# Krestfield EzSign

## Client Integration Guide

Version 4.0


## Introduction

The Krestfield EzSign Client is a lightweight java package which interfaces with the EzSign Server enabling applications to quickly generate and verify digital signatures or encrypt and decrypt data without the need for complex programming

The client also provides utilities to disassemble signatures (e.g. extracting the signer certificates, digest algorithms used etc.) and hash data

This guide details the steps required to integrate the client into applications and make use of the API

For server side setup please refer to the *EzSign Installation and Configuration Guide*

## JAR Files

The client consists of the following JAR files:

- `kezsign-client-x.y.z.jar`
- `kezsign-client-utils-x.y.z.jar`

(Where x.y.z is the version number e.g. 4.0.0)

These are located at `[installation folder]/EzSignClient/lib`

Where `[installation folder]` is the location the server was installed

To make use of the signing, verification and encryption functions, add the `ezsign-client-x.y.z.jar` to your application's class path

To make use of the signature parsing and hashing utilities, add the `kezsign-client-utils-x.y.z.jar` to your application's class path

# Client API

The API to the EzSign client has intentionally been kept simple. Decisions on which keys to use, what certificates to include in signatures and what hashing algorithms to use are configured at the server level

All the client API calls are included in the EzSignClient class, which is included in the `com.krestfield.ezsign` package

## Constructor

The constructor is defined as:

```
public EzSignClient(String host, int port)
```

Where:
`host` is the server hostname (or IP Address)
`Port` is the server listening port number

The default connection timeout to the server is 5 seconds (5000 milliseconds). But this can be overridden by using the following constructor:

```
public EzSignClient(String host, int port,
                    int timeoutInMs, int readTimeoutInMs)
```

Where:
`timeoutInMs` is the connection timeout expressed in milliseconds
`readTimeoutInMs` is the read timeout expressed in milliseconds

Example:

```
import com.krestfield.ezsign;

public class ClientTest
{
      public static void main(String[] args)
      {
            EzSignClient client = new EzSignClient("10.100.12.15", 5656);
      }
}
```

If the communications between the client and server are to be secured, an Authentication Code may be used. The Authentication Code is provided by using one of the following constructors:

```
public EzSignClient(String host, int port, String authCode)
```

```
public EzSignClient(String host, int port,
                    int timeoutInMs, int readTimeoutInMs,
                    String authCode)
```

This will result in the encryption of the traffic between the client and server

There are no restrictions on what Authentication Code can be used, but a longer more complex code will increase security

The same Authentication Code must also be configured on the server. Refer to the Installation and Configuration Guide for details on how to configure this on the server

## Generate Signature Methods

The following method is called to generate a signature:

```
public byte[] signData(String channelName, byte[] dataToSign,
                       boolean isDigest)
```

The signature returned is dependent on the signature type specified at the server and will be either a PKCS#7 formatted signature or a raw signature

Note: For large data sets, it is recommended to hash the data beforehand and provide the hash as the `dataToSign` together with `isDigest=true`. This prevents large amounts of data being passed between the client/server interface. Note: The Client Utils API (see below) can be used to generate the required hash

This method throws the following exceptions:

`KSigningException`
: There was an error during the signing process

`KEzSignException`
: There was an internal error, incorrect parameters or other error

`KEzSignConnectException`
: There was an error connecting to the server

Example:

```
byte[] dataToSign = "Hello".getBytes();
byte[] signature = client.signData("CHANNEL1", dataToSign, false);
```

## Verify PKCS#7 Signature Methods

The following methods are used to verify a PKCS#7 signature and will perform the required path building and revocation checking as configured at the server

Verifying Signature 1

```
public void verifySignature(String channelName, byte[] signature,
                            byte[] contentBytes, boolean dataIsDigest)
```

If `contentBytes` is a hash of the data then `dataIsDigest` must be `true`, otherwise `false`

This method throws the following exceptions:

`KVerificationException`
: There was an error during the verification process

`KPathException`
: There was a path building error

`KRevocationException`
: A certificate is revoked or there was an error during the revocation check process

`KEzSignConnectException`
: There was an error connecting to the server

`KEzSignException`
: There was another error

Example:

```
try {
      client.verifySignature("CHANNEL1", signature, content, false);
} catch (KVerificationException verifyEx) {
      System.out.println("There was a verification error: " +
                         verifyEx.getMessage());
} catch (KPathException pathEx) {
      System.out.println("There was a path build error: " +
                         pathEx.getMessage());
} catch (KRevocationException revEx){
      System.out.println("There was a revocation check error: " +
                         revEx.getMessage());
}
...
```

Verifying Signature 2

This method allows for the by-passing of revocation checking and/or the by-passing of path building i.e. only a simple verification check will be performed to confirm the data was signed by the certificate specified in the signature and has not been altered

```
public void verifySignature(String channelName, byte[] signature,
                byte[] contentBytes, boolean dataIsDigest,
                boolean bypassRevocationCheck, boolean bypassPathBuild)
```

If the values for `bypassRevocationCheck` and `bypassPathBuild` are both set to `false` the signature will be verified in the same way as the previous method

This method throws the following exceptions:

`KVerificationException`
: There was an error during the verification process

`KPathException`
: There was a path building error

`KRevocationException`
: A certificate is revoked or there was an error during the revocation check process

`KEzSignConnectException`
: There was an error connecting to the server

`KEzSignException`
: There was another error

## Verify Raw Signature Methods

The following methods are used to verify Raw (for RSA these are PKCS#1 formatted) signatures and will perform the required path building and revocation checking as configured at the server

### Verifying Raw Signature 1

Note that as a raw signature does not contain the signer certificate, this must be provided

If there are other certificates in the path that are not stored in the channel, use the Verify Raw Signature 2 method below. Path building and revocation checking (if configured at the server) will be performed

```
public void verifySignature(String channelName, byte[] signature,
                byte[] contentBytes, boolean dataIsDigest,
                X509Certificate signerCert)
```

This method throws the following exceptions:

`KVerificationException`
: There was an error during the verification process

`KPathException`
: There was a path building error

`KRevocationException`
: A certificate is revoked or there was an error during the revocation check process

`KEzSignConnectException`
: There was an error connecting to the server

`KEzSignException`
: There was another error

### Verifying Raw Signature 2

This method accepts the signer certificate as well as other certificates in the path. Path building and revocation checking (if configured at the server) will be performed

```
public void verifySignature(String channelName, byte[] signature,
                byte[] contentBytes, boolean dataIsDigest,
                X509Certificate signerCert,
                X509Certificate[] otherCerts)
```

This method throws the following exceptions:

`KVerificationException`
: There was an error during the verification process

`KPathException`
There was a path building error

`KRevocationException`
A certificate is revoked or there was an error during the revocation check process

`KEzSignConnectException`
There was an error connecting to the server

`KEzSignException`
There was another error

Verifying Raw Signature 3

Use this method if you wish to bypass revocation checking (whether configured at the server or not) and/or path building

```
public void verifySignature(String channelName, byte[] signature,
                byte[] contentBytes, boolean dataIsDigest,
                X509Certificate signerCert,
                X509Certificate[] otherCerts,
                boolean bypassRevocationCheck, boolean bypassPathBuild)
```

This method throws the following exceptions:

`KVerificationException`
There was an error during the verification process

`KPathException`
There was a path building error

`KRevocationException`
A certificate is revoked or there was an error during the revocation check process

`KEzSignConnectException`
There was an error connecting to the server

`KEzSignException`
There was another error

Note:
`otherCerts` can be `null` if you do not wish to specify any other certificates in the path
If you wish to perform just the signature verification operation. Set `bypassRevocationCheck` and `bypassPathBuild` both to `true` and `otherCerts` to `null`

## Generate Random Number Methods

The following method is called to generate random data:

```
public byte[] generateRandomBytes(String channelName, int numBytes)
```

The number of random bytes specified will be returned

This method throws the following exceptions:

`KEzSignConnectException`
: There was an error connecting to the server

`KEzSignException`
: There was another error

## Encrypt/Decrypt Methods

These methods provide encryption and decryption using AES keys previously generated on the server. The algorithm used is AES with CBC (Cipher Block Chaining) and PKCS#5 padding. A random IV (Initialisation Vector) is created every time data is encrypted and this IV is placed in the first 16 bytes of the returned data, with the remaining bytes being the encrypted data itself

The following method is called to encrypt data:

```
public byte[] encryptData(String channelName, byte[] dataToEncrypt,
                          String keyLabel)
```

This will encrypt the clear data contained in `dataToEncrypt` using the key referenced by `keyLabel` and return the encrypted data

`keyLabel` must refer to a key which has previously been generated on the server using the management utility. If the key does not exist `KEncipherException` will be thrown

This method throws the following exceptions:

`KEncipherException`
: There was an error whilst encrypting the data

`KEzSignConnectException`
: There was an error connecting to the server

`KEzSignException`
: There was another error

The following method is called to decrypt previously encrypted data:

```
public byte[] decryptData(String channelName, byte[] encryptedData,
                          String keyLabel)
```

This will decrypt data previously encrypted with the `encryptData` method using the key referenced by `keyLabel` and return the clear data

`keyLabel` must refer to a key which has previously been generated on the server using the management utility. If the key does not exist `KEncipherException` will be thrown

This method throws the following exceptions:

`KEncipherException`
: There was an error whilst decrypting the data

`KEzSignConnectException`
: There was an error connecting to the server

`KEzSignException`
: There was another error

Example:

```
byte[] originalData = “Hello”.getBytes();
byte[] encryptedData = encryptData(“CHANNEL1”, originalData, “key1”);
byte[] clearData = decryptData(“CHANNEL1”, encryptedData, “key1”);
// clearData will be equal to originalData i.e. “Hello”
```

# Client Utils API

The Client Utilis API calls are included in the following classes:

`KConvert`
: Converts data from bytes to Base64 or ASCII Hex Strings and back

`KHash`
: Hashes data

`KSignatureParser`
: Parses a signature providing the certificates, signature hash, signer certificate DN etc.

`KCertUtils`
: Provides utilities to check obtain certificate details and compare DNs

These classes are contained in the `kezsign-client-utils-x.y.z.jar` file and held in the `com.krestfield.ezsign.client.utils` package

## KConvert

This class contains the following static methods for data conversion:

```
public static String ToBase64String(byte[] data)
```

Converts binary data to a Base64 string

```
public static byte[] B64StringToBytes(String b64Str)
```

Converts a Base64 string to a byte array

```
public static byte[] HexStrToBytes(String hex)
```

Converts a hex string of the form A05BC9CE to a byte array

```
public static String BytesToHexStr(byte[] bytes)
```

Converts binary data to an ASCII Hex string of the form A05BC9CE...

## KHash

This class contains the following methods for the generation of hashes:

`public KHash(String hashAlgorithm)`

The constructor can be passed the hash algorithm as a string. Accepted values are `SHA-1`, `SHA-224`, `SHA-256`, `SHA-384`, `SHA-512`, `SHA3-224`, `SHA3-256`, `SHA3-384` and `SHA3-512`

`public void update(byte[] data)`

Sections or slices of data can be provided as received or processed and will be included in the overall hash generated

`public byte[] digest()`

This method produces the hash value over all the data provided

## KSignatureParser

This class contains the following methods for the parsing of PKCS#7 signatures:

`public KSignatureParser(String b64Signature)`

The constructor accepts a base64 PKCS#7 formatted signature

`public KSignatureParser(byte[] signature)`

As above but accepts a binary PKCS#7 formatted signature

`public X509Certificate[] getCertificates()`

Returns all the certificates included in the signature

`public X509Certificate getSignerCertificate()`

Returns the certificate that signed the data

`public String getSignersSubjectDn()`

Returns the Distinguished Name of the signer certificate

`public String getSignersIssuerDn()`

Returns the Issuer's Distinguished Name of the signer certificate

`public String getSignatureAlgorithmName()`

Returns the signature's full algorithm name e.g. SHA256withRSA

```
public String getSignatureDigestAlgorithm()
```

Returns the signature's digest algorithm e.g. SHA-1, SHA-256

```
public String getSignatureEncryptionAlgorithm()
```

Returns the signature's encryption algorithm e.g. RSA

## KCertUtils

This class contains the following methods:

```
public boolean isCertificateRoot(X509Certificate cert)
```

Returns true if the provided certificate is self-signed i.e. a root certificate

```
public boolean isCertificateCA(X509Certificate cert)
```

Returns true if the provided certificate is a CA certificate. This examines the Basic Constraints extension and will return true for root and intermediate CA certificates

```
public String getCertificateAlgorithmName(X509Certificate cert)
```

Returns the certificate's signature full algorithm name e.g. SHA256withRSA

```
public String getCertificateDigestAlgorithm(X509Certificate cert)
```

Returns the certificate's signature digest algorithm e.g. SHA-1, SHA-256 etc

```
public String getCertificateEncryptionAlgorithm(X509Certificate cert)
```

Returns the certificate's signature encryption algorithm e.g. RSA, ECDSA etc

```
public boolean dnsAreEquivalent(String dn1, String dn2)
```

Given two string representations of distinguished names, returns true if they are equivalent. This performs a comparison on the DN components rather than a string comparison e.g.

```
CN=Test Cert, OU=Engineering, O=Krestfield,  C=GB
```

is equivalent to

```
CN=Test Cert,O=Krestfield,OU=Engineering,C=GB
```

Even though the strings are not equal

# Sample

An example of how to use the client and utility methods as detailed above is provided in the samples folder

This can be built by running the `buildExample` script and run with the `runExample` script

The source code is contained within the `TestClient.java` file

# Support

All questions, queries around the API described within this document should be directed to Krestfield Support at the following email address:

`support@krestfield.com`

# Sign/Verify Example

The following is a transcript of a sample generating and verifying a signature

```
import com.krestfield.ezsign.EzSignClient;
import com.krestfield.ezsign.KEzSignConnectException;
import com.krestfield.ezsign.KEzSignException;
import com.krestfield.ezsign.KPathException;
import com.krestfield.ezsign.KRevocationException;
import com.krestfield.ezsign.KSigningException;
import com.krestfield.ezsign.KVerificationException;
import com.krestfield.ezsign.client.utils.KHash;
import com.krestfield.ezsign.client.utils.KSignatureParser;
import com.krestfield.ezsign.client.utils.KConvert;

/**
 * TestClient
 *
 * This example demonstrates the use of the client and client utils
 *
 * Copyright Krestfield 2018
 */
 public class TestClient
 {
    public static void main(String args[])
    {
        /**
         * Specify the server IP Address. port number and channel name
         */
        String serverIPAddress = "127.0.0.1";
        int serverPortNumber = 5656;
        String channelName = "TEST";


        /**
         * Create the EzSign Client
         */
        EzSignClient client = new EzSignClient(serverIPAddress, serverPortNumber);

        /**
         * Specify the data to sign
         */
        byte[] dataToSign = "Hello".getBytes();
        byte[] signature = null;

        try
        {
            /**
             * Generate Signature Example
             */
            signature = client.signData(channelName, dataToSign, false);
            System.out.println("Generated signature successfully");
            System.out.println("Returned signature: " + KConvert.ToBase64String(signature));

            // Use the signature parser to obtain the signer DN and signature alg
            KSignatureParser signatureParser = new KSignatureParser(signature);
            System.out.println("Signer Certificate DN: " +
                                        signatureParser.getSignersSubjectDn());
            System.out.println("Signature Algorithm: " +
                                        signatureParser.getSignatureAlgorithmName());

            // Use the hash generation utils to hash the data
            KHash hash = new KHash(KHash.SHA256);
            hash.update(dataToSign);
            System.out.println("Hash over data: " + KConvert.BytesToHexStr(hash.digest()));
        }
        catch (KEzSignConnectException connEx)
        {
            System.out.println("There was an error connecting to the server: " +
                                connEx.getMessage());
```

```
            return;
        }
        catch (KEzSignException ex)
        {
            System.out.println("There was an error signing the data.  Error: " +
                               ex.getMessage());
            return;
        }
        catch (KSigningException sigEx)
        {
            System.out.println("There was an error signing the data.  Error: " +
                               sigEx.getMessage());
            return;
        }
        catch (Exception e)
        {
            System.out.println("There was an error: " + e.getMessage());
            return;
        }

        try
        {
            /**
             * Verify Signature Example
             */
            client.verifySignature(channelName, signature, dataToSign, false);
            System.out.println("Signature verified successfully");
        }
        catch (KVerificationException veriEx)
        {
            System.out.println("There was an error verifying the signature: " +
                               veriEx.getMessage());
        }
        catch (KRevocationException revEx)
        {
            System.out.println("There was a revocation exception : " + revEx.getMessage());
        }
        catch (KPathException pathEx)
        {
            System.out.println("There was a path building exception : " + pathEx.getMessage());
        }
        catch (KEzSignConnectException connEx)
        {
            System.out.println("There was an error connecting to the server: " +
                               connEx.getMessage());
        }
        catch (KEzSignException sigEx)
        {
            System.out.println("There was an error verifying the data.  Error: " +
                               sigEx.getMessage());
        }
    }
}
```