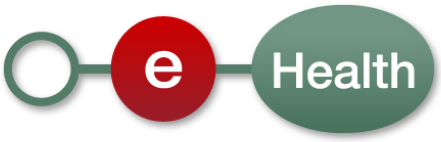

eHealth platform Services Connectors

Version 3

Introduction Guide

This document is provided to you free of charge by

The eHealth platform

Willebroekkaai 38

1000 BRUSSELS

Table of Contents

Document history

Version	Date	Author	Description of changes / remarks
1.0	04/03/2012	eHealth platform	First version of the document, distributed with release 3.0-beta.
1.1	15/05/2013	eHealth platform	Second version of the document, distributed with release 3.1-beta. No major change.
1.2	28/08/2013	eHealth platform	Second version of the document, distributed with release 3.2-beta. Add of paragraph 2.2.5, update of paragraph 4.3..
1.3	05/12/2013	eHealth platform	Revision of the document, distributed with release 3.3-beta. Update Section 3 and 5
1.4	30/04/2014	eHealth platform	Revision of the document, distributed with release 3.4-beta. Update paragraph 2.1
1.5	15/05/2014	eHealth platform	Revision of the document, distributed with release 3.4-beta. Adding paragraph 4.7
1.6	15/06/2014	eHealth platform	Revision of the document, distributed with release 3.4-beta. Adding paragraph 4.8
1.7	12/02/2014	eHealth platform	Revision of the document, distributed with release 3.4-beta. Adding paragraph 4.8
1.8	31/03/2015	eHealth platform	Revision of the document, distributed with release 3.4-beta. Adding paragraph 5

Introduction

Main objective of this document is to provide a technical overview of the usage of the eHealth Connector (version 3). The goal is to guide an application developer of an end-user software application on how to use, configure, extend and integrate his application with the eHealth Connector.

This document is not:

- A complete development cookbook. This document is targeted to skilled Java and/or .Net developers capable of integrating an external framework/component;
- A development cookbook to invoke eHealth-platform services. For such information, the reader should consult the dedicated cookbooks of those services.
- The API (Application Programming Interface) description of the eHealthConnector. For this purpose, the reader is invited to directly consult the available “Javadoc”.
- Every release archive of the connector contains a release notes file with the changes that are made compared to the previous release. It is recommended to read this file before using/installing the new version of the connector.

This document contains five different chapters. The first chapter describes the general architectural principles and the architectural overview. The second chapter, “How to use the connector” gives more information about the content and structure of a release archive, a step by step description how to integrate the connector in your development environment and the prerequisites of the connector. The third chapter describes in detail the different options in the configuration of the connector. The best practices on how to create a session, invoke a business service, the use of the generic service can be found in chapter four. The last chapter describes all the known limitations of the connector.

1 Architectural overview

1.1 Overview

This document describes the usage of the eHealth Connector that facilitates and simplifies the interaction and usage of eHealth-platform services and other value-added services.

The main objective of the eHealth Connector is not only to hide the technical complexity of accessing these remote services, but also certain business complexities. The services and operations provided via this library are grouped in functional blocks that correspond with real end-user related functions. One operation may involve multiple low-level technical services, both local and remote, which are hidden for the client application developer. In addition, an eHealth-platform session is created, maintained and shared as part of this connector.

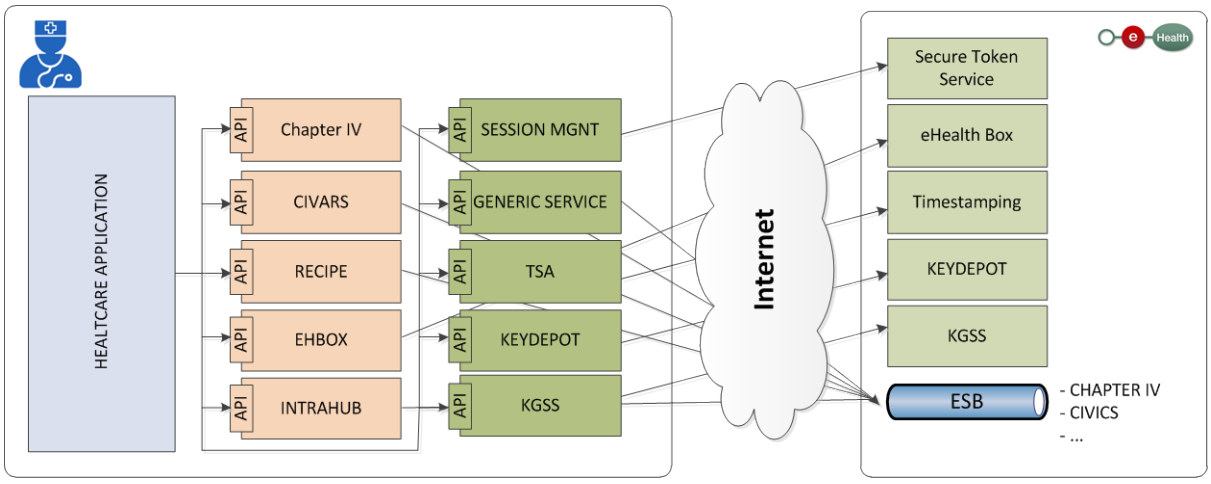

Figure 1: Overview architecture connector

Although a Java and Microsoft .Net version of the Connector are available, this document is written in a technology neutral manner. Actual code examples will be provided both in a Java and .Net flavor. Every release archive business or technical is available in a Java and .NET version. The archive is always self-containing, all the needed dependencies and examples are provided.

There are 2 types of building blocks, the technical and the business building blocks.. The technical blocks are there for general purpose and can be used by other building blocks. The technical blocks don't interact witch each other. Examples of these blocks are the session management and the services needed for end to end encryption. Other services like the eID reader and configuration framework are also 'technical' building blocks. The second type of building blocks are the business blocks. Those blocks interact with a service exposed for target user group. Those services can be exposed via the eHealth Service Bus. More information on the structure of a business block is provided is section 4.2.

1.2 Principles

In order to have a good understanding of the provided functionality and working of the Connector, please review the following base principles that have been used to define the different services.

- **Target user group**: the Connector is aimed at software developers of client applications for the given target group (for example: general practitioner) that are willing to integrate eHealth-platform value-added services in their end-user applications.
- **Service simplification**: the Connector provides a simple way to invoke remote eHealth-platform base and value-added services by hiding the business/functional and technical complexity. A typical service provided by the Business Connector invokes multiple local and remote services, for example, Session Management and end to end encryption related services, in order to achieve the assigned goal
- **Data simplification**: the input and output parameters for each of the operations are simplified to only provide the business relevant data. General or technical parameters are hidden and are added automatically by the connector.
- **Session management:** an end-user must possess a valid session initiated via the connector's Session Management component. It is advised to create a session as soon as the client application embedding the Connector is launched.
- **Builder pattern:** with every complex object needed by the connector a Builder with associated BuilderFactory is provided. Such a builder can generate all of the needed objects and takes the relevant parameters as input.

2 Use of the connector

The connector is available in a Java and a Microsoft .NET version. The Java version is provided as Java Archive (JAR). The Microsoft .NET version is proved as a DLL. However the API for both versions is identical.

The Connector is depending on a series of external components for its correct functioning. As part of the distribution, there external dependencies are includes. The entire list of dependencies could be found in the lib directory of the connector.

Besides a series of external dependencies, the Connector requires a **configuration file** containing a limited number of properties (for example: general information related to the end-user(s) like NIHII, name and SSIN and certificate configuration). In the archive of the connector you will find an example of such a configuration file. The empty properties are user dependent properties. Before using the connector you must fill in those properties. If you want to change the behavior of the connector you could change prefilled properties.

The following **documentation** is available in addition to this cookbook:

- An **API** archive describing the classes and services as exposed by the connector.
- A release package (as an archive file) containing the following directories:
    - A **config** folder containing the following files:
        - be.ehealth.technicalconnector.properties: the sample properties files
        - be.ehealth.technicalconnector-test.properties: the properties needed to execute the tests
        - a P12 directory containing 3 JKS's files
            - truststore.jks contains all the certificates trusted for SSL communication
            - caCertificateKeystore.jks contains all the certificates trusted by the ETEE.
            - tsacertificate.jks contains all the certificates trusted for Timestamping
    - A **lib** folder containing the connector and its external dependencies (classpaht)
    - An **example** folder which contains the examples of use of the connector. This sample/test code functions as reference implementation.
    - A **license** and **notice** directory containing the licenses and notices as specified by the licenses of the connector dependencies.
    - A **test-lib** folder containing the external dependencies needed for testing, using and integrating the Connector.

2.1 Step by step

Following steps must be executed properly to ensure the correct working. Each step is detailed in the next sections.

1. Installation of the VM

Install Java or .NET on your system (if not yet installed), see the pre-requisites section of either Java or .NET for more details.

2. Download the correct version of the connector

The Business connector or the technical connector must be downloaded. Every archive of the connector contains the dependencies it needs. The business connector contains also the required technical connector version.

3. Include the required dependencies in your development project

Include all the third party dependencies in your project. All those dependencies are present in de lib and test-lib directory of the connector.

4. Copy all keystores to a directory in your project

At least 3 keystores are needed for the correct function of the complete connector (depending on the services you will integrate). See section “Properties – Keystore configuration” for more details. The keystores are:

- Certificate Authority Keystore (caCertificateKeystore.jks)
- SSL Truststore (truststore.jks)
- Identification eHealth Certificate Keystore
- Holder-Of-Key eHealth Certificate Keystore
- Encryption eHealth Certificate Keystore

Those keystores must be copied to the directory you specified in the configuration file (property: KEYSTORE_DIR) and the directory must be included and accessible in your development project.

5. Configure the property file

The property file must be configured as described in section “Properties Configuration”.

If the examples need to be run, then the additional test configuration is required as well. The configuration file(s) must be included in your project.

It is advised to copy the “config” directory that is included in the distribution to your project and include it as a part of your run-time configuration (e.g.: for Java this means putting the entire config folder on the class path). This directory already contains a basic properties file and a directory for your keystores with the Certificate Authority Keystore and SSL Truststore already in place.

If the above method is used, in combination with the proposed folder layout, the configuration file included in the distribution config folder does not need to be updated and can be used out of the box to go to the acceptance test environment end-points (production end-points need a small configuration change).

Most properties in the configuration file are user/package independent but some properties are not. For example every property that starts with user, sessionmanager, pharmacy and pharmacy-holder are user dependent and the properties that start with package, mycarenet.licences are package dependent. Those parameters are empty within the distributed property file and must be correctly filled out to have the correct behaviour of the connector.

6. Create custom code using the Connector API

Start writing your own code that uses eHealth-platform and value-added services through the Business Connector API. The examples section at the end of this document provides a complete code sample for invoking one of the services.

2.2 Pre-requisites

2.2.1 Virtual Machine

The Java Runtime Environment (JRE) 1.6 or higher must be installed on each system on which the Java version of the connector is used, for either development or production usage.

.NET Framework 4.0 must be installed on each system on which the .Net version of the connector is used, for either development or production usage.

Note: The use of eID requires a 32-bit JVM.

2.2.2 Dependencies

The connector also requires certain dependencies for its proper functioning. These libraries are mainly used to deal with web services and security. See the content of the lib and test-lib folders for more information.

2.2.3 Internet connectivity

The connector invokes multiple services offered by eHealth-platform (and other service providers). These services are invoked over the Internet via HTTPS. The computer or server on which this Connector for General Practitioners is executed must therefore have access to the Internet.

If you need a proxy for connecting to the internet, please be sure that the ConfigurationModuleProxy module is correctly loaded. See the properties file for more information on how to configure this module.

2.2.4 eID middleware and card reader

The eID middleware permits the creation of a user session based on the user's eID card. It requires having an eID card reader installed. There are 2 implementations available.. One implementation uses the PKCS11 technology as offered by the Belgian eID middleware software (version 3.5). More information, including installation and setup guidelines, are available on the following website: http://eid.belgium.be/.

The other implementation uses the PCSC technology as offered by the standard JVM. This implementation is recommended by fedICT. For the JAVA users is no additional software needed. The .NET users must install the Microsoft Visual C++ 2010 Redistributable Package.

2.2.5 Temporary files

The connector places some files in a temporary directory for temporary storage purposes. This is done with a java temporary file, which is automatically deleted when the jvm shuts down.

Sometimes the ikvm implementations don't do this shutdown correctly and the files are not deleted. To prevent this you could execute a System.exit() command as last statement of your code when exiting the connector.

3 Properties configuration

Every connector is depending on the unified configuration file, which includes eHealth-platform base services and Session Management related properties. Examples: keystore location, connection details, environment details, etc. The properties configuration describes a minimal configuration needed to use the connector in your own development project.

For Java the *default* location of this properties file is the classpath itself. For .NET version the default location is the directory of the different dll's. The default location can be changed by executing the folling code snippet.

Note: Be sure that this is executed before the first usage of the connector; otherwise the default location is loaded.

JAVA
`ConfigFactory.setConfigValidator("location of the properties file");`

.NET C#
`ConfigFactory.setConfigValidator("location of the properties file");`

Sample files are provided as part of the distribution and can be used as a starting point. The following two sections describe the configuration files in more detail.

3.1 Configuration modules

It is possible to automatically load configuration modules when the configuration is initialized. Those modules are defined in the configuration file. The properties must start with connector.configmodule followed with a number. Those numbers must start with 1 and increment with 1, gaps between two entries are not allowed.

The connector ships with

- a logging module (ConfigurationModuleLogging)
- a proxy support module (ConfigurationModuleProxy)
- a module that redirects the System.out to the logging framework (ConfigurationModuleSysOut)
- a module that changes the location of the truststore (ConfigurationModuleSSL)
- a module that links the current classloader to the current thread, only needed for the .NET connector (ConfigurationClassloader)
- a module that changes the default language of your runtime environment (ConfigurationModuleDefaultLanguage)
- a module that loads XmlSec 1.5 instead of the default XmlSec implementation of your runtime environment (ConfigurationModuleXmlSec)

Additional modules can be written by implementing the ConfigurationModule interface.

3.2 Endpoints of the webservices

The properties that start with endpoint.* are the addresses of the end points of the web services exposed by eHealth-platform. The URL's for the acceptance test environment are included in the distribution (only to be used for testing purposes, not for production usage).

Note: These properties must be changed to access production end points.

3.3 Keystore configuration

All keystores (P12/JKS) that the connector uses need to be stored in the same directory: the KEYSTORE_DIR property points to this directory. This location should be relative to the root of the project hierarchy or an absolute path on the file system.

The key stores that should be in this directory are:

- **Certificate Authority Keystore** (*.jks)
  The name and password of this keystore must be specified in the following properties CAKEYSTORE_LOCATION and CAKEYSTORE_PASSWORD
- **SSL Truststore** (*.jks)
  The name and password of this truststore must be specified in the following properties truststore_location and truststore_password
- **Identification Certificate Keystore** (*.p12): a keystore that is used to identify the 'end-user' towards the eHealth SecureTokenService. In case of Sessin creation with eID this property will be ignored. The name of this keystore must be specified in the property sessionmanager.identification.keystore
- **Holder-Of-Key eHealth Certificate Keystore** (*.p12): a keystore that is used to secure the communication between the software and services with added value. The certificate inside this keystore must be an eHealth ETEE certificate.
  The name of this keystore must be specified in the property sessionmanager.holderofkey.keystore.
- **Encryption eHealth Certificate Key Store** (*.p12): a keystore that is used to seal and unsealed the ETEE messages. The certificate inside this keystore must be an eHealth ETEE certificate. The name of this keystore must be specified in the property session.manager.encryption.keystore.

Information regarding requesting and using eHealth certificates can be found on the eHealth portal.

3.4 Session management configuration

The Session Management component takes care of the end-user's session, so that secured eHealth services can easily be accessed through the connector. A session should be created before using any other service of the connector. The session will be cached for re-use until it is expired.

The Session Management offers two ways of creating a session:

- Normal session: the eID is used, the user will be requested to insert the eID card in the card reader and enter his/her PIN code.
- Fallback session: a personal eHealth-platform issued certificate is used.

In addition to creating session, the Session Manager can also load in a session that has previously been stored. If a session is not needed anymore it should be unloaded via the Session Manager. To check if there is a valid session active for the Business Connector to use, the Session Manager has a "hasValidSession" operation.

In short, before invoking any of the business services, the Session Manager should be invoked to make sure that there is a session.

The behavior of the session management is configurable through the configuration file. In the request that the connector sends to the SecureTokenService there are 2 types of attributes present. Attributes that give more information on the subject (samlattributes) and attributes that are verified/certified by eHealth(samlattributedesignators).

The subject attributes must start with sessionmanager.samlattribute followed with an incrementing number. For his type of attributes 3 types of information must be present after the = sign. Each information must be separated with a comma. First of all, the namespace of the attribute is normally this namespace: urn:be:fgov:identification-namespace. The second information is the name of the attribute, and finally the last information is the value of the attribute. If the value is already present in the configuration file, ${key} could be used to dynamically load the property.

The attributes that must be verified must start with sessionmanager.samlattributedesignator followed with an incrementing number. This type of attribute requires only 2 types of information after the = sign, the namespace and the name of the attribute.

3.5 Business service specific properties

Some business services require additional information in the configuration file. For example

- the default hub that must be used to connect to.
- user information needed by the sessionmanagment
- Activating or de-activating the incoming message validation.
- the licence information needed by the ChapterIV webservice

Those specific configurations will be documented in the configuration file in the archive. It is important that those sections are correctly filled in otherwise the business service will not function correctly.

4 HOW TO

4.1 How to setup a user session?

Create a session at the beginning of your application life cycle and keep the resulting SAML Token on a secured location if you wish to re-use it later.

Use the hasValidSession method before invoking a method on the business service, this will ensure that a valid session is available. This will result in less errors because invoking a business service without a session is not possible

Unload a session before creating a new session, use this in combination with the above remark (use the hasValidSession method) to make sure you are not overwriting an active session by accident (checking the hasValidSession method) and to make sure that the new session can start clean (unloading the session).

There are 2 ways to setup a valid user session one with a Belgian eID and another with a Personal eHealth certificate. The first one should be used by default, only in case of loss/theft of the eID the fallback session should be used. The lifetime of session obtained by the fallback way is shorter than one obtained with an eID.

JAVA

```
import be.ehealth.technicalconnector.session.SessionManager;

String hokPassword = “password of Holder-Of-Key eHealth Certificate Key Store”;
String persPassword = “password of Personal eHealth Certificate Key Store ”;
SessionManager sessionmgmt = Session.getInstance();
if (!sessionmgmt.hasValidSession()) {
  sessionmgmt.createSession(hokPassword, persPassword);
}else{
  sessionmgmt.unloadSession();
  sessionmgmt.createSession(hokPassword, persPassword);
}
```

.NET C#

```
using System;
using be.ehealth.technicalconnector.session;

String hokPassword = "password of Holder-Of-Key eHealth Certificate Key Store";
String persPassword = "password of Personal eHealth Certificate Key Store ";
SessionManager sessionmgmt = Session.getInstance();
if (!sessionmgmt.hasValidSession()) {
  sessionmgmt.createSession(hokPassword, persPassword);
}else{
  sessionmgmt.unloadSession();
  sessionmgmt.createSession(hokPassword, persPassword);
}
```

4.2 How to invoke a business service?

Before you want to invoke a business service, please verify that there is an active valid session. The structure of every business service is the same. It contains builders, exceptions and session. The other packages are normally for inside use. The package builders contains the necessary request and response builders. A builder can be obtained by invoking the corresponding static method in the BuilderFactory. The package exceptions contains all the specific exceptions that the business service may throw. The package session is the most important package and it contains all the available webservices and methods. In the session package a class that ends with ServiceFactory is present. This class contains some static methods in order to obtain the correct webservice stub within a session.

This logic is illustrated below with the intrahub service.

1. Check if there is a valid session
2. Obtain a request builder and create the request
3. Invoke the webservice and obtain the response
4. Analyse the response by using the response builder.

JAVA

```
import be.ehealth.technicalconnector.session.SessionManager;
import be.ehealth.technicalconnector.session.Session;

SessionManager sessionmgmt = Session.getInstance();
if (sessionmgmt.hasValidSession()) {
  BuilderFactory factory = BuilderFactory.getInstance();
  RequestBuilder reqBuilder = factory.getRequestBuilder();

  String hcpartyXML = "";
  HCPartyIdType hcparty = reqBuilder.buildHCPartyIdType(hcpartyXML);

  HubService hubs = HubSessionServiceFactory.getHubService();
  ConsentHCPartyType respCons = hub.getHCPartyConsent(hcparty);

  ResponseBuilder respBuilder = factory.getResponseBuilder();
  String consentXML = respBuilder.buildConsentHCPartyTypeResponse(respCons);
  // business logic
}
```

.NET C#

```
using System;
using be.ehealth.technicalconnector.session;


SessionManager sessionmgmt = Session.getInstance();

if (sessionmgmt.hasValidSession()) {

  BuilderFactory factory = BuilderFactory.getInstance();
  RequestBuilder reqBuilder = factory.getRequestBuilder();


  String hcpartyXML = "";
  HCPartyIdType hcparty = reqBuilder.buildHCPartyIdType(hcpartyXML);

  HubService hubs = HubSessionServiceFactory.getHubService();
ConsentHCPartyType respCons = hub.getHCPartyConsent(hcparty);


  ResponseBuilder respBuilder = factory.getResponseBuilder();
  String consentXML = respBuilder.buildConsentHCPartyTypeResponse(respCons);
  // business logic

}
```

4.3 How to use the generic services of the technical connector?

It is possible to invoke a web service within a session that isn't supported by the connector. Or you want to invoke a supported web service without using the facilities offered by the connector. There are 2 methods available sendXML and sendDocument. The first one takes the xml to send a string as input. The second one uses a dom document to transfer the info.

JAVA

```
import java.net.URL;
import org.w3c.dom.Document;
import org.w3c.dom.Node;

import be.ehealth.technicalconnector.generic.session.GenericService;
import be.ehealth.technicalconnector.generic.session.GenericSessionServiceFactory;

if (sessionmgmt.hasValidSession()) {
  GenericService service = GenericSessionServiceFactory.getGenericService();

  String payload = "message to send";
  URL endpoint = new URL("endpoint of the service");
  String response = service.sendXML(payload,endpoint);

  Document doc = "";
  Node respNode = service.sendDocument(doc,endpoint);
}
```

.NET C#

```
using System;
using java.net;
using javax.xml.parsers;
using org.w3c.dom;
using be.ehealth.technicalconnector.session;
using be.ehealth.technicalconnector.generic.session;


if (sessionmgmt.hasValidSession()) {
  GenericService service = GenericSessionServiceFactory.getGenericService();


  String payload = "message to send";
  URL endpoint = new URL("endpoint of the service");
  String response = service.sendXML(payload,endpoint);


  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  DocumentBuilder build = factory.newDocumentBuilder();
  Document doc = builder.newDocument();
  Node respNode = service.sendDocument(doc,endpoint);
}
```

Since version 3.2 of the connector a new method on the GenericService is added. The genericService can be invoked with a GenericRequest. This GenericRequest contains the payload, soapaction, endpoint and the SOAP handlerchain. The payload can be added as string, dom source or as object. Please notice that the java object must be an object generated by JAXB with the annotation @XmlRootElement, otherwise the connector is not able to unmarshall the object. The result of the invoke method is a GenericResponse. You can obtain the response of the invoked webservice as a string, dom source or as object. The method getSOAPException() allows to check if there was no SOAP error present, this is useful for a void operation.

JAVA

```
import be.ehealth.technicalconnector.ws.domain.*
import be.ehealth.technicalconnector.ws.*

GenericRequest request = new GenericRequest();
request.setPayload("message to send");
request.setEndpoint(“);
request.setDefaultHandlerChain();
GenericWsSender sender = ServiceFactory.getGenericWsSender();
GenericResponse response = sender.send(request);
ResponseObj resp = response.asObject(ResponseObj.class);
```

.NET

```
using be.ehealth.technicalconnector.ws.domain.*
using be.ehealth.technicalconnector.ws.*

GenericRequest request = new GenericRequest();
request.setPayload("message to send");
request.setEndpoint(“);
request.setDefaultHandlerChain();
GenericWsSender sender = ServiceFactory.getGenericWsSender();
```

```
GenericResponse response = sender.send(request);
ResponseObj resp = response.asObject(ResponseObj.class);
```

4.4 How to change a property at runtime?

It is possible to change properties at runtime. When an object is created sometimes properties are used to correctly initialize the object. If you change those properties after the object creation off course the property change is not reflected, a new object must be created.

For example when you change a hub.id and endpoint.hub.intra on the fly, verify that a new intrahub service is created otherwise the old properties are used.

JAVA

```
List<String> reqProps = new ArrayList<String>();
ConfigValidor confValidator = ConfigFactory.getConfigValidator(reqProps);
Configuration config = confValidator.getConfiguration();
config.setProperty("hub.id","1990000035");
config.setProperty("endpoint.hub.intra","https://my.hub01.be/IntraHubService");

HubService hubs = HubSessionServiceFactory.getHubService();
// business logic

config.setProperty("hub.id","1990000134");
config.setProperty("endpoint.hub.intra","https://my.hub02.be/IntraHubService");

hubs = HubSessionServiceFactory.getHubService();
// business logic
```

.NET C#

```
using System;
using java.util;
using be.ehealth.technicalconnector.config;
using be.ehealth.businessconnector.hub.session;

List reqProps = new ArrayList();
ConfigValidator confValidator = ConfigFactory.getConfigValidator(reqProps);
Configuration config = confValidator.getConfig();
config.setProperty("hub.id","1990000035");
config.setProperty("endpoint.hub.intra","https://my.hub01.be/IntraHubService");

HubService hubs = HubSessionServiceFactory.getHubService();
// business logic

config.setProperty("hub.id","1990000134");
config.setProperty("endpoint.hub.intra","https://my.hub02.be/IntraHubService");

hubs = HubSessionServiceFactory.getHubService();
// business logic
```

4.5 How to handle a java.util.Collection<?>

In the java version the collections are typed by using generics. This functionality is not present in the .NET connector. When a collection is returned you must iterate over the collection and cast every item to the correct object. See code snipped below.

.NET C#

```
using java.util;

Collection inCollection = server.getCollection();
Iterator collectionIt = inCollection.interator();
Object item = null;
while (collectionId.hasNext()){
  item = (Object) collectionIt.next();
  // business logic
}
```

4.6 How to exit the JVM correctly through IKVM

Apparently if you are using the .NET version of the connector, the Java Virtual Machine is not correctly shutdown. All the shutdown hooks are not executed, for example: the temporary files created by the connector are not deleted afterwards. If you want to shut down the JVM correctly

.NET C#

```
java.lang.Runtime.exit(0)
```

4.6.1 How to use the eID on Windows 8 through IKVM

The system property os.name is not correctly set when you are using IKVM and Windows 8. This property is used inside the commons-eid framework to enable or disable a patch. If you are experiencing “begin exclusive” failures in your project and the application is on a Windows 8 platform, you must apply this patch.

In the startup sequence of your application you must add the following line:

.NET C#

```
java.lang.System.setProperty("os.name","Windows 8");
```

Other operating systems are not impacted.

4.7 How to overload an implementation in the connector

The connector is using the IoC desing pattern. The connector code depends always on the interface and not on the implementation. By changing values in the properties file another implementation class could be loaded. More information on which property must be set in order to overload the default implementation could be found in the javadoc.

Inside the connector we are using the fully qualified name of the class for instantiating the implementation class. For the ikvm java class loader every .NET class is prefixed with cli followed by the fullName of the class. If you are using inner classes in .NET the + sign is used as delimeter within Java the $ sign is used. So the .NET specification must be translated to the JAVA specs.

Below there is an illustration for implementation overloading with JAVA and with .NET

JAVA

```
private class Overloading() implements SomeInterface{
  public void init(){
    //init logic
  }
}

ConfigValidor confValidator = ConfigFactory.getConfigValidator();
Configuration config = confValidator.getConfiguration();
config.setProperty("overloading.prop",Overloading.class.getName());
```

.NET C#

```
using System;
using java.util;
using be.ehealth.technicalconnector.config;

public class Overloading : SomeInterface {
  public void init(){
    //init logic
  }
}

ConfigValidator confValidator = ConfigFactory.getConfigValidator();
Configuration config = confValidator.getConfig();
string className = “cli”+ typeof(Overloading).FullName.Replace(“+”,”$”);
config.setProperty("overloading.prop",className)
```

5 Connecting to a mycarenet asynchronous service

5.1 Code examples

See /examples folder in zip file

There is a generic example under examples\be\ehealth\businessconnector\genericasync\integration and for specific implementations there are examples provided under examples\be\ehealth\businessconnector\{projectName}\

5.2 Usage and configuration parameters

see mycarenet documentation Service_Catalogue_genericAsync.pdf on share.intermut.be

For services tested by ehealth the configuration is already provided.

5.2.1 Example configuration for invoicing

Can be found in property file /config/ be.ehealth.technicalconnector.properties : (these property files are provided for each release)

```
# Configuration of BUSINESS module
#
################################################################################
# configuration for packageInfo
genericasync.invoicing.package.licence.username=${mycarenet.licence.username}
genericasync.invoicing.package.licence.password=${mycarenet.licence.password}
genericasync.invoicing.package.name=${package.name}

#indicate the xades level needed on the get response , possible values : none ,
#xades , xadest
genericasync.invoicing.mycarenet.get.response.neededxadeslevel=xadest

#define the endpoint to use
endpoint.genericasync.invoicing.v1=https://pilot.mycarenet.be/mycarenet-
ws/async/generic/hcpfac

#indicate if you wish to validate the incoming xml with xsd validation ( check if
#the format is correct )
validation.incoming.message.genericasync.invoicing.v1=true

#configure blob type for invoicing : fill these properties with the values
specified in the mycarenet documentation
#indicate that you don't want to use the default values for this project
mycarenet.blobbuilder.invoicing.usedefaultproperties=false
#indicate the name of the id , by default we use the name blob
mycarenet.blobbuilder.invoicing.id=blob
#indicate the encoding type, possible values none, deflate
mycarenet.blobbuilder.invoicing.encodingtype=deflate
#indicate the content mime type
mycarenet.blobbuilder.invoicing.contenttype=text/plain
```

5.3 usage

5.3.1 Create the service

JAVA

```
GenAsyncService service = GenAsyncSessionServiceFactory.getGenAsyncService(projectName);
```

Configuration property

```
endpoint.genericasync.{projectName}.v1=https://…mycarenet.be/mycarenet-ws/async/generic/…
```

5.4 Create requests

5.4.1 Create post Request

JAVA

```
import be.ehealth.business.mycarenetcommons.builders.BlobBuilderFactory;
import be.ehealth.business.mycarenetcommons.builders.CommonBuilder;
import be.ehealth.business.mycarenetcommons.builders.RequestBuilderFactory;
import be.ehealth.business.mycarenetcommons.domain.Blob;
import be.ehealth.business.mycarenetcommons.mapper.SendRequestMapper;
import be.ehealth.technicalconnector.config.util.domain.PackageInfo;
import be.ehealth.businessconnector.genericasync.builders.BuilderFactory;

byte[] contentBytes = … // create your content here
BlobBuilder bbuilder = BlobBuilderFactory.getBlobBuilder({projectName});
Blob blob = bbuilder.build(contentBytes);
blob.setMessageName("someMessagename");//messagename found in mycarenet
documentation

String inputReference = IdGeneratorFactory.getIdGenerator("xsid").generateId();
PackageInfo packageInfo = ConfigUtil.retrievePackageInfo("genericasync." +
"{projectName}");
//for more
//usage of mycarenet factories and builders : see javadoc
CommonInput ci =
CommonInputMapper.mapCommonInputType(RequestBuilderFactory.getCommonBuilder("{pro
jectName}").createCommonInput(packageInfo, true, inputReference));

 be.cin.types.v1.Blob det = SendRequestMapper.mapBlobToCinBlob(blob);
 BlobType blobForXades = SendRequestMapper.mapBlobToBlobType(blob);
//generate xades if needed ( see mycarenet documentation and javadoc)
//usage of BlobUtils  see javadoc
 byte[] xades = BlobUtil.generateXades(blobForXades).getValue();
RequestObjectBuilder requestBuilder =
BuilderFactory.getRequestObjectBuilder({projectName});
```

```
Post post = requestObjectBuilder.buildPostRequest(ci, det, xades);
```

Configuration property
PackageInfo : see genericasync.{projectName}.package.*
BlobBuilder : configured with mycarenet.blobbuilder.{projectName}.* , or mycarenet.blobbuilder.default.* properties , see Javadoc
CommonBuilder : configured via mycarenet.{projectName} properties , see Javadoc
RequestObjectBuilder : no configuration properties at the moment

5.4.2 Create get request

JAVA

```
import be.ehealth.business.mycarenetcommons.builders.CommonBuilder;
import be.ehealth.business.mycarenetcommons.builders.RequestBuilderFactory;
import be.ehealth.businessconnector.genericasync.mappers.CommonInputMapper;
import be.ehealth.businessconnector.genericasync.builders.BuilderFactory;
import be.ehealth.technicalconnector.config.util.ConfigUtil;

CommonBuilder commonBuilder =
RequestBuilderFactory.getCommonBuilder({projectName});
PackageInfo packageInfo = ConfigUtil.retrievePackageInfo("genericasync." + {
projectName});
OrigineType origin =
commonInputMapper.mapOrigin(commonBuilder.createOrigin(packageInfo));


MsgQuery msgQuery = new MsgQuery();
msgQuery.setInclude(true);
msgQuery.setMax(100);
msgQuery.getMessageNames().add("MessageType1"); //see mycarenet documentation for
messageTypes
msgQuery.getMessageNames().add("MessageType1");"); //see mycarenet documentation
for messageTypes

Query tackQuery = new Query();
tackQuery.setInclude(true);
tackQuery.setMax(100);
GetRequest getRequest =
BuilderFactory.getRequestObjectBuilder({projectName}).buildGetRequest(origin,
msgQuery, tackQuery);
```

Configuration property
packageInfo : genericasync.{projectName}.package.* properties
RequestBuilderFactory.getCommonBuilder : normally default values , see Javadoc for configuration
RequestObjectBuilder : no configuration properties at the moment

5.4.3 Create confirm request

JAVA

```
//retrieve message hash values to confirm
for (MsgResponse msgResp : responseGet.getReturn().getMsgResponses()) {
   final byte[] hashValue = msgResp.getDetail().getHashValue();
   msgHashValues.add(hashValue);
}
// tack

//retrieve technical messages hash values to confirm : same as messageResponses,
// but from responseGet.getReturn().getTackResponses()
Confirm request = new Confirm();
request.setOrigin(origin); //origin is same as in getRequest
request.getMsgHashValues().addAll(msgHashValues);
//msgHashValues can be found in GetResponse.
request.getTAckContents().addAll(tackHashValues);
```

5.5 Call methods

5.5.1 WsAddressingHeader

For the asynchronous message a header needs to be provided which contains the type of message , and optionally other parameters (see documentation mycarenet)

There is a WsAddressingUtil that creates the header with the destination (optional, contains the identification number of the mutuality) and the type of message

Other parameters can be added on the object afterwards if needed

JAVA

```
import be.ehealth.business.mycarenetcommons.util.WsAddressingUtil;

WsAddressingHeader header = WsAddressingUtil.createHeader(mutualityNumber,
"urn:be:cin:nip:async:generic:get:query");

header.setTo(new URI(""));
header.setFaultTo("http://www.w3.org/2005/08/addressing/anonymous");
responseConfirmHeader.setReplyTo("http://www.w3.org/2005/08/addressing/anonymous"
);
header.setMessageID(new
URI(IdGeneratorFactory.getIdGenerator("uuid").generateId()));
```

5.5.2 post

JAVA

```
import be.ehealth.business.mycarenetcommons.util.WsAddressingUtil;
```

```
WsAddressingHeader header = WsAddressingUtil.createHeader(mutuality,
"urn:be:cin:nip:async:generic:post:msg");
PostResponse responsePost = service.postRequest(postRequest, header);
```

5.5.3 get

JAVA

```
import be.ehealth.business.mycarenetcommons.util.WsAddressingUtil;

WsAddressingHeader header = WsAddressingUtil.createHeader(null,
"urn:be:cin:nip:async:generic:get:query");
PostResponse responsePost = service.getRequest(getRequest, header);
```

5.5.4 confirm

JAVA

```
import be.ehealth.business.mycarenetcommons.util.WsAddressingUtil;

WsAddressingHeader responseConfirmHeader =
WsAddressingUtil.createHeader(mutuality, "
urn:be:cin:nip:async:generic:confirm:hash");
responseConfirmHeader.setTo(new URI(""));
responseConfirmHeader.setFaultTo("http://www.w3.org/2005/08/addressing/anonymous"
);
responseConfirmHeader.setReplyTo("http://www.w3.org/2005/08/addressing/anonymous"
);
responseConfirmHeader.setMessageID(new
URI(IdGeneratorFactory.getIdGenerator("uuid").generateId()));

PostResponse responsePost = service.postRequest(postRequest, header);
```

5.6 Handle and validate responses

5.6.1 Handle post response

JAVA

```
import be.ehealth.businessconnector.genericasync.builders.ResponseObjectBuilder;

ResponseObjectBuilder responseBuilder =
BuilderFactory.getResponseObjectBuilder();
boolean hasWarnings = responseBuilder.handlePostResponse(responsePost);
//if the result is not success, an exception is thrown, if its success but there are warnings , false is returned
```

5.6.2 Handle get response

JAVA

```
import be.ehealth.business.mycarenetcommons.domain.Blob;
import be.ehealth.business.mycarenetcommons.mapper.SendRequestMapper;
import be.ehealth.business.mycarenetcommons.builders.BlobBuilderFactory;


GetResponse responseGet = …;
// the get response contains multiple business and technical response messages
for (MsgResponse msgResponse : responseGet.getReturn().getMsgResponses()) {
   Blob mappedBlob = SendRequestMapper.mapToBlob(msgResponse.getDetail());
   byte[] unwrappedMessageByteArray =
BlobBuilderFactory.getBlobBuilder({projectName}).checkAndRetrieveContent(mappedBl
ob);
   //Handle the business response message here
}
for (TAckResponse tackResponse : responseGet.getReturn().getTAckResponses()) {
   byte[] tackResponseBytes = tackResponse.getTAck().getValue();
   // handle the technical response here
}
```

Configuration property

Configuration properties for BlobBuilder : see javadoc

5.6.3 Handle confirm response

ConfirmResponse is an empty object , no need to handle it.

6 Known limitations

6.1 Limitations of Java Architecture for XML Binding (JAXB)

The connector uses JAXB for the marshaling and unmarshalling of XML documents. It uses a standardized binding file so that all the JAXB objects are similar. The consequence of the use of this framework is only one version of an XSD is supported for each namespace. For example the KMERH xsd is used in the connector and must be the same for every business service. Otherwise some security exceptions are thrown at runtime.

6.2 Limitations of IKVM

The connector is written in Java but compiled for .NET C# using the IKVM framework (more information www.ikvm.net). The advantage of this choice is that there is only one code base, what is better for maintenance but it isn't native .NET code.

Here are some tricks and tips to speed up your development process

- Don't use the DEBUG mode of .NET
- Don't use the console logging of the logging framework
- Set the connector in DEBUG mode for logging.

If you take this in to account you will find all the info needed for debugging in the log file. Only the first time IKVM is loaded it takes some time (± 2 seconds) but after the initial loading it is fast.

For compiling this DDL a modified version of IKVM is used based on version 7.2.4630.6. In the standard version of IKVM javax.smartcardio is not implemented. Those classes are needed by the Belgian eID. In the eHealth version of IKVM those interfaces are removed and replaced with the ones provided in the technical connector.

6.3 Limitations of Public-Key Cryptography Standards #11 (PKCS11)

Only the 32bit JVM of Sun is shipped with an implementation of PKCSS11. So if you are using a 64bit version of the Sun JVM make sure that the sunpkcs11.jar is present on your classpath. Otherwise the use of PKCS11 as technology for eID reading and signing is not available.

6.4 Limitations of Personal Computer/Smart Card (PC/SC)

If the connector is used in a GNU/Linux environment it needs the installation of de pcsclite library. Otherwise the technology for eID reading and signing is not available. More information on this library could be found on the following address (http://pcsclite.alioth.debian.org/pcsclite.html)

6.5 Limitations of Connector

- The current version of the connector doesn't support multiple session within the same connector instance. Only one user is allowed for each session. If you want to perform a session switch you must do an unload session and create a new session.
- The eHealth platform only ensures the correct behavior by using the provided classpath inside the package. If you change any dependency (upgrade or downgrade) we aren't able to give any support.