ID-cards).

4.7 Use case 5: Request an Identity token from a STS

The following use case goes through the process of requesting an Identity token from a STS.
The use case assumes that the user already has a valid IDCard, see 4.3.

4.7.1 Create IDWSHFactory

All model objects in IDWSH are created through the IDWSHFactory, which must be initialized before creating model objects:

```
Properties properties = new Properties();
// possibly specify other properties here....
Federation federation = new SOSIFederation(properties);

CredentialVault cv = new ClasspathCredentialVault(properties, KEYSTORE_PASSWORD);

IDWSHFactory factory = new IDWSHFactory(federation, credentialVault);
```

Please note that the actual type of `CredentialVault` may vary depending on the type of environment you need to run the application in, see 4.3.1.

4.7.2 Create a request.

Use the IDWSHFactory to create an `IdentityTokenRequestDOMBuilder`. The `IdentityTokenRequestDOMBuilder` takes a number of arguments:

- The IDCard – used for extracting information about the user.
- The expected audience.
- The end address.

```
IdentityTokenRequestDOMBuilder itrdb =
factory.createIdentityTokenRequestDOMBuilder();

itrdb.setAudience(…);
itrdb.setIdCard(…);
itrdb.setWSAddressingTo(…)

Document document = itrdb.build();
```

The resulting document can now be transmitted to a STS.

4.7.3 Retrieve the Identity token.

Once the request has been handled by the STS, a response is returned. To handle this response, and extract the embedded Identity token, the IDWSHFactory includes the `IdentityTokenResponseModelBuilder` class.

```
IdentityTokenResponseModelBuilder itrdb = factory.
createIdentityTokenResponseModelBuilder ();

IdentityTokenResponse itr = itrdb.build(documentFromSTS);

IdentityToken identityToken = itr.getIdentityToken();
```

Hereafter the Identity token can be used for calling a supported service provider.

4.8 Use case 6: Call a service provider using an Identity token.

The Identity token can be utilized in two ways, either directly in a SOAP message or in a serialized form, which can be used as part of an URL.
Utilizing the Identity token within a normal SOAP message, follows standard pattern, and will not be further handled in this document.
Utilizing the Identity token as part of an URL requires the following steps.

```
IdentityToken identityToken = itr.getIdentityToken();

String urlToken = identityToken. createURLBuilder().encode();
```

This operation returns a GZipped, Base64 encoded URL safe string.

NOTE: There is a big difference in the number of characters allowed in the URL bar by the different browsers. Currently Internet Explorer sets the bar, by only allowing 2000 characters. URL containing more than 2000 characters will therefore fail with unexpected exceptions.

4.9 Use case 7: Issue an Identity token.

The following use case takes you through the process of retrieving an Identity token request, creating an Identity token and creating an Identity token response.

4.9.1 Retrieve the Identity token request.

Once the STS receive a request, the request document can be transformed into an IdentityTokenRequest through the IDWSHFactory.

```
IdentityTokenRequestModelBuilder itrmb = factory.
createIdentityTokenRequestModelBuilder();

IdentityTokenRequest itr = itrdb.buildModel(documentFromClient);
```

4.9.2 Creating an Identity token

Based on the information in the request the STS can now create an Identity token.

```
IdentityTokenBuilder itb = factory.createIdentityTokenBuilder();
itb.setXXX(…);
itb.setXXX(…);
itb.setXXX(…);
itb.setXXX(…);
itb.setXXX(…);
```

Please note, that the build() method is not called on the IdentityTokenBuilder. The IdentityTokenResponseDOMBuilder handles this.

4.9.3 Creating an Identity token response

Once the `IdentityTokenBuilder` has been populated, a response can be created using the IDWSHFactory.

```
IdentityTokenResponseDOMBuilder itrdb = factory.
createIdentityTokenResponseDOMBuilder();

itrdb.setIdentityToken(…);
itrdb.setXXX(…);
itrdb.setXXX(…);
itrdb.setXXX(…);

IdentityTokenResponse itr = itrdb.build();
```

Calling `build()` on the `IdentityTokenResponseDOMBuilder` instance automatically redirects the call to the `build()` method on the `IdentityTokenBuilder`.

4.10 Use case 8: Retrieve and use an Identity token.

Once the client has received a response from a STS, the identity token can be extracted using the IDWSHFactory.

```
IdentityTokenResponseModelBuilder itrmb = factory.
createIdentityTokenResponseModelBuilder();

IdentityTokenResponse itr = itrdb.build(…);
```

Once the identity token has been extracted, it is now ready for use.
The identity token can be used either as part of a SOAP message, or as an HTTP get parameter.
Using the identity token in a message is straight forward – however using the identity token as part of an http request is new.

```
IdentityTokenResponse itr = …;

IdentityTokenURLBuilder itub = itr.createURLBuilder();

String urlEncoded = itub.encode();
```

The resulting string is a GZipped BASE64 representation of the identity token.
The BASE64 is URL safe, and is encoded using a modified BASE64 encoding algorithm ('base64url' encoding).

4.11 Use case 9: Exchange an OIOSAML assertion to an IDCard at a STS

The following steps illustrate how an OIOSAML assertion that is received from an IdP can be exchanged to a SOSI IDCard at an STS.

4.11.1 Create an OIOSAMLFactory

All OIOSAML model objects are created through the OIOSAMLFactory:

```
OIOSAMLFactory factory = new OIOSAMLFactory(credentialVault);
```

4.11.2 Create an OIOSAMLAssertionToIDCardRequest

Use the OIOSAMLFactory to create an `OIOSAMLAssertionToIDCardRequestDOMBuilder`. The `OIOSAMLAssertionToIDCardRequestDOMBuilder` takes a number of arguments:

- The OIOSAML assertion for the user
- The name of the clients IT-System
- A CredentialVault used to sign the request

```
Document doc = < DOM parsed OIOSAMLAssertion from an IdP >;
CredentialVault vault = < CredentialVault containing signing key pair >;
OIOSAMLAssertion assertion = new OIOSAMLAssertion(doc.getDocumentElement());

OIOSAMLAssertionToIDCardRequestDOMBuilder domBuilder =
            factory.createOIOSAMLAssertionToIDCardRequestDOMBuilder();
domBuilder.setOIOSAMLAssertion(assertion);
domBuilder.setITSystemName("Harmoni/EMS");
domBuilder.setSigningVault(vault);
domBuilder.setXXXXX(…);
domBuilder.setXXXXX(…);
  . . .
```

The resulting document can now be transmitted to an STS.

4.11.3 Parse an OIOSAMLAssertionToIDCardResponse

Once the request has been handled by the STS, a response is returned. To handle this response, and extract the embedded IDCard, the OIOSAMLFactory includes the `OIOSAMLAssertionToIDCardResponseModelBuilder` class.

```
Document responseDoc = < DOM parsed response from an STS >;
OIOSAMLAssertionToIDCardResponse assertionToIDCardResponse =
      factory.createOIOSAMLAssertionToIDCardResponseModelBuilder().build(responseDoc);

IDCard idcard = assertionToIDCardResponse.getIDCard();
```

Hereafter the IDCard can be used for calling a regular DGWS service.

4.12 Use case 10: Issue an IDCard based on a OIOSAML assertion

The following use case takes you through the process of parsing an OIOSAML to IDCard request and creating an OIOSAML to IDCard response.

4.12.1 Parse an OIOSAMLAssertionToIDCardRequest

Once the STS has received a request, the request document can be parsed into an OIOSAMLAssertionToIDCardRequest through the OIOSAMLFactory.

```
Document requestDoc = < DOM parsed client request >;

OIOSAMLAssertionToIDCardRequest assertionToIDCardRequest =
     factory.createOIOSAMLAssertionToIDCardRequestModelBuilder().build(requestDoc);

assertionToIDCardRequest.validateSignature();
X509Certificate cert = assertionToIDCardRequest.getSigningCertificate();
. . .
OIOSAMLAssertion assertion = assertionToIDCardRequest.getOIOSAMLAssertion();
assertion.validateSignatureAndTrust(<CredentialVault containg IdP certificate>);
```

4.12.2 Create an OIOSAMLAssertionToIDCardResponse

Once the STS has issued an IDCard based upon the received OIOSAML assertion, a response can be created through the OIOSAMLFactory.

```
OIOSAMLAssertionToIDCardResponseDOMBuilder domBuilder =
     factory.createOIOSAMLAssertionToIDCardResponseDOMBuilder();

IDCard idCard = <issued IDCard>;
domBuilder.setIDCard(idCard);
domBuilder.setSigningVault(stsVault);
Document responseDoc = domBuilder.build();
```

4.13 Use case 11: Exchange an IDCard to an encrypted OIOSAML assertion at a STS

The following steps illustrate how an STS-signed IDCard can be exchanged to an encrypted OIOSAML assertion at an STS. The encrypted assertion can then be used to start a web browser session by posting the assertion in unsolicited SAML response to the web application for which the assertion was issued.

4.13.1 Create an OIOSAMLFactory

All OIOSAML model objects are created through the OIOSAMLFactory:

```
OIOSAMLFactory factory = new OIOSAMLFactory();
```

4.13.2 Create an IDCardToOIOSAMLAssertionRequest

Use the OIOSAMLFactory to create an `IDCardToOIOSAMLAssertionRequestDOMBuilder`. The `IDCardToOIOSAMLAssertionRequestDOMBuilder` takes a number of arguments:

- The STS-signed IDCard for the user
- The ‘audience’ of the OIOSAML assertion, that is the web application

```
UserIDCard idcard = < STS-signed UserIDCard>;

IDCardToOIOSAMLAssertionRequestDOMBuilder domBuilder =
            factory.createIDCardToOIOSAMLAssertionRequestDOMBuilder();
domBuilder.setUserIDCard(idcard);
domBuilder.setAudience("http://sundhed.dk");
domBuilder.setXXXXX(…);
domBuilder.setXXXXX(…);
  . . .
Document requestDoc = domBuilder.build();
```

The resulting document can now be transmitted to an STS.

4.13.3 Parse an IDCardToOIOSAMLAssertionResponse

Once the request has been handled by the STS, a response is returned. To handle this response, and extract the embedded encrypted OIOSAML assertion, the OIOSAMLFactory includes the `IDCardToOIOSAMLAssertionResponseModelBuilder` class.

```
Document responseDoc = < DOM parsed response from an STS >;
IDCardToOIOSAMLAssertionResponse response =
      factory.createIDCardToOIOSAMLAssertionResponseModelBuilder().build(responseDoc);

Element encryptedAssertion = response.getEncryptedOIOSAMLAssertionElement();
```

Hereafter the encrypted OIOSAML assertion can be used for generating an unsolicited SAML response to start a web browser session.

4.14 Use case 12: Issue an encrypted OIOSAML assertion based on an IDCard

The following use case takes you through the process of parsing an IDCard to OIOSAML assertion request and creating an IDCard to OIOSAML assertion response.

4.14.1 Parse an IDCardToOIOSAMLAssertionRequest

Once the STS has received a request, the request document can be parsed into an IDCardToOIOSAMLAssertionRequest through the OIOSAMLFactory.

```
Document requestDoc = < DOM parsed client request >;

IDCardToOIOSAMLAssertionRequest request =
     factory.createIDCardToOIOSAMLAssertionRequestModelBuilder().build(requestDoc);

request.validateSignature();
X509Certificate cert = request.getSigningCertificate();
. . .
UserIDCard idcard = request.getUserIDCard();
idcard.validateSignatureAndTrust(<CredentialVault containg IdP certificate>);
```

4.14.2 Create an OIOSAMLAssertionToIDCardResponse

Once the STS has issued an OIOSAML assertion based upon the received IDCard, a response can be created through the OIOSAMLFactory.

```
IDCardToOIOSAMLAssertionResponseDOMBuilder domBuilder =
    factory.createIDCardToOIOSAMLAssertionResponseDOMBuilder();

OIOSAMLAssertion assertion = <issued OIOSAML assertion>;
domBuilder.setOIOSAMLAssertion(assertion);
domBuilder.setEncryptionKey( < public key for audience >);
domBuilder.setSigningVault( < STS vault> );
Document responseDoc = domBuilder.build();
```

5 Customizing the SOSI library

As it is envisaged that the SOSI library will be used in many different scenarios and environments the library has been prepared for customizations. However, in most situations the default implementations will be suitable – and this chapter is only relevant if you are specifically aware of any special needs for your application.

5.1 Audit logging

The SOSI library defines a number of audit events suitable for logging:

5.1.1 Informational events

```
public static final String EVENT_TYPE_INFO_FEDERATION_INITIALIZED
public static final String EVENT_TYPE_INFO_CREDENTIAL_VAULT_INITIALIZED
public static final String EVENT_TYPE_INFO_SOSI_XML_VALIDATED
public static final String EVENT_TYPE_INFO_CERTIFICATE_VALIDATED
public static final String EVENT_TYPE_INFO_FULL_CRL_DOWNLOADED
```

5.1.2 Warning events

```
public static final String EVENT_TYPE_WARNING_NO_REVOCATION_CHECK
```

5.1.3 Error events

```
public static final String EVENT_TYPE_ERROR_DOWNLOADING_FULL_CRL
public static final String EVENT_TYPE_ERROR_FULL_CRL_EXPIRED
public static final String EVENT_TYPE_ERROR_PARSING_SOSI_XML
public static final String EVENT_TYPE_ERROR_VALIDATING_SOSI_MESSAGE
public static final String EVENT_TYPE_ERROR_VALIDATING_STS_CERTIFICATE
public static final String EVENT_TYPE_ERROR_VALIDATING_CERTIFICATE
```

By implementing the associated interface `dk.sosi.seal.pki.AuditEventHandler` the logging events can be handled by this customized class by setting the `sosi:federation.audithandler` property defined in `dk.sosi.seal.SOSIFactory` to the full qualified name of the custom AuditHandler.

The default implementation uses apache commons-logging, more specifically under the following logger:

```
private static final String AUDIT_LOGGER_NAME = "dk.sosi.seal.AUDIT";
private static final Log log = LogFactory.getLog(AUDIT_LOGGER_NAME);
```

If the user application uses log4j the apache commons-logging implementation automatically detects that, and the desired log behavior of the SOSI library can be controlled by inserting the following in your log4j.xml settings file

```
<category name="dk.sosi.seal.AUDIT">
    <priority value="warn" />
    <appender-ref ref="STDOUT" />
</category>
```

If no logging from the SOSI library is preferred the `"sosi:federation.audithandler"` property can be set to the convenience class `"dk.sosi.seal.pki.CommonsLoggingAuditEventHandler"`

The default implementation is: `"dk.sosi.seal.pki.NoAuditEventHandler"`

5.2 Revocation Control

Revocation control in SOSI is primarily concerned with checking the revocation status of the STS certificate. This is a core functionality built into the Federation that the system is running under.

By specifying `"sosi:certificate.checker"` as an implementation of interface "dk.sosi.seal.pki.CertificateStatusChecker" the client application can control how to perform certificate revocation control.

The default implementation of "`dk.sosi.seal.pki.NaiveCertificateStatusChecker`" does not support certificate revocation control.

Another built-in mechanism "dk.sosi.seal.pki.FullCRLCertificateStatusChecker" is based on downloading the full CRL, which is done when the actual Federation is constructed. If the user application does nothing the CRL will eventually expire causing an event

```
public static final String EVENT_TYPE_ERROR_FULL_CRL_EXPIRED
```

Under the FullCRLCertificateStatusChecker mechanism it is the responsibility of the user application to periodically call:

```
federation.getCertificationAuthority().refreshCRL();
```

This will cause a synchronous download of the CRL – not interfering with or delaying any other operations of the SOSI library. In a production environment a full download will amount to 2-10 seconds and approximately 4-5 MB of bandwidth (December 2006).

If the default mechanism is not suitable, the CRL mechanism can be customized in the 3 following ways – with the escalating responsibility for the Federation taken by the customizing application.

5.2.1 IntermediateCertificateCache

The IntermediateCertificateCache is passed to the constructor of the Federation. The user application can also define a custom implementation of the IntermediateCertificateCache interface or use default implementation (`HashMapCertificateCache`) supplied with seal.

5.2.2 CertificateStatusChecker

The CertificateStatusChecker is passed to the constructor of the Federation.
The user application can also define a custom implementation of the CertificateStatusChecker interface

This allows for implementing a custom CertificateStatusChecker using partitioned CRL's, databases or other mechanisms.

The CertificationAuthority is used for validating whether any downloaded CRL's is issued by the expected CA.

5.2.3 Federation

By implementing a custom Federation the SOSIFactory object can be instantiated using the custom Federation.
A custom Federation can be implemented extending one of the builtin Federation of the SOSI library:

```
public class SOSIFederation extends Federation {
public class SOSITestFederation extends Federation {
```

This allows for full flexibility in implementing the behavior of the Federation as desired.

6 How to install the demos

The Seal library ships with some demo applications that illustrate its use.

6.1 AXIS based demo

The use of the SOSI library is illustrated via the official Test-STS webservice (http://pan.certifikat.dk/sts/services/SecurityTokenService), a regular webservice provider, and a simple web service client. Another demo shows a CPR webservice example.
The source code and binary files for the demos are NOT included in the distribution, but you may check out the entire source base, as this includes a configured Tomcat web application server, on which the web server demos rely. Make sure you have a command-line Subversion client handy for this task.

Please refer to our wiki site (http://www.sosi.dk/twiki/bin/view/ProjectManagement/SOSISetup) for instructions on how to install Subversion, Maven etc. To check out the project, issue the following command:

```
svn co http://svn.softwareboersen.dk/sosi/trunk/
```

Now CD into the modules root directory and compile the entire source base:

```
cd modules
bootstrap.sh
```

6.1.1 Install service provider demo

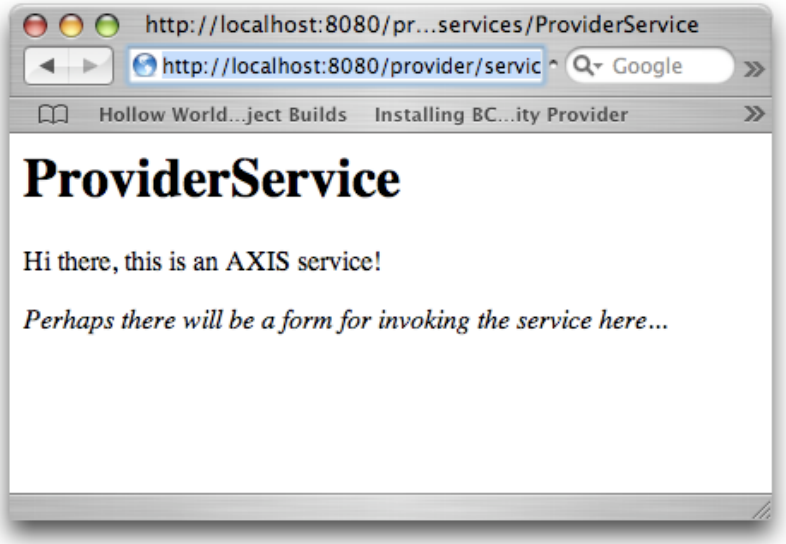
Switch to the demo web service provider, and install it in the following way:

```
cd ../provider
./redeploy.sh
```

Then verify the installation by opening the following URL in your browser:

```
http://localhost:8080/provider/services/ProviderService
```

And check that the result looks something like:

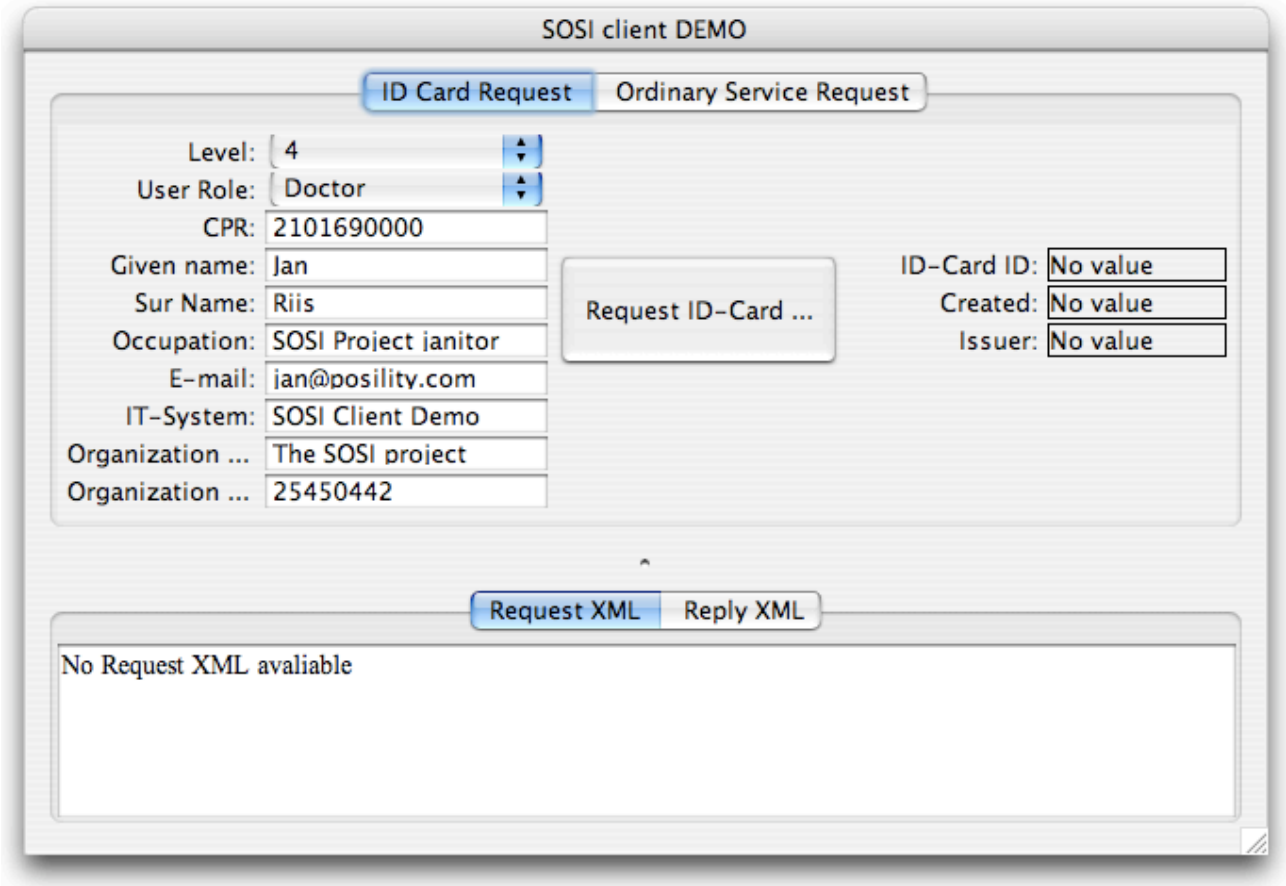


6.1.2 Start the client demo

Finally switch to the client directory, and run the client script:

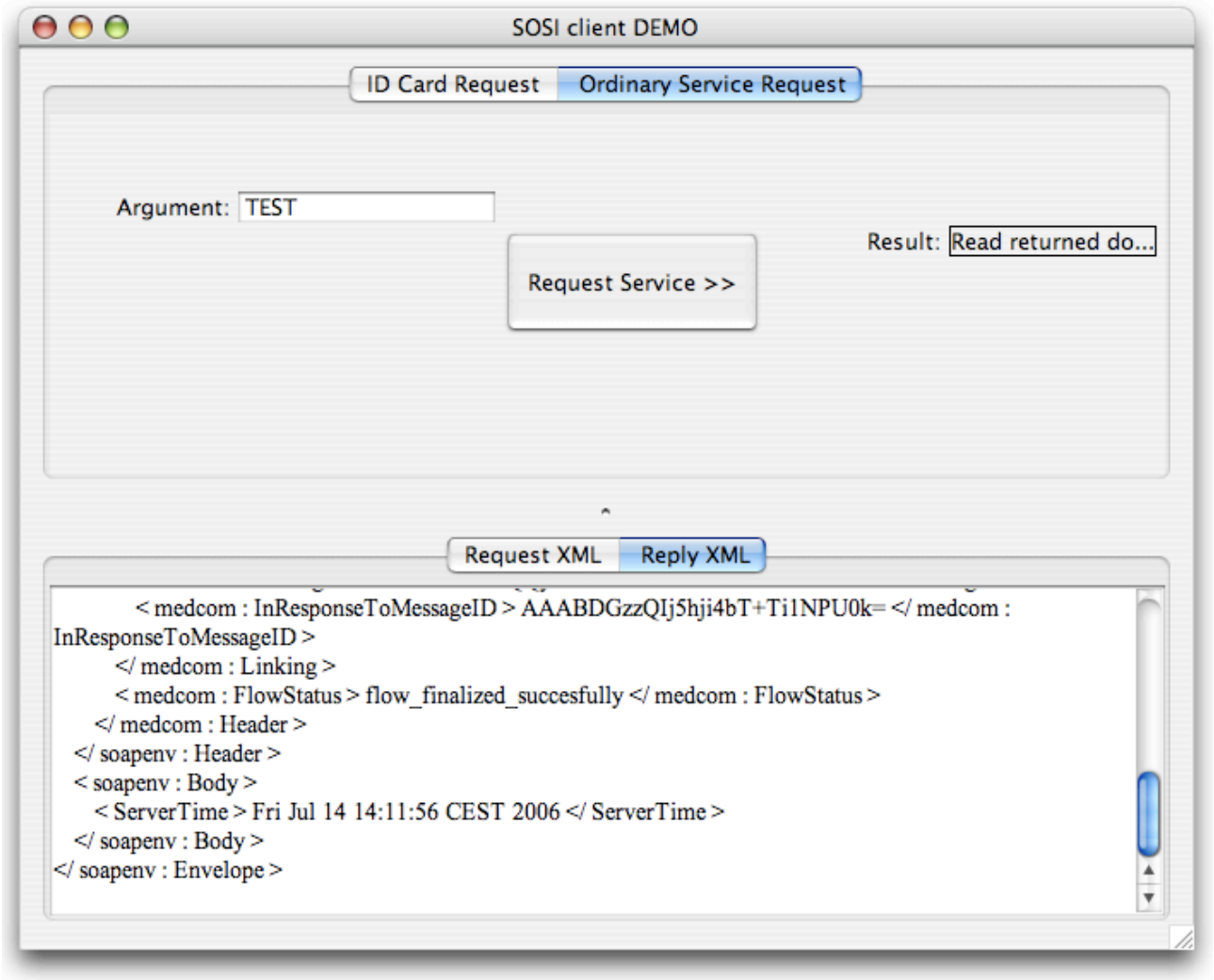
```
cd ../client
./run.sh
```

When the client application launches, you will get a window that looks something like this:



The client expects the demo web services to be running on localhost port 8080. To test the connection to the STS, please select the “Request ID-Card” button, and wait for request and reply XML to be drawn in the lower window.

To test the connection to the service provider, click “Ordinary Service Request” and invoke the web service by pushing the “Request Service” button. The client will request an ID-Card from the service provider, then call the web service with the ID-Card in the soap:Header. The server will check the current time, and return the value to the client as shown below:



6.2 AXIS2 based demo

The axis2 demo application consists of a simple webservice provider and a webservice client which are based on axis2. Both provider and client make use of a custom-built axis2 module that handles the SOSI-specific logic. For more on axis2 modules see http://ws.apache.org/axis2/0_94/userguide4.html.

To run the demo checkout the source base which also includes a configured Tomcat web application server. Make sure you have a command-line Subversion client handy for this task. Please refer to our wiki site (http://www.sosi.dk/twiki/bin/view/ProjectManagement/SOSISetup) for instructions on how to install Subversion, Maven etc. To check out the project, issue the following command:

```
svn co http://svn.softwareboersen.dk/sosi/trunk/
```

Now CD into the modules root directory and compile the entire source base:

```
cd modules
bootstrap.sh
```

6.2.1 Install the axis2 module

Switch to the axis2 module, and install it in the following way:

```
cd demo/axis2-module
./deploy.sh
```

This builds the module and copies it to both the client and provider demos.

6.2.2 Install the provider demo

Switch to the demo web service provider and install it in the following way:

```
cd ../provider-axis2
./redeploy.sh
```

6.2.3 Run the client test suite

Switch to the demo web service client and run the test suite:

```
cd ../client-axis2
mvn test
```

6.2.4 Exploring and modifying the axis2 demo code

To work with the code in your favorite IDE (Eclipse or IntelliJ Idea) you can generate project configuration files for each of the three parts of the axis2 demo by issuing the following:

```
cd ../axis2-module                (or ../client-axis2 or ../provider-axis2)
mvn eclipse:eclipse               (or idea:idea)
```

Each part can then be imported as a separate project.

7 The SOSI Command-line Tool

The SOSI library ships with a command-line tool, which can manage the X.509 certificates and public-private key-pairs that are required for signing and validating signatures. Invoke the tool via the appropriate shell script using either seal.cmd for Windows or seal.sh for Unix systems.

The toolkit will create a .jar file, which has a single java.security.KeyStore inside where all certificates and keys are stored. Use the tool to manipulate the PKCS entries inside as if you were dealing directly with the KeyStore. When used in conjunction with the `ClasspathCredentialVault`, the command-line tool allows cobundling of PKCS entries with e.g. a J2EE application without having to worry about violating the specs by reading from files. The downside is that when keys expire or are revoked, the .jar file will have to be updated and redeployed.

From the code that initializes the Seal, you can now do the following:

```
CredentialVault credentialVault = new ClasspathCredentialVault("xyz987423f");
SOSIFactory sosiFactory = new SOSIFactory(credentialVault, properties);
```

When a certificate or key is needed, Seal will look for the PKCS entries on the classpath assuming that the .jar file has the structure as generated by the command-line tool. Specifically, the ClasspathCredentialVault will try to locate a resource named "SealKeystore.jks" via the context classloader.

Alternatively, if manipulating files is an option in the environment, the toolkit offers direct access to renewing system credentials saved in a Java keystore. This requires the application to use `FileBasedCredentialVault`. This approach circumvents the challenge of the `ClasspathCredentialVault` approach regarding key expiry.

When using the tool, the following syntax applies:

```
-importcert  <path to .cer> -alias <alias> -vault <vault.jar> -vaultpwd
<password> [-props <seal.properties>]

-importpkcs12 <path to .pkcs12> -vault <vault.jar> -vaultpwd <password> -
pkcs12pwd <password> [-props <seal.properties>]

-removealias -alias <alias> -vault <vault.jar> -vaultpwd <password> [-
props <seal.properties>]

-list -vault <vault.jar> -vaultpwd <password> [-props <seal.properties>]

-list -keystore <keystore.jks> -keystorepwd <password> [-props
<seal.properties>]

-renew -vault <vault.jar> -vaultpwd <password> [-props <seal.properties>]

-renew -keystore <keystore.jks> -keystorepwd <password> [-props
<seal.properties>]

-issue -referencenumber <refno> -installationcode <instcode> -vault
<vault.jar> -vaultpwd <password> [-test] [-props <seal.properties>]
```

7.1 ImportCert

Import an X.509 certificate into the vault.jar file under a custom alias. This alias can be used at a later time to remove the entry again using `-removealias`.

```
seal.sh -importcert mycert.cer -alias idpcert -vault vault.jar -vaultpwd
xyz987423f
```

7.2 Importpkcs12

Import a public-private keypair stored in a PKCS12 file into the vault.jar. Seal requires a private key for generating XML signatures (for VOCES signing) and hence only a single keypair is allowed. The keypair is stored under a special alias, "`SOSI:ALIAS_SYSTEM`" which is reserved for this purpose.

```
seal.sh –importpkcs12 voces.p12 -vault vault.jar -vaultpwd xyz987423f –
pkcs12pwd 6tfgdshj1Zxd
```

7.3 Removealias

Remove a PKCS entry from the vault.jar.

```
seal.sh –removealias -alias idpcert -vault vault.jar -vaultpwd xyz987423f
```

7.4 List

List the PKCS entires contained in the vault.jar or in the keystore.jks:

```
seal.sh -list -vault vault.jar -vaultpwd xyz987423f
seal.sh -list -keystore keystore.jks -keystorepwd xyz987423f
```

The contents of the KeyStore are listed to stdout e.g.:

```
Listing contents:

1 : sosi:alias_system (private key, 258 days to expiry)
2 : mycert2 (trusted certificate)
3 : mycert (trusted certificate)
```

Note, that for private key entries, the number of days before expiry of the certificate is also listed. This is useful for determining whether the system credentials are up for renewal.

7.5 Renew

Will perform a renewal of the SOSI system credentials stored in keystore.jks:

```
seal.sh -renew -keystore keystore.jks -keystorepwd xyz987423f
```

A listing of the contents after renewal will reveal that the old certificate and private key is preserved in the keystore as an archive entry:

```
Listing contents:

1 : sosi:alias_system (private key, 730 days to expiry)
2 : sosi:alias_system_1 (private key, 258 days to expiry)
```

See `RenewableFileBasedCredentialVault` for details.

Please note, that renewing a production-grade VOCES certificate is a chargeable service.

7.6 Issue

Will issue a new certificate (test) and store it in keystore.jks:

```
seal.sh  -issue –referencenumber 12344321 –installationcode 98765432
         -vault keystore.jks –vaultpwd xsi78!sD -test
```

Please note, that If the argument `–test` is omitted, a production certificate will be issued. The reference number is the 8 digit number encoded in the URL in the email sent by DanID and the installation code is the 8 digit number found in the letter from DanID.

8 Test tools

As of version 1.4 the library includes tools for generating valid and invalid test STS requests (and corresponding responses) as well as provider requests with valid and invalid ID-cards.

The tools can be used in two ways. The tools can be run by invoking a command script (or shell script). You can also choose to integrate your own unittests with the unittests of the tool or just use the tools directly in your own tests. In either case you preferably should have access to the test STS web service via internet to have the best result.

The following classes are the main classes of the testtools.
STSMessageGenerator contains methods for generating different valid and invalid STS requests.
ProviderMessageGenerator contains methods for generating different valid and invalid Provider requests defined by a xml template.

The unittest classes are the following:
TestSTSMessageGenerator contains tests that generate the different valid and invalid STS requests. It also saves the request and response of each test.
TestProviderMessageGenerator contains tests that generate the different valid and invalid Provider requests. It also saves the request of each test.

The different files pointed on by properties must be included on classpath.

8.1 Properties

The tools include some properties, which you can use to change the setup.
Notice that the classes use the properties defined above them as well

8.1.1 STSMessageGenerator properties

`sositest:saveoutput` = `<true/false>`
This tells the testtool to save the requests/responses that are created.
The default value is **true**

`sositest:outputfolder` = `<output folder>`
This points to which folder the generated requests and responses will end.
The default value is **target**

`sositest:jksfolder` = `<jks folder>`
This points to which folder the used jks keystores must be located.
The default value is **jks**

8.1.2 TestSTSMessageGenerator properties

`sositest:outputfolder`
`sositest:jksfolder`

`sositest:stsrequesttemplate` = < path to template>
This points to which xml template to use when generating STS requests.
The default value is "`requests/lvl4stsrequest.xml`"

8.1.3 ProviderMessageGenerator properties

`sositest:outputfolder`
`sositest:jksfolder`
`sositest:stsrequesttemplate`

`sositest:expiredstsresponse` = < path to template>
This points to the expired STS response xml file which is used when testing provider with expired id cards.
The default value is "`responses/expiredVocesLevel3SystemResponse.xml`"

8.1.4 TestProviderMessageGenerator properties:

`sositest:outputfolder`
`sositest:jksfolder`
`sositest:stsrequesttemplate`
`sositest:expiredstsresponse`

`sositest:providerrequesttemplate` = <path to template>
This points to which xml template to use when generating provider requests.
The default value is "`requests/lvl4providerrequest.xml`"

8.2 Running the tools from a command shell

After installing the SOSI library in your preferred directory on your hard disk, change to the "`testtools`" directory and issue the "`runtesttools`" command script or shell script.
This should result in something like this:

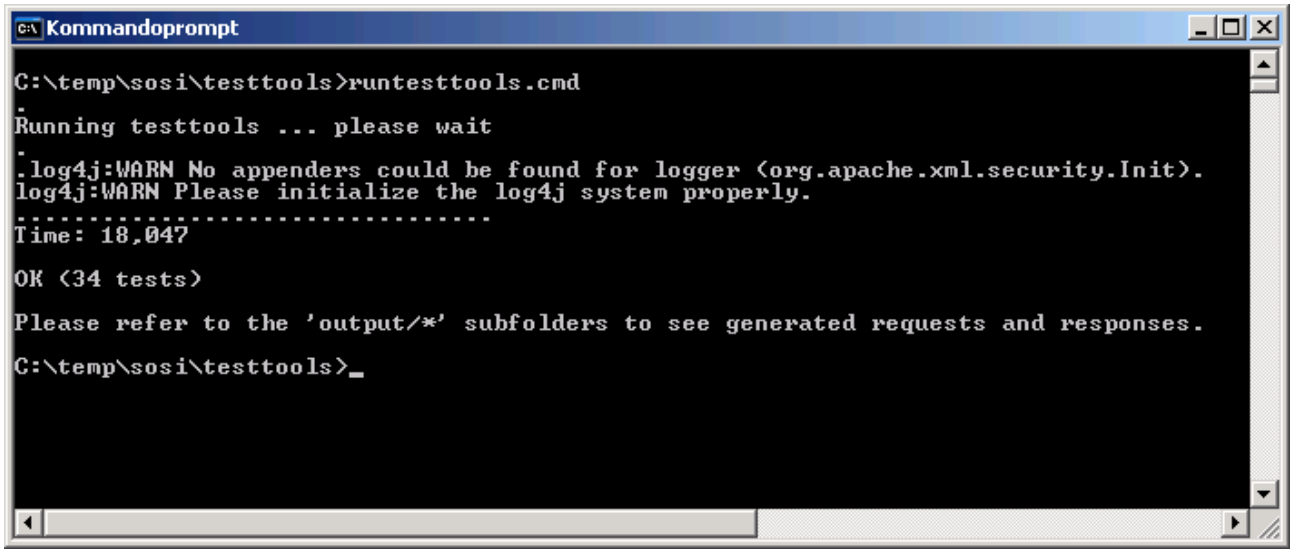


Figure 9 - Expected output when running the "runtesttools.cmd" in a windows shell.

As shown on the screen, the resulting XML documents can be found in subfolders in the 'output' directory.

The `providerrequests` folder contains generated requests with various valid and invalid certificates and ID-cards, based on the methods found in the `TestProviderMessageGenerator` class. These XML files can be used for testing security related behavior at service providers.

The **`requests`** folder contains generated valid and invalid STS requests based on the methods in `TestSTSMessageGenerator` class. If the Test STS is reachable (on the net) these requests has been sent to the Test STS and the resulting STS responses can be found in the **`responses`** folder. As the tool is based on unittests, the resulting response has also gone through various assertions to validate that the STS actually replies what is expected.

9 Known bugs and bug reporting

At present time of writing there are no known bugs in the main library (seal). However, the demos are not that well documented and tested since most of the energy has been spent in stabilizing the main library.

Right now the test coverage of the main library is approximately 89%. The performance is tuned so that all of the main use cases amounts to less than 40 milliseconds on a standard server.