

iDIN Software Libraries Integration Manual

Document version 1.046



Contents

Contents	2
1 Introduction	5
1.1 Target audience	5
1.2 Document scope	5
1.3 Revisions	5
1.3.1 Integration Manual.....	5
1.3.2 .NET Software Library	6
1.3.3 Java Software Library.....	7
1.3.4 PHP Software Library.....	7
1.4 Document contents	8
1.5 Other references	8
1.6 Definitions	8
2 Introduction to iDIN.....	9
2.1 Basic product description	9
2.2 iDIN Software Libraries overview	10
3 Generic integration guidelines	11
3.1 Certificates	11
3.1.1 .NET Certificate use	11
3.1.2 Java Certificate use	12
3.1.3 PHP Certificate use	13
3.2 .NET integration steps	13
3.2.1 Basic guidelines	13
3.2.2 Add a reference to the library	14
3.2.3 Configure the library	14
3.2.4 Write code using the library.....	15

3.3	Java integration steps	16
3.3.1	Additional extension software needed: JRE add-on	16
3.3.2	Basic guidelines	17
3.3.3	Add the library reference in the project	18
3.3.4	Create a configuration file in your project.....	19
3.3.5	Write code using the library.....	19
3.4	PHP integration steps	21
3.4.1	Required PHP extensions	21
3.4.2	Basic guidelines	21
3.4.3	Create a new PHP application project.....	21
3.4.4	Create a configuration file in the project.....	22
3.4.5	Reference autoload.php file and write code	22
4	Requirements	24
4.1	System Settings	24
4.2	Library Settings	24
4.2.2	.NET Library settings.....	25
4.2.3	Java Library settings	26
4.2.4	PHP Library settings.....	28
5	Integration Workflow.....	30
5.1	Retrieve Issuer List (Directory request)	30
5.1.1	.NET Directory request.....	31
5.1.2	Java Directory request	32
5.1.3	PHP Directory request.....	33
5.2	Issue a new Authentication request (Transaction request).....	34
5.2.1	.NET Authentication request	35
5.2.2	Java Authentication request.....	36
5.2.3	PHP Authentication request	37

5.3	Request the authentication Status (Status request)	38
5.3.1	.NET Status request.....	40
5.3.2	Java Status request.....	41
5.3.3	PHP Status request.....	42
6	Testing	44
7	Error Handling and Logging.....	45
7.1	.NET Logging	46
7.2	Java Logging.....	46
7.3	PHP Logging	47

1 Introduction

1.1 Target audience

This document is intended for developers who are responsible for integrating the iDIN Software Libraries using Microsoft .NET, Java or PHP development platforms. Merchants can use these libraries as an aid for implementing iDIN.

To learn more about iDIN and the data elements that are used, please consult the iDIN Merchant Implementation Guidelines (*also available in Dutch*) that is distributed together with this Integration Manual.

1.2 Document scope

The iDIN Software Libraries aim at facilitating an easy and flawless implementation of iDIN for Merchants (i.e. the exchange of messages between Merchant and their Routing Service).

The scope of this document is to support the integration of the iDIN Software Libraries with other components, such as web applications, online shops and websites, by providing examples and guidelines.

The iDIN Software Libraries are offered in three languages: .NET Framework 4.5, Java SE 7 and PHP 5.5. This document contains a generic description of these libraries, and provides code examples in each of the development languages.

1.3 Revisions

1.3.1 *Integration Manual*

Version	Description	Revision date
v1.00	Initial release	7 th of October 2015
v1.04	<ol style="list-style-type: none">1. Addition of two return parameters in the SAML Response. The libraries now return the <code>DeliveredServiceID</code> and a 2nd SAML status code as described in Section 5.3 of this document;2. Updated the version of PHP from 5.4 to 5.5. Note: No other versions are published between v1.00 and v1.04.	27 th of November 2015

Version	Description	Revision date
	This is due to levelling the version number with other documentation.	
v1.041	<ol style="list-style-type: none"> 1. Included revision tables in the manual; 2. Restructured Chapter 3; 3. Added more elaborate description for the parameters used in the configuration, see Section 4.2. 	22 nd of January 2016
v1.042	<ol style="list-style-type: none"> 1. Added rows in the revision tables for the updated .NET (v1.041) and Java (v1.041) Software Libraries 	22 nd of March 2016
v1.043	<ol style="list-style-type: none"> 1. Added row in the revision table for the updated .NET (v1.042) Software Library 	7 th of April 2016
v1.044	<ol style="list-style-type: none"> 1. Added row in the revision table for the updated PHP (v1.042) Software Library 	14 th of April 2016
v1.045	<ol style="list-style-type: none"> 1. Added row in the revision table for the updated PHP (v1.042) Software Library 	9 th of May 2016
v1.046	<ol style="list-style-type: none"> 1. Added rows in revision table for the updated PHP (v1.045) and .NET (v1.045) Software Libraries 	6 th of June 2016

Table 1: Software Libraries Integration Manual revision table

1.3.2 .NET Software Library

Version	Description	Revision date
v1.00	Initial release	5 th of August 2015
v1.04	<ol style="list-style-type: none"> 1. Addition of two return parameters in the SAML Response. The library now returns the <code>DeliveredServiceID</code> and a 2nd SAML status code as described in Section 5.3 of this document; 2. Possibility to separately request the gender attribute of the Consumer with the <code>RequestedServiceID</code>. Previously it was part of the attribute group name; 3. Fixed issue in handling of multiple namespaces in the Signature node of the Library; 4. Fixed issue in decrypting the attributes by adding ISO10126 padding to the <code>AesManaged</code> instance; 5. Fixed issue of not properly returning the SAML <code>NameID</code> element in the Response. 	27 th of November 2015
v1.041	<ol style="list-style-type: none"> 1. Fixed issue in the processing of optional <code>xsi:type</code> definitions for the encrypted SAML Attribute elements. 	22 nd of March 2016
v1.042	<ol style="list-style-type: none"> 1. Fixed issue of the validation on the <code>TransientID</code> by the Software Library being too strict. This led to errors in processing messages where non-alphanumeric characters were included in the <code>TransientID</code> 	7 th of April 2016
v1.045	<ol style="list-style-type: none"> 1. Fixed issue in messages sent by the library using local time instead of UTC in the <code>createDatetimestamp</code> element 2. Updated the .NET library to process specific namespace-handling inside the encrypted SAML Attribute elements which may be used by an Issuer 	3 rd of June 2016

Table 2: .NET Software Library revision table

1.3.3 Java Software Library

Version	Description	Revision date
v1.00	Initial release	5 th of August 2015
v1.04	<ol style="list-style-type: none"> 1. Addition of two return parameters in the SAML Response. The library now returns the <code>DeliveredServiceID</code> and a 2nd SAML status code as described in Section 5.3 of this document; 2. Possibility to separately request the gender attribute of the Consumer with the <code>RequestedServiceID</code>. Previously it was part of the attribute group name; 3. Fixed issue in HTTP header Charset UTF-8 having quotation marks instead of none. 	27 th of November 2015
v1.041	<ol style="list-style-type: none"> 1. Fixed issue in handling optional <code>X509SubjectName</code> element in the SAML Signature element. 	22 nd of March 2016

Table 3: Java Software Library revision table

1.3.4 PHP Software Library

Version	Description	Revision date
v1.00	Initial release	7 th of October 2015
v1.04	<ol style="list-style-type: none"> 1. Addition of two return parameters in the SAML Response. The library now returns the <code>DeliveredServiceID</code> and a 2nd SAML status code as described in Section 5.3 of this document; 2. Possibility to separately request the gender attribute of the Consumer with the <code>RequestedServiceID</code>. Previously it was part of the attribute group name; 3. Fixed issue in HTTP header Charset UTF-8 having quotation marks instead of none. 	27 th of November 2015
v1.041	<ol style="list-style-type: none"> 1. Fixed issue in the Issuer element inside the Authentication request being hardcoded to a value, instead of taking the <code>MerchantID</code> parameter from the configuration file. 	22 nd of January 2016
v1.042	<ol style="list-style-type: none"> 1. Fixed issue in not properly loading PHP classes on Linux systems due to case sensitivity of the file path to the Schema directory 2. Fixed issue of not properly loading the SAML certificate when the embedded certificate is on one line only 3. Fixed issue in failure of SAML signature validation when the status response message contains all namespaces prefixes declared in the root of the IDX message 	14 th of April 2016
v1.045	<ol style="list-style-type: none"> 1. Fixed issue in SAML signature validation caused by certain namespace prefixes being falsely identified as undefined 	18 th of May 2016

Table 4: PHP Software Library revision table

1.4 Document contents

Following this introductory chapter, this document describes the purposes of the iDIN Service and the generic integration and security guidelines for the iDIN Software Libraries.

Next, the main phases of the integration workflow are described, which is the environment setup, development and testing, and code examples for the main functions in each of the development languages.

1.5 Other references

Title	Version	Issued by
iDIN Merchant Implementation Guidelines (<i>also available in Dutch</i>)	1.041	Betaalvereniging Nederland

1.6 Definitions

The following terms are used in this document, related to the iDIN service:

- The Merchant: The owner of a web application in which iDIN is an authentication method, and the user of the Software Libraries;
- The Acquirer: The bank with which the Merchant has a contract for iDIN;
- Routing Service: The technical role of the Acquirer which handles the message exchange;
- The Consumer: The customer who uses iDIN provided by the Merchant;
- The Issuer: The bank that communicates on behalf of the Consumer;
- Validation Service: The technical role of the Issuer that handles the message exchange.

2 Introduction to iDIN

This chapter contains a brief description of iDIN and the underlying iDx protocol. A more elaborate description can be found in the iDIN Merchant Implementation Guidelines (also available in Dutch), which is distributed along with this integration manual.

2.1 Basic product description

. The iDIN services can be used for the following:

- Identification and authentication of Consumers, based on Issuer issued credentials;
- Provisioning of Consumer attributes from the Issuer administration;
- Consumer age verification.

Core principles of iDIN:

- Consumer consent at the Issuer side is required explicitly for all iDIN services and provisioning of Consumer data, for every request initiated by the Merchant;
- Consumers can be persistently identified using a Bank Identification Number (BIN), which will be Issuer generated (so not shared among Issuers) and unique per Consumer-Merchant combination; however when only attribute provisioning is requested by the Merchant, a transient identifier is received, that is just valid for one session;
- A Consumer has only one identity per Issuer, regardless of the number of credentials;
- Consumer information in messages will be shielded from Acquirers by means of end-to-end encryption.

iDIN uses the iDx standards as a messaging standard. It uses the generic information container in the iDx protocol to embed SAML 2.0 messages. From the Merchant's perspective, iDIN uses the three iDx functions:

- **Directory Request:** This request is run between the Merchant and Routing Service. Its purpose is to retrieve the list of Issuers offering iDIN;
- **Transaction Request:** This is run between the Merchant and Routing Service to process an authentication request;

- **Status Request:** This is run between the Merchant and Routing Service to retrieve the status of a previously processed authentication request, and if it is successful - also retrieve the values of requested data.

2.2 iDIN Software Libraries overview

The iDIN Software Libraries are a set of software libraries created to facilitate the integration of Merchants for supporting iDIN messages based on the iDx standards. The libraries are responsible for validating and authenticating incoming messages against its XML schema and certificates. The libraries save Merchants the need to implement all functionality described in the iDIN product and iDx standards.

The iDIN Software Libraries expose the following operations:

- Retrieve the list of the Issuers (implements a Directory Request at iDx level);
- Send an authentication request (implements a Transaction Request at iDx level);
- Retrieve the status of an earlier authentication request (implements a Status Request at iDx level).

Note: In this document the iDIN process of providing one of the above services is referred to as a iDIN transaction.

3 Generic integration guidelines

When integrating the iDIN Software Libraries into your solution, the guidelines in the upcoming sections should be followed.

3.1 Certificates

In order to be able to function, the library needs to have access to three certificates:

1. **Merchant's certificate:** This is the private certificate used to sign messages sent by the Merchant to the Acquirer's Routing Service platform. Its public key is also used by the Acquirer to authenticate incoming messages from the Merchant;
2. **The SAML certificate:** This is the certificate owned by the Merchant, and its public key is used by the Issuer to encrypt information. The Merchant can then use the private key to decrypt that information;
3. **The Acquirer's public certificate:** This is the certificate used to authenticate incoming messages from the Acquirer. The library only needs its public key.

The certificate that is used to sign the messages (1), and the certificate that is used by the Issuer to encrypt information (2) may be the same certificate. Hence, depending on your preferences you can use two or the same certificate for both (1) and (2).

Note: How to obtain these certificates is outside the scope of this manual, more details are available in the iDIN Merchant Implementation Guidelines.

3.1.1 .NET Certificate use

By default, the library will try to load the certificates from the Certificate Store, using the fingerprints specified in the configuration. The library will first search the Local Machine store, followed by the Current User certificate store. In order to succeed, the host application's identity must have at least read permissions for the certificates.

Loading the certificates from locations other than the Certificate Store is possible as the Configuration class has 4 constructors taking in different versions of parameters.

Please note that, for Windows .NET environments, the signing certificate should have a cryptographic service provider (CSP) reference to Microsoft Enhanced RSA and AES Cryptographic Provider to support the algorithms required by iDIN and the underlying iDx protocol

3.1.2 *Java Certificate use*

Java certificates are stored in Java keystores, which are files on the disk – they are protected by a password and have the `.jks` extension. These certificates need to be imported into the keystore using the `keytool` application, installed as part of the Java Development Kit. Creating and managing keystores is outside of the scope of this manual, please consult the Oracle documentation on the following link: <https://docs.oracle.com/javase/7/docs/technotes/tools/#security>. There, more elaborate documentation on this tool can be found. Below are some examples on how to use the `keytool` application for a few common scenarios:

1. Importing an existing certificate into a keystore and assigning it an alias:

```
keytool -import -alias "server" -file server.cer -keystore .\certificates.jks
```

2. Listing certificates in a keystore:

```
keytool -list -keystore .\certificates.jks
```

3. Importing a PKCS12 certificate, together with the private key, into a keystore:

```
keytool -importkeystore -destkeystore .\certificates.jks -srckeystore  
privatekey.p12 -srcstoretype pkcs12
```

The Java iDIN Software Library needs to access a keystore located in the Java classpath. Furthermore, the private key of a certificate can also be protected by a password. As such, the Java library has three more configuration options:

- **keystore location:** A file path and name, accessible to the library, which is the Java keystore where the certificates are stored;
- **keystore password:** The password used to access the keystore;
- **signing certificate password:** The password used to access the private key of the signing certificate.

All the certificates need to be imported in the same keystore. When doing that, you must also provide an alias for each of them, and while the .NET version uses certificate thumbprints to search for existing certificates, the Java version uses these aliases. So certificate thumbprints in the .NET version are certificate aliases in the Java version:

- **Signing certificate alias:** The alias assigned to the signing certificate in the keystore. This could be the alias you supplied explicitly when importing an existing certificate in the keystore, or it could be an alias automatically assigned by the keytool application.
- **Acquirer's certificate alias:** The alias assigned to the Acquirer's certificate in the keystore. This could be the alias you supplied explicitly when importing an existing certificate in the keystore, or it could be an alias automatically assigned by the keytool application.
- **SAML certificate alias:** The alias assigned to the SAML certificate in the keystore. This could be the alias supplied explicitly when importing an existing certificate in the keystore, or it could be an alias automatically assigned by the keytool application.

3.1.3 *PHP Certificate use*

For PHP, the Merchant certificate has to be in PKCS#12 format (typically with .p12 or .pfx extensions) and the Acquirer's certificate in PEM format (typically with .pem or .cer extensions). They need to be located in a path that is readable by the web server and PHP interpreter. The php.ini can be modified to add the directory or directories where the certificates are located to the global PHP includes (<http://php.net/manual/en/ini.core.php#ini.include-path>), in order to only specify the file name in the configuration file. As such, the PHP version of the library needs path names for configuration options related to certificates:

- **Merchant certificate file:** The file name to be used as Merchant certificate (PKCS#12 format);
- **Acquirer's certificate file:** The file name to be used as Acquirer certificate (PEM format);
- **SAML certificate file:** The file name to be used as SAML certificate (PKCS#12 format).

3.2 .NET integration steps

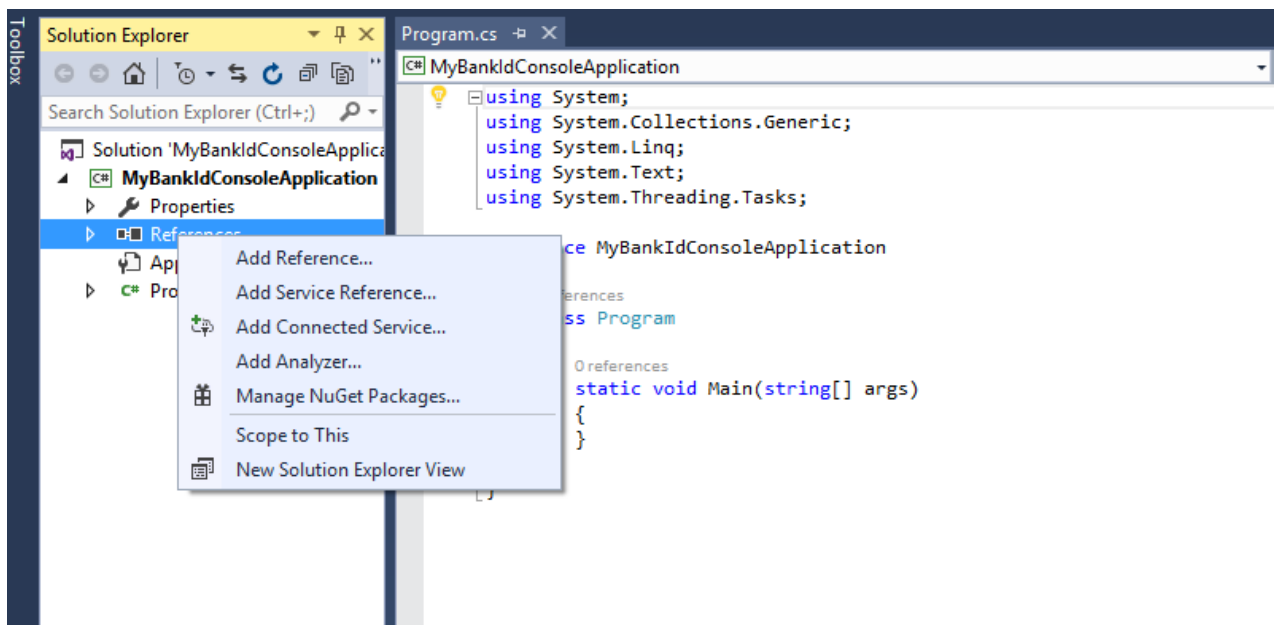
3.2.1 *Basic guidelines*

The package has the following noteworthy files:

- **BankID.Merchant.Library.dll**: This file is the actual library, which can be used to integrate into the code. More details in the next paragraph;
- **BankID.Merchant.Library.dll.config**: This file contains an example of configuration settings that need to be provided in order for the library to function correctly. Some of these settings (such as BankID.Contract.Id) have a specific value, and are received from the Acquirer;
- **BankID.Merchant.Library.xml**: This file contains Documentation Comments for the library public classes and methods. When the library is integrated into projects, users will be able to access, for example, the Visual Studio Intellisense feature to give quick documentation.

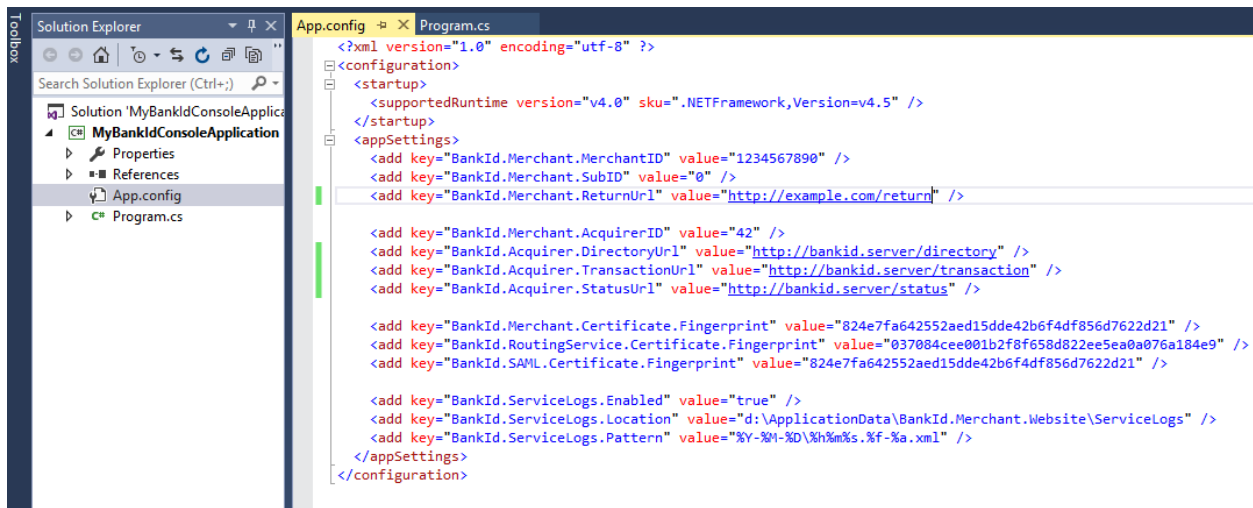
First, an existing project is required (for example, an ASP.NET web site, or a simple C# Console Application) into which the library can be added. Then, unpack the library files into a designated folder. A subfolder of your project would be a good choice. After that, you need to provide the configuration settings in either the App.config or Web.config of your project. Below is an example that shows how to do that in Visual Studio:

3.2.2 Add a reference to the library



3.2.3 Configure the library

Create a configuration section and change settings as needed (this screenshot shows only an example):



3.2.4 Write code using the library

The library is now ready to perform requests. Chapter 5 in this manual provides more information on how to perform these operations.

The following code snippet can be copied/pasted into a new console application project to start using the library right away. Make sure the namespace and class names are equal to the names chosen when the project was set-up, and change the configuration settings as necessary (check Section 4.2 in this manual for more information about specific settings):

```

using BankID.Merchant.Library;
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyBankIDConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            BankID.Merchant.Library.Configuration.Load();

            var communicator = new Communicator();
            var directoryResponse = communicator.GetDirectory();

```

```
        if (directoryResponse == null)
            Console.WriteLine("Directory Response is null");

        if (directoryResponse.IsError)
        {
            Console.WriteLine(String.Format("Received ErrorCode: {0}",
directoryResponse.Error.ErrorCode));
            Console.WriteLine(String.Format("Received ErrorDetails: {0}",
directoryResponse.Error.ErrorDetails));
            Console.WriteLine(String.Format("Received ErrorMessage: {0}",
directoryResponse.Error.ErrorMessage));

            return;
        }

        foreach (var issuer in directoryResponse.Issuers)
        {
            Console.WriteLine(issuer.Country + " " +
                               issuer.Id + " " +
                               issuer.Name);
        }

        Console.ReadLine();
    }
}
```

3.3 Java integration steps

3.3.1 *Additional extension software needed: JRE add-on*

By default, the Java Runtime Environment does not support AES 256-bit encryption, due to import control restrictions of some countries. The iDIN protocol requires AES 256-bit encryption support. Therefore the add-on JRE, called "Unlimited Strength Jurisdiction Policy Files", needs to be downloaded and installed.

This is available at the official oracle.com website. Make sure the correct version is downloaded and installed. For Java 7, the necessary files can be downloaded here:

<http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html> Please refer to the README.txt in the downloaded file for more information about how to install it. Below is an extract from the Readme file, giving information on the installation process:

- 1) Download the unlimited strength JCE policy files.
- 2) Uncompress and extract the downloaded file.

This will create a subdirectory called jce.

This directory contains the following files:

README.txt	This file
local_policy.jar	Unlimited strength local policy file
US_export_policy.jar	Unlimited strength US export policy file

- 3) Install the unlimited strength policy JAR files.

In case you later decide to revert to the original "strong" but limited policy versions, first make a copy of the original JCE policy files (US_export_policy.jar and local_policy.jar). Then replace the strong policy files with the unlimited strength versions extracted in the previous step.

The standard place for JCE jurisdiction policy JAR files is:

<java-home>/lib/security	[Unix]
<java-home>\lib\security	[Windows]

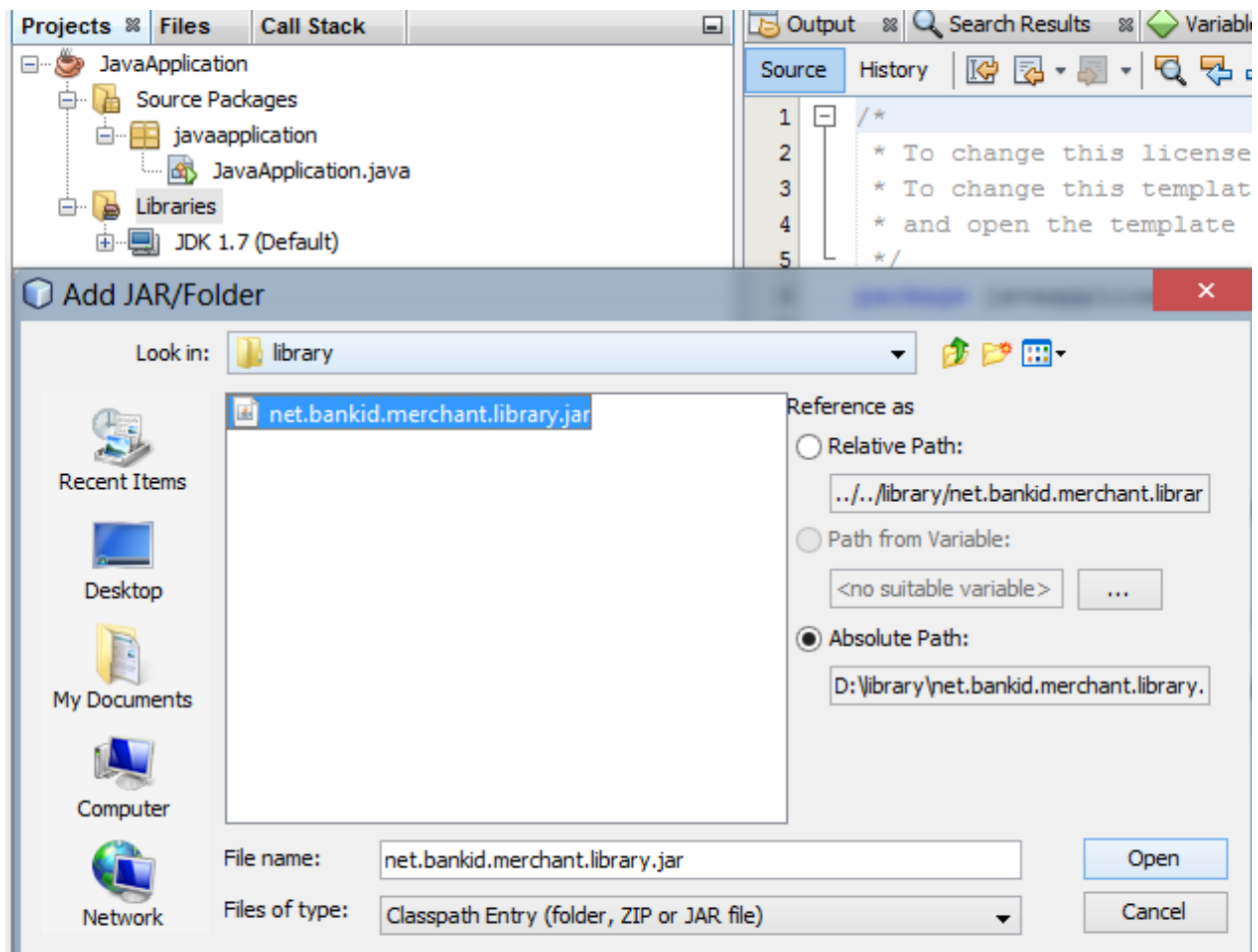
3.3.2 *Basic guidelines*

The package has the following noteworthy files:

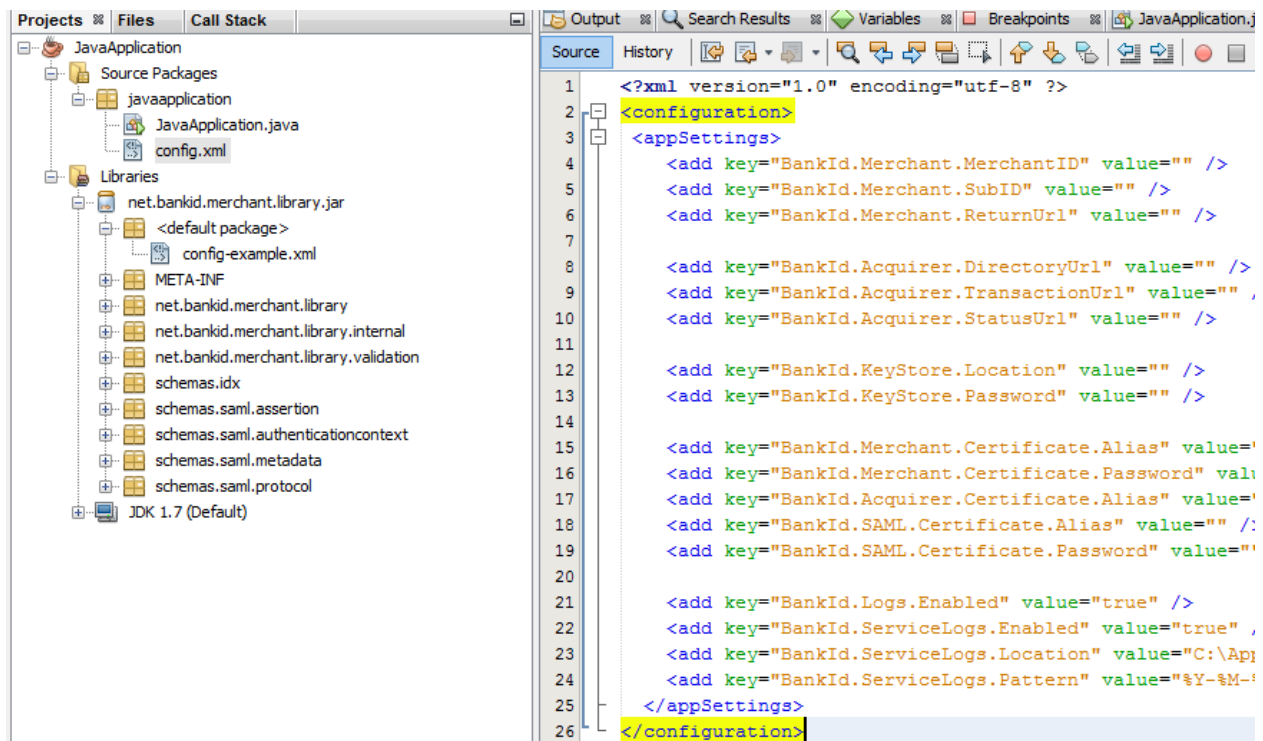
- **net.BankID.merchant.library.jar:** This file is the actually library itself, which can be integrated into the code. More details in the next paragraph;
- **BankID -config-example.xml:** This file contains an example of configuration settings that must be provided in order for the library to function correctly. Some of these settings (such as BankID.Merchant.MerchantID) have a specific value that are received from the Acquirer.

First, there has to be an existing project (for example, a Java web application) into which the library can be added. Subsequently, the library files must be unpacked into some folder. A subfolder of the project would be a good choice. After that, a configuration file needs to be created in the project, so the library can load your desired settings. Below are a few screenshots which show how to do that in Netbeans:

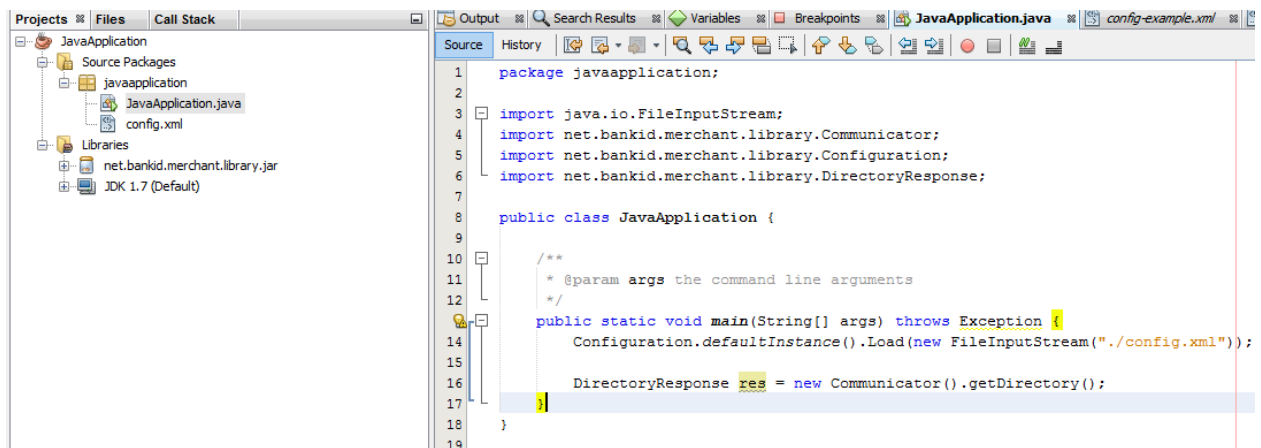
3.3.3 Add the library reference in the project



3.3.4 Create a configuration file in your project



3.3.5 Write code using the library



The library can now be used to perform request. Chapter 5 in this manual provides more information on how to perform these operations.

The following code snippet can copied/pasted into a new console application project to start using the library right away. Make sure the namespace and class names are equal to the names chosen when the project was set-up, and change the configuration settings as necessary (check Section 4.2 in this manual for more information about specific settings):

```
package console;

import java.io.IOException;
import net.BankID.merchant.library.Configuration;
import net.BankID.merchant.library.Communicator;
import net.BankID.merchant.library.DirectoryResponse;

public class Console {
    public static void main(String[] args) throws IOException {
        Configuration.defaultInstance().Setup(new Configuration(
            "1111111111"
            , 0
            , "https://my.web.shop/transaction/return"
            , "keystore.jks"
            , "keystore_password"
            , "merchant_certificate_alias"
            , "privatekey_password"
            , "routing_service_certificate_alias"
            , "https://my.bank/url_for_directory_requests"
            , "https://my.bank/url_for_transaction_requests"
            , true
            , true
            , "D:\\My.Logs\\ServiceLogs"
            , "%Y-%M-%D\\%h%m%s.%f-%a.xml"
        ));

        DirectoryResponse res= new Communicator().getDirectory();
        if (res.getIsError()) {
            System.out.println(res.getErrorResponse().getErrorMessage());
        }
        else {
            for (DirectoryResponse.Issuer issuer : res.getIssuers ()) {
                System.out.println(issuer.getIssuerCountry() + " " +
                                    issuer.getIssuerId() + " " +
                                    issuer.getIssuerName());
            }
        }
    }
}
```

3.4 PHP integration steps

3.4.1 *Required PHP extensions*

The PHP version of the iDIN Merchant Library requires three PHP extensions to be enabled:

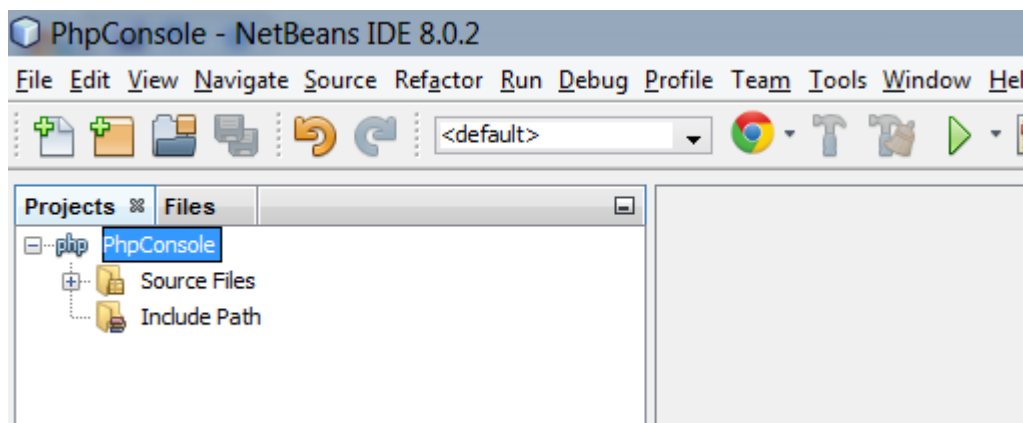
- **php_curl:** This is required so that the library can make HTTP requests;
- **php_openssl:** This is required for the library to correctly handle loading certificates, signing messages and checking the signatures of incoming message. OpenSSL 1.0 or higher should be used;
- **php_mcrypt:** this is required for decryption routines (some fields in the status response are encrypted using AES-256).

These extensions can all be enabled in the php.ini file.

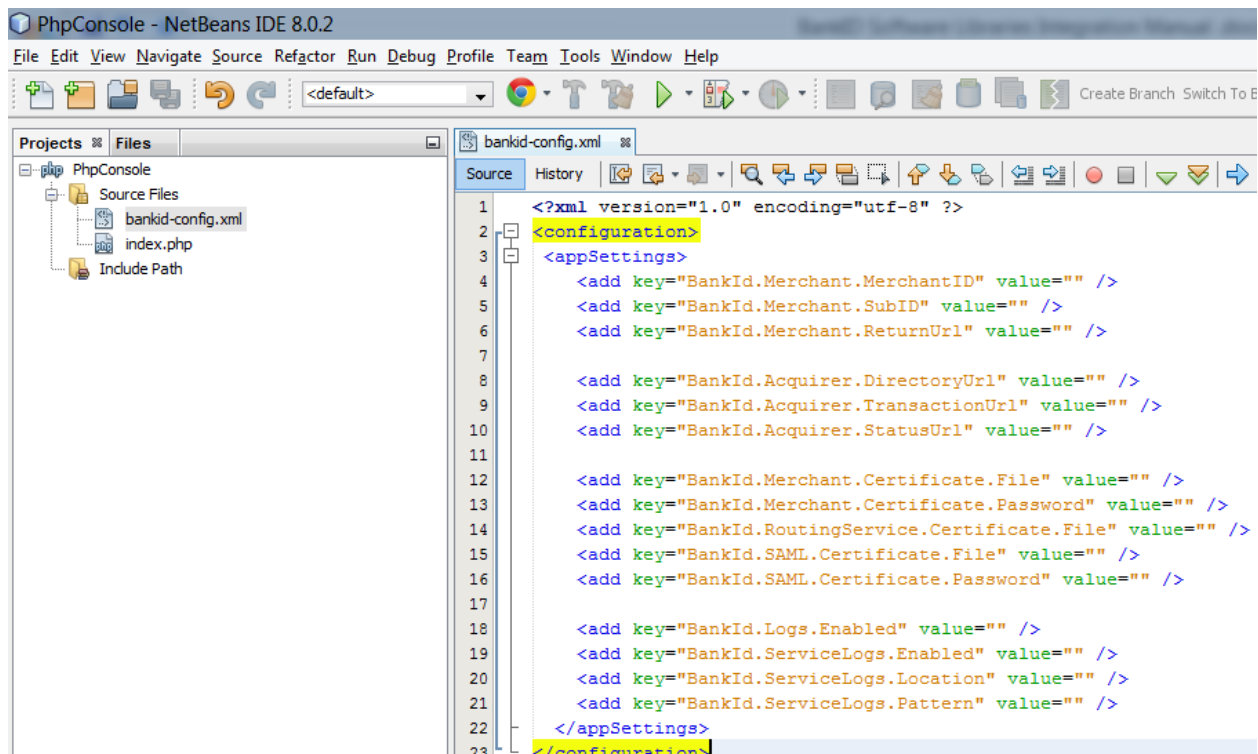
3.4.2 *Basic guidelines*

In the package, there is a directory named "Library/PHP". This directory can be copied to any directory on the disk that is accessible by the webserver, for example "C:\php\bankid_library". In the PHP project, just require "C:\php\php_library\vendor\autoload.php" and the classes in the library can be directly used. Also, add an XML configuration file to the project, using the same structure as .NET or Java. Below are a few screenshots which show how to do this in Netbeans:

3.4.3 *Create a new PHP application project*

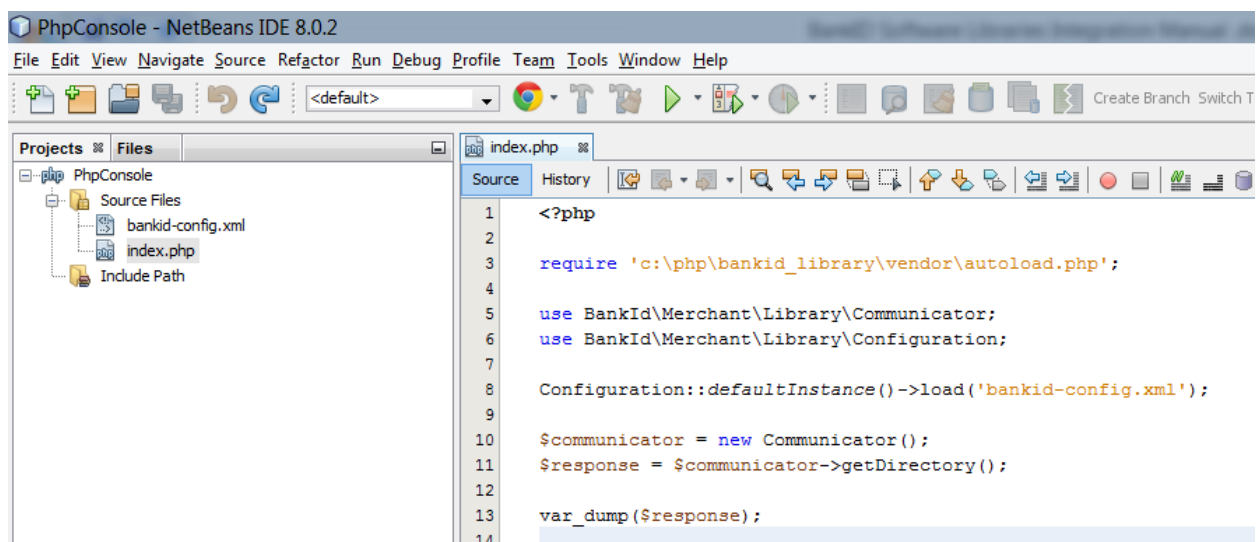


3.4.4 Create a configuration file in the project



3.4.5 Reference autoload.php file and write code

Require LIBRARY_DIRECTORY/vendor/autoload.php and write code using the library:



The library can now be used to perform requests. Chapter 5 in this manual provides more information on how to perform these operations.

The following code snippet can be copied/pasted into a new console application project to start using the library right away. Make sure the namespace and class names correspond to the names chosen when the project was set-up, and change the configuration settings as necessary (check Section 4.2 in this manual for more information about specific settings)

```
<?php
```

```
require 'c:\php\bankid_library\vendor\autoload.php';

use BankId\Merchant\Library\Communicator;
use BankId\Merchant\Library\Configuration;

$config = new Configuration();
$config->set(
    '111111111',
    '0',
    'https://my.web.shop/transaction/return',
    'https://my.bank/url_for_directory_requests',
    'https://my.bank/url_for_transaction_requests',
    'https://my.bank/url_for_status_requests',
    'my_certificate.p12',
    'routing_service_certificate.cer',
    'my_saml_certificate.p12',
    NULL,
    'password_for_my_certificate_private_key',
    NULL,
    NULL,
    'password_for_my_saml_certificate_private_key',
    'D:\My.Logs\ServiceLogs',
    TRUE,
    '%Y-%M-%D\%h%m%s.%f-%a.xml');

Configuration::defaultInstance()->setup($config);

$communicator = new Communicator();
$response = $communicator->getDirectory();

if ($response->getIsError()) {
    print 'error = ' . $response->getErrorResponse()->getMessage();
}
else {
    foreach ($response->getIssuers() as $issuer) {
        print $issuer->getCountry() . ' ' . $issuer->getName() . ' ' .
        $issuer->getID();
    }
}
```

4 Requirements

This chapter describes the settings that are required for the successful integration of the iDIN Software Libraries into the Merchant's system.

4.1 System Settings

The following pre-requisites are needed on the system before integrating the iDIN Software Libraries:

- It must be possible to make a TLS connection to the Routing Service's web service from the web server through port 443.
- It must be possible to access the local machine certificate store during the installation of the certificates on the web server. The merchant certificate private key needs to be exportable.

4.2 Library Settings

The configuration of the libraries is possible at runtime during initialization or via a configuration file (development platform specific). The following parameters are configurable in the libraries configuration (not all parameters are used in all libraries):

Parameter	Description
<code>Merchant.MerchantID</code>	This is the contract number for iDIN the Merchant received from its Acquirer after registration, and is used to unambiguously identify the Merchant. This number is 10-digits long, where the first four digits are equal to the <code>AcquirerID</code> . Leading zeros must be included.
<code>Merchant.SubID</code>	Unless agreed otherwise with the Acquirer, the Merchant has to use 0. The <code>SubID</code> that uniquely defines the name and address of the Merchant to be used for iDIN, if operating under different brands or trading entities. The Merchant obtains the <code>SubID</code> from its Acquirer after registration for iDIN. A Merchant can request permission from the Acquirer to use one or more <code>SubIDs</code> .
<code>Merchant.ReturnURL</code>	The web address provided by the Merchant in the transaction request that is used to redirect the Consumer back to the Merchant after completing the authentication in the Issuer domain. The URL does not necessarily begins with <code>http://</code> or <code>https://</code> , it can also start with an app handler e.g. <code>companyname-nl-service://</code> .
<code>Merchant.Certificate.Fingerprint</code>	The fingerprint of the certificate used to sign messages sent by the Merchant to the Acquirer's Routing Service platform. It is also used by the Acquirer to authenticate incoming messages from the Merchant.
<code>SAML.Certificate.Fingerprint</code>	The fingerprints of the certificate of the Merchant used to decrypt the SAML Response.
<code>Acquirer.AcquirerID</code>	Unique four-digit identifier of the Acquirer. These are equal to the first four digits of the <code>MerchantID</code> . Leading zeros must be included.

Parameter	Description
Acquirer.Certificate.Fingerprint	The fingerprint of the certificate used to validate incoming messages from Acquirer to the Merchant's application.
Acquirer.DirectoryURL	The web address of the Acquirer's Routing service platform from where the list of Issuers is retrieved (using a directory request). Note: The directory URL, transaction URL and status URL can be the same.
Acquirer.TransactionURL	The web address of the Acquirer's Routing Service platform where the transactions (authentication requests) are initiated. Note: The directory URL, transaction URL and status URL can be the same.
Acquirer.StatusURL	The web address of the Acquirer's Routing Service platform to where the library sends status request messages. Note: The directory URL, transaction URL and status URL can be the same.
Logs.Enabled	Enable the saving of service logs (all raw messages exchanged by the library). Either "true" or "false".
ServiceLogs.Enabled	Enable the saving of service logs (all raw messages exchanged by the library). Either "true" or "false".
ServiceLogs.Location	The disk location where the request messages sent by the library and the response messages received from the Acquirer will be logged.
ServiceLogs.Pattern	A string describing what the file names of the service logs should look like, so they can be organized in different folders, by year, month, day, time of day and type of message. As an example pattern in the form of %Y-%M-%D\%h%m%s.%f-%a.xml will create a directory for each day (in the form on 2014-01-20), and a file for each operation in the form of 102045.924-AcquirerTrxReq.xml. When creating a pattern, you can use the following placeholders: %Y = current year, %M = current month, %D = current day, %h = current hour, %m = current minute, %s = current second, %f = current millisecond and %a = current action.

Table 5: Parameters used in the configuration of the libraries

In the next sections the approach for the configuration is explained, on a platform-by-platform basis.

4.2.2 .NET Library settings

Sample configuration file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <appSettings>
    <add key="BankID.Merchant.MerchantID" value="0042567890" />
    <add key="BankID.Merchant.SubID" value="0" />
    <add key="BankID.Merchant.ReturnUrl" value="http://example.com/return" />

    <add key="BankID.Merchant.AcquirerID" value="0042" />
    <add key="BankID.Acquirer.DirectoryUrl" value="http://BankID.server/directory" />
    <add key="BankID.Acquirer.TransactionUrl" value="http://BankID.server/transaction" />
    <add key="BankID.Acquirer.StatusUrl" value="http://BankID.server/status" />
  </appSettings>
</configuration>
```

```

    <add key="BankID.Merchant.Certificate.Fingerprint"
value="824e7fa642552aed15dde42b6f4df856d7622d21" />
    <add key="BankID.RoutingService.Certificate.Fingerprint"
value="037084cee001b2f8f658d822ee5ea0a076a184e9" />
    <add key="BankID.SAML.Certificate.Fingerprint"
value="824e7fa642552aed15dde42b6f4df856d7622d21" />

    <add key="BankID.ServiceLogs.Enabled" value="true" />
    <add key="BankID.ServiceLogs.Location"
value="d:\ApplicationData\BankID.Merchant.Website\ServiceLogs" />
    <add key="BankID.ServiceLogs.Pattern" value="%Y-%M-%D\%h%m%s.%f-%a.xml" />
</appSettings>
</configuration>

```

Usage:

- To load settings from configuration file:

```
Configuration.Load();
```

- To setup configuration from code:

```

BankID.Merchant.Library.Configuration.Setup(
    new BankID.Merchant.Library.Configuration(
        "acquirerId",
        "merchantId",
        new Uri("merchantReturnUrl"),
        new Uri("acquirerDirectoryUrl"),
        new Uri("acquirerTransactionUrl"),
        new Uri("acquirerStatusUri"),
        "merchantCertificateFingerprint",
        "routingServiceCertificateFingerprint",
        "samlCertificateFingerprint",
        "serviceLogsLocation",
        true,
        "serviceLogsPattern"));

```

4.2.3 Java Library settings

Class that handles configuration settings (see Section 3.3 for more information about some settings which are different from the .NET version):

```

public class Configuration {
    private String merchantID;
    private int merchantSubID;
    private String merchantReturnUrl;
    private String keyStoreLocation;
    private String keyStorePassword;
    private String merchantCertificateAlias;

```

```

private String merchantCertificatePassword;
private String acquirerCertificateAlias;
private String acquirerDirectoryURL;
private String acquirerTransactionURL;
private String acquirerStatusURL;
private boolean serviceLogsEnabled;
private String serviceLogsLocation;
private String serviceLogsPattern;
}

```

Sample configuration file:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
  <add key="BankID.Merchant.MerchantID " value="0123456789" />
  <add key="BankID.Merchant.SubID " value="0" />
  <add key="BankID.Merchant.ReturnUrl" value="https://webshop.nl/return" />
  <add key="BankID.KeyStore.Location" value="certificatestore.jks" />
  <add key="BankID.KeyStore.Password" value="password" />
  <add key="BankID.Merchant.Certificate.Alias " value="client" />
  <add key="BankID.Merchant.Certificate.Password" value="password" />
  <add key="BankID.Acquirer.Certificate.Alias" value="server" />
  <add key="BankID.Acquirer.DirectoryUrl" value="https://my.bank.nl/directory" />
  <add key="BankID.Acquirer.TransactionUrl" value="https://my.bank.nl/trx" />
  <add key="BankID.Acquirer.StatusUrl" value="https://my.bank.nl/status" />
  <add key="BankID.Logs.Enabled" value="true" />
  <add key="BankID.ServiceLogs.Enabled" value="true" />
  <add key="BankID.ServiceLogs.Location" value="D:\WebShop\ServiceLogs" />
  <add key="BankID.ServiceLogs.Pattern" value="%Y-%M-%D\%h%m%s.%f-%a.xml" />
</appSettings>
</configuration>

```

Usage:

- To load settings from configuration file:

```

Configuration.defaultInstance()
    .Load(
        getServletContext().getResourceAsStream("/config.xml")
    );

```

- To setup configuration from code:

```

Configuration.defaultInstance().Setup(new Configuration(
    "0000000000"
    , 0
    , "https://web.shop/return"
    , "certificatestore.jks"
    , "password"

```

```

        , "client"
        , "password"
        , "server"
        , "https://my.bank.nl/directory"
        , "https://my.bank.nl/trx"
        , "https://my.bank.nl/status"
        , true
        , true
        , "D:\\WebShop\\Logs"
        , "%Y-%M-%D\\%h%m%s.%f-%a.xml"
    );

```

4.2.4 *PHP Library settings*

Class that handles configuration settings (see Section 3.4 on more information about some settings which are different from the .NET version):

```

class Configuration {
    public $MerchantID;
    public $MerchantSubID;
    public $MerchantReturnUrl;
    public $AcquirerDirectoryUrl;
    public $AcquirerTransactionUrl;
    public $AcquirerStatusUrl;
    public $MerchantCertificateFile;
    public $RoutingServiceCertificateFile;
    public $SamlCertificateFile;
    public $MerchantCertificate;
    public $MerchantCertificatePassword;
    public $RoutingServiceCertificate;
    public $SamlCertificate;
    public $SamlCertificatePassword;
    public $LogsEnabled;
    public $ServiceLogsLocation;
    public $ServiceLogsEnabled;
    public $ServiceLogsPattern;
}

```

Sample configuration file:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
    <add key="BankId.Merchant.MerchantID" value="0123456789" />
    <add key="BankId.Merchant.SubID" value="0" />
    <add key="BankId.Merchant.ReturnUrl" value="https://webshop.nl/return" />
    <add key="BankId.Merchant.Certificate.File" value="certificate.p12" />
    <add key="BankId.Merchant.Certificate.Password" value="password" />
    <add key="BankId.RoutingService.Certificate.File" value="server.cer" />
    <add key="BankId.SAML.Certificate.File" value="saml.p12" />

```

```
<add key="BankId.SAML.Certificate.Password" value="password" />
<add key="BankId.Acquirer.DirectoryUrl" value="https://my.bank.nl" />
<add key="BankId.Acquirer.TransactionUrl" value="https://my.bank.nl" />
<add key="BankId.Acquirer.StatusUrl" value="https://my.bank.nl" />
<add key="BankId.Logs.Enabled" value="true" />
<add key="BankId.ServiceLogs.Enabled" value="true" />
<add key="BankId.ServiceLogs.Location" value="D:\WebShop\ServiceLogs" />
<add key="BankId.ServiceLogs.Pattern" value="%Y-%M-%D/%h%m%s.%f-%a.xml" />
</appSettings>
</configuration>
```

Usage:

To load settings from configuration file:

```
Configuration::defaultInstance()->load('bankid-config.xml')
```

To setup configuration from code:

```
$config = new Configuration();
$config->set(
    '111111111',
    '0',
    'https://my.web.shop/transaction/return',
    'https://my.bank/url_for_directory_requests',
    'https://my.bank/url_for_transaction_requests',
    'https://my.bank/url_for_status_requests',
    'my_certificate.p12',
    'routing_service_certificate.cer',
    'my_saml_certificate.p12',
    NULL,
    'password_for_my_certificate_private_key',
    NULL,
    NULL,
    'password_for_my_saml_certificate_private_key',
    'D:\My.Logs\ServiceLogs',
    TRUE,
    '%Y-%M-%D/%h%m%s.%f-%a.xml');
Configuration::defaultInstance()->setup($config);
```

5 Integration Workflow

This chapter contains integration guidelines and sample code for each of the main operations specific to the iDIN service.

To facilitate an integration process, the sample code from the package illustrates how a web application could integrate with an iDIN library. Parts of this code are presented below as an example for each of the iDIN service specific operations.

The sample code is a basic website integration with the libraries where the user can choose an Issuer, initiate a transaction and request the transaction status.

5.1 Retrieve Issuer List (Directory request)

The Directory request involves querying the Acquirer, so it returns a list of Issuers. A customer on the website will then select its Issuer he/she has an account with, and the process continues. When issuing the request, no additional parameters have to be specified.

It is not allowed to perform the Directory protocol for each transaction. Because the list of Issuers only changes occasionally, it is sufficient to execute the Directory protocol on a weekly basis and check if the list has changed based on the `DirectoryDateTimestamp`. If the Issuer list has changed, the latest version has to be saved and used for any subsequent transaction. Routing Services will normally also inform all Merchants (e.g. by email) about changes in their Issuer list. The Directory protocol should at least be executed once a week.

The response will include the following parameters:

- `IsError`: Field that allows to check if an error occurred;
- `Error`: An `ErrorResponse` object containing error information, if one occurred;
- `DirectoryDateTimestamp`: This field contains the date when the directory was last updated by the Acquirer's Routing Service;
- `Issuers`: The returned list of Issuers, each described by an identifier, country name and bank name;
- `RawMessage`: The actual XML message received from the Routing Service.

Examples are presented below on of how to perform a directory request, check the response and display the result on each of the supported platforms.

5.1.1 *.NET Directory request*

To perform a directory request, instantiate a `Communicator` instance and call the `GetDirectory()` method:

```
var response = new Communicator().GetDirectory();

if (response.IsError)
{
    // show or log error message
}
else
{
    // handle response: display list of issuing banks
}
```

Here is the structure of the directory response:

```
public class DirectoryResponse
{
    public bool IsError { get; }
    public ErrorResponse Error { get; }
    public DateTime DirectoryDateTimeStamp { get; }
    public Issuer[] Issuers { get; }
    public string RawMessage { get; }
}
```

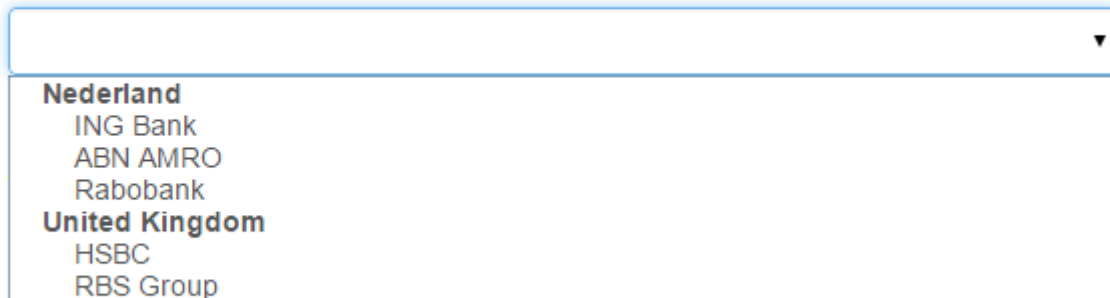
An example on how to display the list of issuing banks in ASP.NET Razor views is presented below, once the Issuer list is retrieved:

```
@model BankID.Merchant.Library.SampleWebsite.Models.DirectoryModel
```

```
@Html.DropDownListFor(m => m.Issuers, new SelectList(Model.Issuers, "Id", "Name",
"Country", 0), new { @class = "form-control", @name = "issuer" })
```

This will generate HTML mark-up for a drop-down list with all the available Issuers, grouped by Country:

Select the issuing bank



The image shows a web form with a dropdown menu. The dropdown is open, displaying a list of banks. The list is organized into two main sections: 'Nederland' and 'United Kingdom'. Under 'Nederland', the banks listed are 'ING Bank', 'ABN AMRO', and 'Rabobank'. Under 'United Kingdom', the banks listed are 'HSBC' and 'RBS Group'.

Figure 1 – Issuers list example

When the Consumer selects an Issuer from this drop-down list, an Authentication Request can be performed, which is shown in the next step.

5.1.2 *Java Directory request*

To perform a directory request, instantiate a `Communicator` object, and call the `getDirectory()` method:

```
DirectoryResponse response = new Communicator().getDirectory();
if (response.getIsError()) {
    // log error, show error message
}
else {
    // handle response: display list of issuing banks
}
```

Here is the structure of the directory response:

```
public class DirectoryResponse {
    public boolean getIsError();
    public ErrorResponse getErrorResponse();
    public XMLGregorianCalendar getDirectoryDateTimestamp();
    public final List<Issuer> getIssuers();
    public String getRawMessage();
}
```


Once retrieved, here is an example of how to display the list of issuing banks in Java Server Pages:

```
<!--
    request.setAttribute("Model", new Communicator().getDirectory());
-->
<select class="form-control">
    <option>Choose a bank</option>
    <c:forEach items="${Model.getIssuersByCountry()}" var="country">
        <optgroup label="${country.key}">
            <c:forEach items="${country.value}" var="issuer">
                <option value="${issuer.getIssuerID()}">${issuer.getIssuerName()}</op
tion>
            </c:forEach>
        </optgroup>
    </c:forEach>
</select>
```

This will generate HTML mark-up for a drop-down list with all the available Issuers, grouped by country, just like in Figure 1.

5.1.3 *PHP Directory request*

To perform a directory request, instantiate a `Communicator` object, and call the `getDirectory()` method:

```
$comm = new Communicator();
$response = $comm->getDirectory();
if ($response->getIsError()) {
    // log error, show error message
}
else {
    // handle response: display list of issuing banks
}
```

Here is the structure of the directory response:

```
class DirectoryResponse {
    public function getIsError();
    public function getErrorResponse();
    public function getDirectoryDateTimeStamp();
    public function getIssuers();
    public function getRawMessage();
}
```

Once retrieved, here is an example of how to display the list of issuing banks using PHP:

```

<!--
$Model = $comm->getDirectory();
-->
<select class="form-control">
  <option>Choose a bank</option>
  <?php foreach ($Model->getIssuersByCountry() as $key => $value) { ?>
    <optgroup label="<?php echo $key; ?>">
      <?php foreach ($value as $issuer) { ?>
        <option value="<?php echo $issuer->getID(); ?>">
          <?php echo $issuer->getName(); ?>
        </option>
      <?php } ?>
    </optgroup>
  <?php } ?>
</select>

```

This will generate HTML mark-up for a drop-down list with all the available Issuers, grouped by country, just like in Figure 1.

5.2 Issue a new Authentication request (Transaction request)

The process of starting a new authentication request is initiated on the Merchant's website by the Consumer. The Merchant enters the information in the message and this information set is then completed with Acquirer information (by the Routing Service) and sent to the Validation Service/Issuer.

The library simplifies this by requiring input for only a limited number of fields, with the rest being filled in automatically, and handles the process in the `NewAuthenticationRequest()` method.

Request parameters:

- **IssuerId:** The identity of the Issuer that was selected by the Consumer;
- **Language:** This field enables the Issuer's website or mobile app to select the Consumer's preferred language (e.g. the language that was selected on the Merchant's website or mobile app), if this language is supported by the Issuer's website or mobile app. The code list according to ISO 639-1. (e.g. 'en' for English, 'nl' for Dutch) should be used. If a non-supported or non-existing language is entered, the standard language of the Issuer is used;
- **ExpirationPeriod:** If specified, the Consumer must approve the transaction in the specified number of seconds. The Consumer must complete the transaction within this period; otherwise, the Issuer sets the transaction status to "Expired". Minimum is 60 seconds and maximum and

default is 300 seconds. The Merchant **does not** have to enter a value for this field. It is recommended for Merchants to leave the `ExpirationPeriod` empty;

- `RequestedServiceId`: ID of the list of services provided. Please refer to the iDIN Merchant Implementation Guidelines document for more information;
- `MerchantReference`: Unique identifier for this transaction, set by the Merchant;
- `AssuranceLevel`: Describes the minimum level of assurance required; can be either `AssuranceLevel.Loa2` or `AssuranceLevel.Loa3`.

Response parameters:

- `IsError`: True if there was an error, or false if the response was received without errors;
- `Error`: An `ErrorResponse` object containing error information, if one occurred;
- `IssuerAuthenticationUrl`: An URL given by the Issuer, to which the Merchant will have to redirect the Consumer to, so they can authenticate this transaction;
- `TransactionId`: The identifier assigned by the Routing Service for this transaction. This parameter is required to request the status in a later stage, see Section 5.3;
- `TransactionCreateDateTimeStamp`: The specific date and time when this transaction was registered by the Routing Service of the Acquirer;
- `RawMessage`: The actual XML message received from the Routing Service.

Below examples are presented on how to perform a new authentication request and check the response, for each of the supported platforms.

5.2.1 .NET Authentication request

To perform an authentication request, instantiate a `Communicator` instance and call the `NewAuthenticationRequest()` method:

```
var authenticationRequest = new BankID.Merchant.Library.AuthenticationRequest(
    "entranceCode",
    ServiceIds.Address | ServiceIds.DateOfBirth,
    "issuerId"
);

var authenticationResponse =
communicator.NewAuthenticationRequest(authenticationRequest);

if (authenticationResponse.IsError)
{
```

```
        // display error: response.Error.ErrorMessage, response.Error.ErrorDetails
    }
    else
    {
        // redirect to response.IssuerAuthenticationURL
    }
}
```

Here is the structure of the authentication request:

```
public class AuthenticationRequest
{
    public AssuranceLevel AssuranceLevel { get; }
    public DateTime CreateDateTimeStamp { get; }
    public string EntranceCode { get; }
    public TimeSpan? ExpirationPeriod { get; }
    public string IssuerId { get; }
    public string Language { get; }
    public string MerchantReference { get; }
    public ServiceIds ServiceId { get; }
}
```

And this is the structure of the authentication response:

```
public class AuthenticationResponse
{
    public bool IsError { get; }
    public ErrorResponse Error { get; }
    public Uri IssuerAuthenticationUrl { get; }
    public string TransactionId { get; }
    public DateTime TransactionCreateDateTimeStamp { get; }
    public string RawMessage { get; }
}
```

5.2.2 Java Authentication request

To perform an authentication request, instantiate a `Communicator` instance and call the

`newAuthenticationRequest()` method:

```
AuthenticationRequest nar = new AuthenticationRequest(
    "entranceCode",
    ServiceIds.IsEighteenOrOlder | ServiceIds.Address,
    null,
    AssuranceLevel.Loa2,
    "en",
    null);
```

```
AuthenticationResponse response = new Communicator()
    .newAuthenticationRequest(nar);
```

```

if (response.getIsError()) {
    // log or display error message
}
else {
    // redirect to response.getIssuerAuthenticationURL()
}

```

Here is the structure of the authentication request:

```

public class AuthenticationRequest {
    public void setEntranceCode(String entranceCode);
    public void setLanguage(String language);
    public void setExpirationPeriod(Duration expirationPeriod);
    public void setMerchantReference(String merchantReference);
    public void setRequestedServiceID(ServiceId serviceID);
    public void setAssuranceLevel(AssuranceLevel assuranceLevel);
}

```

And this is the structure of the authentication response:

```

public class AuthenticationResponse {
    public boolean getIsError();
    public ErrorResponse getErrorResponse();
    public String getIssuerAuthenticationURL();
    public String getTransactionID();
    public XMLGregorianCalendar getTransactionCreateDateTimeStamp();
    public String getRawMessage();
}

```

5.2.3 PHP Authentication request

To perform an authentication request, instantiate a Communicator instance and call the `newAuthenticationRequest()` method:

```

$comm = new Communicator();
$ar = new AuthenticationRequest();
$ar->setAssuranceLevel(AssuranceLevel::$Loa2);
$ar->setEntranceCode('entranceCode');
$ar->setExpirationPeriod(NULL);
$ar->setIssuerID('ISSUERID');
$ar->setLanguage('en');
$ar->setMerchantReference('merchantReference');
$ar->setServiceID(ServiceIds::$IsEighteenOrOlder | ServiceIds::$Address);
$response = $comm->newAuthenticationRequest($ar);
if ($response->getIsError()) {
    // log error, show error message
}
else {
    // redirect to $response->getIssuerAuthenticationURL()
}

```

```
}
```

Here is the structure of the authentication request:

```
class AuthenticationRequest {  
    public function setEntranceCode($entranceCode);  
    public function setLanguage($language);  
    public function setExpirationPeriod($expirationPeriod);  
    public function setMerchantReference($merchantReference);  
    public function setServiceID($serviceID);  
    public function setAssuranceLevel($assuranceLevel);  
}
```

And this is the structure of the authentication response:

```
class AuthenticationResponse {  
    public function getIsError();  
    public function getErrorResponse();  
    public function getIssuerAuthenticationURL();  
    public function getTransactionID();  
    public function getTransactionCreateDateTimeStamp();  
    public function getRawMessage();  
}
```

5.3 Request the authentication Status (Status request)

To retrieve requested information after a successful authentication request en response, the Merchant will start the status protocol by sending a status request to the Routing Service.

If no errors are encountered, the Routing Service will respond with the full Assertion received from the Validation Service, containing the requested information, but encrypted. The library uses the SAML certificate to decrypt that information and returns a `SamlResponse` object.

Request parameters:

- `TransactionId`: the transaction ID.

Response parameters:

- `IsError`: True if there was an error, or false if the response was received without errors.
- `Error`: An `ErrorResponse` object that contains error-related information if an error occurs, or null if there was no error.

- **TransactionId**: Transaction ID for this specific transaction;
- **Status**: String that contains the status message and can be one of the following:
 - *Success*: Positive result; the transaction is or has been executed;
 - *Cancelled*: Negative result due to cancellation by Consumer; the transaction will not be executed;
 - *Expired*: Negative result due to expiration of the transaction; the transaction will not be executed. This status is only given when the Consumer has not approved the transaction within the Expiration period;
 - *Failure*: Negative result due to other reasons; the transaction will not be executed;
 - *Open*: Final result not yet known;
 - *Pending*: Transaction has not yet been completed;
- **StatusDateTimeStamp**: Field set to the time of the transaction's status change;
- **RawMessage**: The actual XML message received from the Routing Service;
- **SamlResponse**: An object that contains more information. This field is only present when Status is "*Success*".

Here are the fields in the SAML response:

- **TransactionId**: Identification of the current transaction;
- **MerchantReference**: The merchant reference, set by the Merchant;
- **AcquirerId**: Contains the ID of the party that created the SAML Response, which is always the Acquirer's Routing Service;
- **Version**: SAML version;
- **Attributes**: A key-value map of all the decrypted attributes that are provided by the Issuer. Depending on what service the Merchant has requested one or more attributes are returned. Here are a few examples (please refer to the iDIN Merchant Implementation Guidelines document for more information and formatting rules):
 - **consumer.gender**: Gender of Consumer;
 - **consumer.initials**: Initials of Consumer;
 - **consumer.18orolder**: Specifies whether the Consumer is 18 or older;
 - **BankID.DeliveredServiceID**: Returned to the Merchant in the Response to indicate which requested attributes are delivered conform the minimal set as defined in the Merchant Implementation Guidelines;

- **SamlStatus:** A structure further detailing the response status. This contains:
 - **StatusMessage:** A general-purpose message. Determined by Routing or Validation Service;
 - **StatusCodeFirstLevel:** A status code representing the status of the SAML request;
 - **StatusCodeSecondLevel:** A status code representing the status of the handling of **DeliveredServiceId**. If not all the attributes can be provided conform a specified set as specified in Section 5.5 of the Merchant Implementation Guidelines by the Validation Service, this field is set to “urn:nl:bvn:bankid:1.0:status:IncompleteAttributeSet”.

Restrictions on requesting the status

A Merchant may only initiate the `getResponse()` method when the Consumer is successfully redirected back to the Merchant by the Issuer using the Merchant Return URL (that the Merchant has configured in the configuration file).

The SAML Assertion issued by the Validation Service is only valid for 30 seconds. From the moment the Consumer has been successfully redirected to the Merchant up to the time the Assertion is expired, the Merchant can make status requests. However, the Merchant should only perform more than one status request if it has received a timeout from the Routing Service (see Chapter 9 of the iDIN Merchant Implementation Guidelines for more detail). The Merchant is not allowed to perform any more status requests once it received notification from the Validation Service that the Assertion has expired.

Merchants will be perceived as committing undesirable actions if they use the `getResponse()` method more than the above described limitation, as doing so places unnecessarily heavy demands on the Acquirer's and Issuer's system.

Below examples are presented on how to perform requests and check the responses, for each of the supported platforms.

5.3.1 .NET Status request

To perform the authentication status request, instantiate a `Communicator` instance and call the `GetResponse()` method:

```
var statusRequest = new StatusRequest
{
    TransactionId = "transactionId"
};
```



```
var statusResponse = communicator.GetResponse(statusRequest);
if (response.IsError)
{
    // store error, or display it
}
else
{
    if (response.Status.Equals(StatusResponse.Success))
    {
        // extract information from statusResponse.SamlResponse.AttributeStatements
    }
}
```

Here is the structure of the status request:

```
public class StatusRequest
{
    public string TransactionId { get; }
}
```

Here is the structure of the status response:

```
public class StatusResponse
{
    public bool IsError { get; }
    public ErrorResponse Error { get; }
    public string TransactionId { get; }
    public string Status { get; }
    public DateTime? StatusDateTimeStamp { get; }
    public SamlResponse SamlResponse { get; }
    public string RawMessage { get; }
}
```

5.3.2 Java Status request

To perform the authentication status request, instantiate a `Communicator` instance and call the `getResponse()` method:

```
StatusRequest sr = new StatusRequest("1234567890");

StatusResponse response = new Communicator().getResponse(sr);
if (response.getIsError()) {
    // log or display error message
}
else {
    if (response.getStatus() == StatusResponse.Success) {
```

```

        // extract information from response.getAcceptanceReport()
    }
}

```

Here is the structure of the status request:

```

public class StatusRequest {
    public void setTransactionID(String transactionID);
}

```

Here is the structure of the status response:

```

public class StatusResponse {
    public boolean getIsError();
    public ErrorResponse getErrorResponse();
    public String getTransactionID();
    public String getStatus();
    public XMLGregorianCalendar getStatusDateTimestamp();
    public String getRawMessage();
    public SamlResponse getSamlResponse();
}

```

5.3.3 PHP Status request

To perform the authentication status request, instantiate a `Communicator` instance and call the `getResponse()` method:

```

$comm = new Communicator();
$sr = new StatusRequest();
$sr->setTransactionID('1234567890');
$response = $comm->getResponse($sr);
if ($response->getIsError()) {
    // log error, show error message
}
else {
    if (strcmp($response->getStatus() == StatusResponse::$Success) == 0) {
        // extract information from $response->getSamlResponse()
    }
}

```

Here is the structure of the status request:

```

class StatusRequest {
    public function setTransactionID($entranceCode);
}

```

Here is the structure of the status response:

```
class StatusResponse {  
    public function getIsError();  
    public function getErrorResponse();  
    public function getTransactionID();  
    public function getStatus();  
    public function getStatusDateTimeStamp();  
    public function getRawMessage();  
    public function getSamlResponse();  
}
```

6 Testing

The testing activity for the iDIN Software Libraries requires integration with a Routing Service and a test account from the Acquirer that can be used. We recommend contacting your Acquirer to request a test account for testing the libraries integration. The test cases could include scenarios for performing the following actions:

- Directory Requests (successful or with error);
- Authentication Requests (successful or with error);
- Status requests (successful or with error);
- Verifying the information logged for each action;
- Verifying the messages that are exchanged.

7 Error Handling and Logging

Error handling is part of the Directory, Authentication and Status methods. When a request is rejected for system-related or field-related reasons, an error message is sent in response to the occurrence of this error.

A generic error response message is sent as a reaction to:

- The Directory request;
- The Authentication request;
- The Status request.

For all messages received by the iDIN Software Library, a log will be created containing references to the `DateTimeStamp`, and the actual message including any errors the message might contain.

The libraries provide extensive logging support for monitoring the messages between the Merchant and Routing Service. This will be optional and configurable from the application configuration file (see Section 4.2 – Library Settings, for more information about how you can configure the library to output these messages).

In addition to logging the response and request messages to disk, the libraries allow programmatic access to the messages, to be used in cases in which simply logging the message to disk doesn't fulfill your requirements (e.g. running in a cluster, saving the messages in a database, sending a notification when an error message is received, etc.). This can be achieved using any of the following approaches:

- The original XML response messages are accessible directly from the response objects for the directory request, new authentication request and status request. This approach enables developers to associate a response with its XML representation, but it is only available for responses, and only for the responses with no parsing errors. For more details on how to access the XML messages for each response, see Chapter 5;
- The internal logging code uses a pluggable architecture, allowing developers to plug in their own logging code and handle the messages however they see fit. This is the more powerful approach, allowing Merchants to save all messages (both request and response), even if parsing errors occur. The messages will not be associated with their object representation however, so if this is needed then the first approach should be used. For more information on how to plug your own logging code, see the platform-specific sections below.

Additionally, apart from logging the request and response, the libraries will also log diagnostic information, enabling you to investigate potential configuration issues. Instructions for how to enable the logging of diagnostic information for each platform are included below.

7.1 .NET Logging

The following steps should be followed in order to plug in your own logging code:

- Implement interface `ILogger`, specifically the `Log` and `LogXmlMessage` methods. The `Log` method will be called by the library for logging trace/diagnostic messages, such as when a signature is being computed, the result of a message validation checks, etc. The `LogXmlMessage` will be called by the library when logging a request/response message, and will receive the entire XML message as an argument. This method can be used to save the messages wherever you see fit;
- Create a configuration class that implements `BankID.Merchant.Library.IConfiguration` and attach your own `ILogger` implementation to the class.

For logging diagnostics, the library is using the standard `Trace` class, and in order to enable the saving of these messages to disk they have to be configured to output for standard .NET trace diagnostics. For more information on how to do this, see this MSDN article - [https://msdn.microsoft.com/en-us/library/xe8whywc\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/xe8whywc(v=vs.110).aspx).

Additionally, the library will itself throw errors whenever it detects invalid input data. In order to handle these errors, you should catch the library-specific exception, `CommunicatorException`.

7.2 Java Logging

The following steps should be followed in order to plug in your own logging code:

- Implement interface `ILogger`, specifically the `Log` and `LogXmlMessage` methods. The `Log` method will be called by the library for logging trace/diagnostic messages, such as when a signature is being computed, the result of a message validation checks, etc. The `LogXmlMessage` will be called by the library when logging a request/response message, and will receive the entire XML message as an argument. You can use this method to save the messages wherever you see fit;

- Implement interface `ILoggerFactory` to return your own implementation of `ILogger`. The interface only has one method – `Create`, which you will have to implement so that it returns your own custom implementation of `ILogger`. The factory will be used internally by the library for creating logger instances for each communicator;
- Call the `Configuration.setLoggerFactory()` method to overwrite the default logger factory with your own implementation that the library will use for creating loggers.

For logging diagnostics, the library is using the standard `java.util.logging.Logger` class, and when the library is run in the context of a Java web application server (such as Apache Tomcat, or Glassfish), these should be available in the server's log files. If you wish to turn these messages on/off, you may call the `Configuration.setLogsEnabled()` method.

Additionally, the library will itself throw errors whenever it detects invalid input data. In order to handle these errors, you should catch the library-specific exception, `CommunicatorException`. All other exceptions that might occur during library operations will be encapsulated, and you can access the original exception with `CommunicatorException.getCause()`.

7.3 PHP Logging

The following steps should be followed in order to plug in your own logging code:

- Implement interface `ILogger`, specifically the `Log` and `LogXmlMessage` methods. The `Log` method will be called by the library for logging trace/diagnostic messages, such as when a signature is being computed, the result of a message validation checks, etc. The `LogXmlMessage` will be called by the library when logging a request/response message, and will receive the entire XML message as an argument. You can use this method to save the messages wherever you see fit;
- Call the `Configuration.setLogger()` method to overwrite the default logger with your own implementation that the library will use for creating loggers.

For logging diagnostics, the library is using the standard `error_log()` PHP function, and if you overwrite the appropriate section in `php.ini`, you are able to direct the logging to a file of your choice. See <http://php.net/manual/en/function.error-log.php> and <http://php.net/manual/en/errorfunc.configuration.php#ini.error-log> for more information about how to do that.

Additionally, the library will itself throw errors whenever it detects invalid input data. In order to handle these errors, you should catch the library-specific exception, `CommunicatorException`. All other exceptions that might occur during library operations will be encapsulated, and you can access the original exception with `CommunicatorException.getPrevious()`.