Ascertia Limited
40 Occam Road
Guildford
Surrey
GU2 7YG
United Kingdom

Tel: +44 1483 685500
Fax: +44 1483 573704

www.ascertia.com

ADSS GoSign Applet Developer's Guide

Document Version: 3.3.0.1

Document Issued: March 2008

Contents

1 Introduction

1.1 Scope

This document provides information on GoSign applet, its working and its integration with online web application. It also provides details on how the web application communicates with the ADSS server.

1.2 Intended Readership

This guide is intended for developers who are interested in writing applications which will use the GoSign applet for client side signing of PDF documents. Intended readers must have good knowledge of JavaScript, HTML, web services, Java and/or .NET.

1.3 Conventions

The following typographical conventions are used in this guide to help locate and identify information:

Bold text identifies menu names, menu options, items you can click on the screen, file names, folder names, and keyboard keys.

`Courier` font identifies code and text that appears on the command line.

`Bold courier` identifies commands that you are required to type in.

1.4 Technical support

If Technical Support is required, Ascertia has a dedicated support team providing debugging assistance, integration assistance and general customer support. Ascertia Support can be accessed in the following ways:

Support Email	support@ascertia.com
Support MSN Messenger	support@ascertia.com

In addition to the free support service describe above, Ascertia provides formal support agreements with all product sales. Please contact sales@ascertia.com for more details.

A Product Support Questionnaire should be completed to provide Ascertia Support with further information about your system environment. When requesting help it is always important to confirm:

- System Platform details;
- ADSS Server version number and build date;
- Details of specific issue and the relevant steps taken to reproduce it;
- Database version and patch level;
- The product log files.

2 Overview

GoSign is a client side digital signing applet that integrates via a business web application with the ADSS (Advanced Digital Signature Services) Server.

The business web application will execute a number of business functions that result in a PDF being sent to the ADSS Server and a hash of the PDF being returned. Optionally, organisation or corporate signatures also can be applied to the PDF by the ADSS Server.

The PDF's hash is returned to the GoSign applet running in the user's browser where the user can sign it using X509 certificate credentials held within the Windows keystore. The credentials may optionally be held on a smart card. The PDF and the signature are finally assembled by the ADSS Server when requested to by the business web application.

See the ADSS Administration Guide for a detailed description for configuring the ADSS Server and the ADSS Demo Guide for configuring the server for use with GoSign.

2.1 Applet Runtime Requirements

The signed Java applet requires that *Java Plug-In* version 1.5.0 or later is installed on the client machine. This is necessary because the underlying APIs use functionality which is only available from JDK 1.5.0 or above

The applet does not work with the standard virtual machine distributed with some versions of Internet Explorer. If the appropriate Java Plug-in version is not found the GoSign applet either tries to upgrade it (behavior varies from browser to browser) or provides an error message and details of where the correct Java plug-in can be downloaded from.

The applet is digitally signed using a key certified by a public CA already trusted within the browser so that it can gain access to the user's local file system and resources.

3 GoSign Applet Integration

The GoSign applet is a web page component for a custom web application (see section 4) built to sign documents using client side keys. The applet does not present any HTML controls to the user but will accept certificates and signing actions from the custom designed web page. The application developer needs to design the web application to load the GoSign applet into a browser web page and for the web application to communicate with the ADSS Server to complete the document signing process.

The following sections describe how to embed the GoSign applet into a web page and how to make use of a Javascript interface to filter client side certificates, of which there may be many, and to sign a document.

3.1 The GoSign Package

A JavaScript file, **gosign.js**, is required to place the GoSign applet into a web page and also to read the data from the HTML form fields if present on the web page where GoSign applet is going to be displayed.

The **gosign.js** file contains constants and functions required for:

1- embedding the applet into the web page

2- reading the values of HTML form fields to be included in a document signing request

3- interacting with applet

GoSign is provided as a complete package in the form of a folder containing applet jar, dependency jars and libraries, configuration files etc. The root folder in the GoSign package hierarchy is named as GoSign. Here is the directory structure for GoSign:

GoSign/gosign.version

GoSign/gosign.license

GoSign/conf/lang/gosign.properties

GoSign/lib/asc_gosign.jar

GoSign/lib/assembla_jce_msks.jar

GoSign/lib/MSKeyStoreJNI.dll

GoSign/lib/gosign.js

The section below provides the details of different functions available in **gosign.js** file:

3.2 GoSign Javascript Interface

GoSign_EmbedApplet (*String GoSignRootFolderURL, String SignatureType, String SignatureMechanism, String ContentSource***)**

This function is used to embed the GoSign applet in a web page

Where

- **GoSignRootFolderURL:** It's the path of the root folder of GoSign default directory structure e.g. http://www.gosigndemo.com/demo/GoSign. The path must be accessible to GoSign Applet.
- **SignatureType:** It's the basic signature type. The only supported value is PDF
  SignatureMechanism: It's the signature production mechanism. For each SignatureType there are different supported signature mechanisms. The only supported signature mechanism value is ZERO_FOOTPRINT
- **ContentSource:** It specifies that how the content to be signed will be provided to applet for signing. The only supported value is REMOTE

GoSign_SetFilterCriteria(*String CriteriaIdentifier, String CriteriaValue***)**

This function sets the certificate filter criteria for the filtering the list of certificates loaded from Windows Keystore.

Where

- **CriteriaIdentifier:** It's the id of the specific filtering criteria. Following filtering criteria identifier are supported:
  - SUBJECT_DN_CONTAINS
  - ISSUER_DN_CONTAINS
  - SHOW_EXPIRED_CERTIFICATES
  - KEY_USAGE_CONTAINS
  - POLICY_OID_CONTAINS
  - ALGORITHM_OID_CONTAINS
- **CriteriaValue:** value for the specific filter criteria identifier.

Each certificate filter criteria identifier along with its possible values with examples is explained below:

- **SUBJECT_DN_CONTAINS**

  If SUBJECT_DN_CONTAINS is specified as a key in **GoSign_SetFilterCriteria()** method then there are two possible ways to assign its value that are explained below:

  - Value in form of RDN

    Value can be assigned in form of key/value pair separated with equal sign; multiple values are separated with comma (**,**), but null or empty string can't be used as value. Possible values are:

    1) If "O=Ascertia" is set as a criteria value then certificates will be fetched with organization named "Ascertia" in its distinguished name (Subject DN).

    2) If "O=Ascertia,OU=Development" is set as a criteria value then certificate with organization named "Ascertia" and organizational unit named "Development" will be fetched.

    Note: Use of a null string **"**O=Ascertia,OU=" is not permitted.

  - Value in form of String

    The criteria value can be set in the form of string e.g. "Ascertia". When this is set in criteria value then all certificates will be fetched/loaded with **"Ascertia"** in its distinguished name (Subject DN).

- **ISSUER_DN_CONTAINS**

  The same rule applies for ISSUER_DN_CONTAINS only the difference is that instead of Subject DN (Issuer DN) is used and criteria identifier is set ISSUER_DN_CONTAINS.

Note: The combination of RDN and String cannot be used for this criteria value at the same time. E.g. O=Ascertia,"Development" is illegal. Filtering can only be performed using the SUBJECT_DN_CONTAINS identifier or the ISSUER_DN_CONTAINS filter at any one time, and either the **RDN** or **String** format must be used in the call.

In an RDN, an email address key/value must be in form of EmailAdress=support@ascertia.com.

- **SHOW_EXPIRED_CERTIFICATES**

  Only two possible values for **SHOW_EXPIRED_CERTIFICATES** criteria identifier are: TRUE or FALSE.

  E.g.

```
GoSign_SetFilterCriteria(GoSign_Constants.
SHOW_EXPIRED_CERTIFICATES, GoSign_Constants.TRUE)
```

When above filter criterion is set then all certificates will be loaded including expired certificates. GoSign_Constants.TRUE can be found in gosign.js file.

- **KEY_USAGE_CONTAINS**

  Multiple values can be specified in key usage identifier using a comma (,) separater.

  E.g.

  When supplying a single value:

  ```
  GoSign_SetFilterCriteria(GoSign_Constants.KEY_USAGE_CONTAINS,
  GoSign_Constants. DIGITAL_SIGNATURE)
  ```

  When multiple values are supplied:

  ```
  GoSign_SetFilterCriteria(GoSign_Constants.
  KEY_USAGE_CONTAINS,GoSign_Constants.DIGITAL_SIGNATURE+",”+
  GoSign_Constants.NON_REPUDIATION)
  ```

- **POLICY_OID_CONTAINS**

  It will load all the certificates that match the certificate policy OID with OID set as value of POLICY_OID_CONTAINS identifier.

  E.g.

  ```
  GoSign_SetFilterCriteria(GoSign_Constants.POLICY_OID_CONTAINS,”
  1.2.3.4.5”)
  ```

- **ALGORITHM_OID_CONTAINS**

  It will load all the certificates that match the signature OID with OID set as value.

  E.g.

  ```
  GoSign_SetFilterCriteria(GoSign_Constants.POLICY_OID_CONTAINS,”
  1.2.3.4.5”)
  ```

GoSign_ShowCertificates(*String certs,String aliases***)**

This function populates a drop down control with certificates fetched from windows keystore using filter criteria if specified. The custom web application's drop down control should generate an HTML Select element whose 'name' and 'id' tags are set to name="GoSignCertificateList" and id="GoSignCertificateList" so that the function can identify the drop down to populate:

Where

Certs: Subejct DN of certificate

Aliases: Certificate Alias taken from windows keystore

GoSign_SetTargetURL(*String TargetUrl***)**

This function sets the URL at which request will be forwarded.

GoSign_SetResultPage(*String ResultPage***)**

This function sets the result page at which signed document will be displayed.

GoSign_SignDocument()

This function starts the signing process to sign document when an HTML sign button is pressed.

3.3 Embedding the GoSign Applet into a Web Page

This **gosign.js** file should be configured in the web page that contains GoSign applet by using the following code:

```
<head>
<script language="JavaScript" src="GoSign/lib/gosign.js" ></script>
</head>
```

Above assumes that the web page is placed at the same level where the GoSign directory is placed.

The following JavaScript code can be used to embed the applet on the web page:

```
<script language="JavaScript">
var b_result = GoSign_EmbedApplet('http://www.gosigndemo.com/demo/GoSign',
'PDF', 'ZERO_FOOTPRINT', 'REMOTE');
if( b_result != true){
            alert(GoSign_GetErrorReason());
}
</script>
```

Once the applet has been embedded successfully then, optionally, certificate filter criteria can be specified by using the following JavaScript code:

```
<script language="JavaScript">
var b_result =
Sign_SetFilterCriteria(GoSign_Constants.SUBJECT_DN_CONTAINS,
"O=Ascertia");
if( b_result != true){
      alert(GoSign_GetErrorReason());
}
</script>
```

Once a filter criteria has been set the following JavaScript code is used to display certificates loaded from windows keystore in a drop down list. The drop down list's HTML Select element is defined by the web application but it should have its 'name' and 'id' tags set to name="GoSignCertificateList" and id="GoSignCertificateList" so that GoSign_ShowCertificates() can identify the drop down to populate:

```
<script language="JavaScript">
var b_result = GoSign_ShowCertificates ();
if( b_result != true){
            alert(GoSign_GetErrorReason());
}
</script>
```

A button control, generated in the page by the web application, should call the **GoSign_SignDocument()** function when clicked. Any certificate selected in the "GoSignCertificateList" drop down list will be used by the GoSign applet to identify the signing key. The following JavaScript code is used to call **GoSign_SignDocument()** for signing.

```
<script language="JavaScript">
```

```
var b_result = GoSign_SignDocument ();
if( b_result != true){
            alert(GoSign_GetErrorReason());
}
</script>
```

Note: It's not recommended to set **null** or **empty string** as a value of criteria value but if you try to set it an error message **"Unsupported filter criteria"** will be returned.

Criteria values must be delimited by double quotes.

If errors occur during applet initialization, signing or other functions, an error reason and error code will be returned.

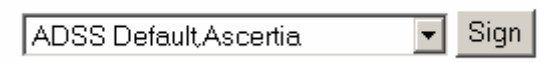
The error codes and their descriptions can be found below, and in gosign.properties under GoSign/conf/lang directory.

Error Code	Description
1001	Unable to download property file
1002	Unsupported paramerters
1003	License does not allow to perform specific operation
1004	Unable to download MSKeyStoreJNI.dll file
1005	Unable to load keystore
1006	Unsupported filter criteria
1007	Unable to find GoSignCertificateList control on page
1008	Unable to find any certificate
1009	Target URL not set
1010	Result Page not defined
1011	Selected Certificate is expired and can not be used for Signing
1012	Hashing Response Failure
1013	Assembly Response Failure

Note: Error description can be changed using the appropriate **gosign.properties** file

3.4 How the GoSign Applet Works

When the web page referencing the applet is loaded by the browser, the applet is downloaded and executed. It loads the user's personal certificates from the Windows Keystore (or from any attached smartcards accessible through Windows) and populates the certificates in the configured drop down list. User selects the certificate from the drop down list and press the Sign button, the applet performs the document signing process in two steps as described in the following sections.



Note: Although business web application workflow will vary, depending on business requirements, it is expected that at this point the PDF or data to be signed is displayed to the user in the browser, so that they can see what they are signing.

3.4.1 Step 1 - Document Hashing

When the user clicks the GUI Sign button the applet reads the HTML form and optionally the Signing Location, Signing Reason and Contact info and posts all the data including the user certificate to the web application. If the signing attributes such as Signing Reason are not filled in then default values can be set by the ADSS Server Signing Profile (see the ADSS Admin Manual for further details).

The web application and ADSS Server perform the following actions:

1. The web application extracts the data from HTTP post request, picks up the document to be signed. The location or preparation of the document is the concern of the business logic. A hashing request containing the document and signing attributes is sent to the ADSS Server.
2. The ADSS Server generates the hash of the document and responds with it to the web application.
3. The web application extracts the hash value from response and sends it to the GoSign applet to be signed by the user.

3.4.2 Step 2 – Document and Signature Assembly

The applet receives the hash from the web application and the user's certificate credentials are used to sign the hash and return a signature to the web application to be embedded in the PDF, as follows:

1. The applet generates a PKCS#7 signature over the hash by using the Windows Cryptographic API. The PKCS#7 is then posted to the web application.
2. The web application extracts the PKCS#7 from the HTTP post request and makes a Signature Assembly request to the ADSS Server.
3. The ADSS server embeds the PKCS#7 signature into the PDF document and responds with the signed document back to the web application. The web application receives the signed PDF document and notifies the applet about the completion of the process.
4. The signed document can be passed pack to the user's browser so that the signature can be examined with the document in a PDF viewer. The Ascertia GoSign Demo demonstrates this feature.

Note: Although it is not shown in this flow, the PDF must have an Empty Signature applied to it before the assembly of with user's signature. The web application will use the ADSS Server to add the Empty Signature Field at the required time in the business flow (see the ADSS Developer's Guide for information on how to add an Empty Signature Field to a PDF). Optionally with the Empty Signature Field call to the ADS Server an organization or company signature can be applied to the document.

4 Developing a GoSign Web Application

A web application can be built to interact with the GoSign applet by using the ADSS Client SDK. ADSS Server libraries for Java or .NET provide APIs that drive the required ADSS Server web services for document signing (see the ADSS Developer's Guide for more information on developing with the ADSS Client SDK).

This section describes the API calls that need to be made in order to perform the steps in 3.4 to provide the server side functions requested by the web application. Sample Java and .NET code use the ADSS Client SDK Java and .NET libraries to create requests and receive response from the ADSS Server.

Note: The code is extracted from the GoSign Demo source files included with the ADSS Client SDK.

4.1 Applet to Web Application Architecture

The GoSign demonstration uses a particular architecture for handling the request/response protocol between the applet and the web application and it is a useful starting point for customisation.

The GoSign applet sends requests to the web application which are handled by:

- A servlet for the Java implementation
- An HTTP handler for the DotNet (C#) implementation

The request protocol is simple. The posted request contains a parameter called "serverHit". The processing for this parameter determines whether a Hashing request or a document signature Assembly request is made, as the following pseudo code demonstrates:

```
if ( serverHit == "FIRST")
{
   // build, send and handle response for document Hashing request
}
else
{
   // build, send and handle response for document signature Assembly
   // request
}
// respond with HTTP status code to applet
```

The request and response handling are defined in more detail for Java and DotNet in the following sections.

4.2 Document Hashing

The web application extracts the signing parameters along with the user certificate from the HTTP post request sent by the GoSign applet. It then generates/loads a document that is going to be signed and creates a Hash request, containing the document, to be sent to ADSS server. The hash response is returned to the applet for signing by the user's credentials.

4.2.1 .NET Document Hashing Request/Response in C#

C# code for the Hash request from the web application to the ADSS Server

```
// Create hash request object
HashADSSRequest hashRequest = new HashADSSRequest();

// acquiring lock here to avoid any kind of data sharing
```

```
context.Application.Lock();

// the pdf document to be signed
hashRequest.Document = ADSSUtils.DocToBase64String(

String.Concat(context.Request.PhysicalApplicationPath,
      "Resources\\PDFs\\ServerSignedPOrdr.pdf"));

// releasing the acquired lock
context.Application.UnLock();

// Profile ID for hashing
hashRequest.ProfileID = "adss:signing:profile:008";
hashRequest.RequestID = "1";

// Using signing reason provided by applet to populate
// Profile Attribute.
ProfileAttribute Attributes = new ProfileAttribute();
Attributes.SigningReason = context.Request["txtSigningReason"];
// Using location provided by user at the asp page to populate Profile
// Attribute.
Attributes.SigningLocation = context.Request["txtLocation"];
// Using contact details provided by user at the asp page to populate
// Profile Attribute.
Attributes.ContactInfo = context.Request["txtContactDetails"];
// Using hard coded Signing Field to populate Profile Attribute.
Attributes.SigningField = "user_signature";
hashRequest.Attributes = Attributes;

// Getting certificate from request and replacing BLANK SPACES with +
// PLUS character.
// It is a must doing thing in GoSign Applet base64 string retrieval.
String signerCert = context.Request["certificate"];
signerCert = signerCert.Replace(' ', '+');
hashRequest.UserCertificateChain.Add(signerCert);

// Providing user name and password for the profile provided above.
hashRequest.OrignatorInfo.OrignatorID = "samples_test_client";
//hashRequest.OrignatorInfo.Password = "password";

// Create an ADSS Client object for transporting the hash request
ADSSClient HashClientRequest = new ADSSClient();

// replace localhost with the the server name and port you have
// configured in your environment.
HashClientRequest.UniformResourceID =
      "http://localhost:8777/adss/signing/dhi";

// Generate the SOAP XML
 HashClientRequest.Content = hashRequest.Generate();

// SendRequest method will send the request to coded URL and return
// the ADSSWebSerCom.Response.IADSSResponse class object which must
// type-cast this interface intelligently as required.
HashADSSResponse hashResponse =
      (HashADSSResponse)HashClientRequest.SendRequest();
```

C# Code for handing the Hash response from the ADSS Server

```
if (String.Compare(hashResponse.ResponseStatus, "SUCCESS") == 0)
{
    // Storing the document id in cookies because the session can not
    // be used with the current architecture.
    context.Response.Cookies["UserSettings"]["DocumentId"] =
      hashResponse.DocumentID;

    // Creating byte array from received hash.
    byte[] baResponseData = Convert.FromBase64String(hashResponse.Hash);

    // populating response object with the hash and status information.
    context.Response.Status = "200 OK";
    context.Response.ContentType = "text/xml";

    context.Response.OutputStream.Write(baResponseData, 0,
      baResponseData.Length);
}
else
    // return error in case of failure.
    context.Response.Status = "500 Internal Server Error";
```

4.2.2 Java Document Hashing Request / Response

Java code for the Hash request from the web application to the ADSS Server

```
//URL of deployed PDF Signer Server
String ADSS_URL = getServletContext().getInitParameter("ADSS_URL");
// URL of ADSS Signer Server, where will be sent
String HASHING_URL = ADSS_URL + "/dhi";
// User name, registered on ADSS Server
String ORIGINATOR_ID =
      getServletContext().getInitParameter("ORIGINATOR_ID");
// Signing profile id that will be used to sign the existing user's empty
// signature field
String USER_PROFILE_ID = getServletContext().getInitParameter(
      "USER_PROFILE_ID");

/* get parameters' values from the applet's http request */
str_signingReason = a_objRequest.getParameter("txt_signingReason");
str_signingLocation = a_objRequest.getParameter("txt_location");
str_contactInfo = a_objRequest.getParameter("txt_contactDetails");
str_targetPDF = a_objRequest.getParameter("hdn_targetPDF");
str_certificate = a_objRequest.getParameter("certificate");

obj_session = a_objRequest.getSession(true);

/* Constructing request for document hashing */
DocumentHashingRequest obj_documentHashingRequest = new
    DocumentHashingRequest( ORIGINATOR_ID, USER_PROFILE_ID,
        str_path + "/" + str_targetPDF + ".pdf",
        Base64.decode(str_certificate));

obj_documentHashingRequest.overrideProfileAttribute(
    SigningRequest.SIGNING_REASON, str_signingReason);
obj_documentHashingRequest.overrideProfileAttribute(
```

```
    SigningRequest.SIGNING_LOCATION, str_signingLocation);
obj_documentHashingRequest.overrideProfileAttribute(
    SigningRequest.CONTACT_INFO, str_contactInfo);

/* Sending request to the ADSS server */
DocumentHashingResponse obj_documentHashingResponse =
    (DocumentHashingResponse) obj_documentHashingRequest.
     send(HASHING_URL);
```

Java code for getting the Hash response from the ADSS Server

```
if (obj_documentHashingResponse.isResponseSuccessfull())
{
    isResponseSuccessfull = true;
    byte_soapResponseBytes =
        obj_documentHashingResponse.getDocumentHash();
   obj_session.setAttribute("DocumentId",
       obj_documentHashingResponse.getDocumentId());
}
else
{
    str_errorMessage = obj_documentHashingResponse.getErrorMessage();
}
```

4.3 Document and Signature Assembly Request

The web application receives the PKCS#7 formatted signature from the applet and generates a document signature Assembly request to the ADSS Server which has stored the document ready for the assembly. The ADSS Server response contains the assembled signed document which could be delivered back to the user's browser for viewing if required.

4.3.1 .Net Document and Signature Assembly Request / Response in C#

C# code for the document signature Assembly request from the web application to the ADSS Server

```
// Create assembly request object
AssemblyADSSRequest assemblyRequest = new AssemblyADSSRequest();

// Getting document id from cookie.
assemblyRequest.DocumentID =
      context.Request.Cookies["UserSettings"]["DocumentId"];

// Getting PKCS#7 from GoSign Applet
Stream reqInputStream = context.Request.InputStream;
byte[] baPKCS7 = new byte[reqInputStream.Length];
reqInputStream.Read(baPKCS7, 0, (int)reqInputStream.Length);

// creating base64 encoded string from received PKCS#7.
assemblyRequest.Signature = Convert.ToBase64String(baPKCS7);

// Providing user name and password for the profile provided above.
assemblyRequest.OrignatorInfo.OrignatorID = "samples_test_client";
//assemblyRequest.OrignatorInfo.Password = "password";
```

```
// Create an ADSS Client object for transporting the assembly request
ADSSClient AssemblyClientRequest = new ADSSClient();
// replace localhost with the the server name and port you have
// configured in your environment.
AssemblyClientRequest.UniformResourceID =
      "http://localhost/adss/signing/sai";

// Generate the SOAP XML
AssemblyClientRequest.Content = assemblyRequest.Generate();

// SendRequest method will send the request to coded URL and return
// the ADSSWebSerCom.Response.IADSSResponse class object which must
// type-cast this interface intelligently as required.
 AssemblyADSSResponse assemblyResponse =
      (AssemblyADSSResponse)AssemblyClientRequest.SendRequest();
```

C# Code for handing the document signature assembly response from the ADSS Server

```
if (String.Compare(assemblyResponse.ResponseStatus, "SUCCESS") == 0)
{
    // Creating Signed pdf file name to save it at physical path.
    String strResultPDFPath =
      String.Concat(context.Request.PhysicalApplicationPath,
      "Resources\\PDFs\\");
    strResultPDFPath = String.Concat(strResultPDFPath,
      assemblyRequest.DocumentID);
    strResultPDFPath = String.Concat(strResultPDFPath, ".pdf");
    FileStream fs = new FileStream(strResultPDFPath, FileMode.Create);

    // Saving received signed document with the name of it's id.
    byte[] baPDFDoc =
      Convert.FromBase64String(assemblyResponse.Document);
    fs.Write(baPDFDoc, 0, baPDFDoc.Length);
    fs.Close();

    // populating response object with the hash and status information.
    context.Response.Status = "200 OK";
    context.Response.ContentType = "text/xml";
    context.Response.OutputStream.Write(null, 0, 0);
}
else
{
    // return error in case of failure.
    context.Response.Status = "500 Internal Server Error";
}
```

4.3.2 Java Document Signature Assembly Request / Response

```
// URL of ADSS Server, where request(s) will be sent
String ASSEMBLY_URL = ADSS_URL + "/sai";

obj_session = a_objRequest.getSession(false);

// PKCS#7 bytes, gneerated by the GoSign applet
InputStream obj_inputStream = a_objRequest.getInputStream();

ByteArrayOutputStream obj_bos = new ByteArrayOutputStream();
// PKCS#7 bytes
b_pkcs7 = obj_bos.toByteArray();

String str_docId = (String) obj_session.getAttribute("DocumentId");
String str_signedDocPath = str_path + "/" + str_docId + ".pdf";

/* Constructing request for signature assembly */
SignatureAssemblyRequest obj_signatureAssemblyRequest = new
      SignatureAssemblyRequest( ORIGINATOR_ID,
            b_pkcs7,
            (String) obj_session.getAttribute("DocumentId"));

/* Sending request to the ADSS server */
SignatureAssemblyResponse obj_signatureAssemblyResponse =
    (SignatureAssemblyResponse) obj_signatureAssemblyRequest.
                    send(ASSEMBLY_URL);
```

Java Code for handing the document signature Assembly response from the ADSS Server

```
/* Setting dummy bytes */
byte_soapResponseBytes = new byte[] {'1', '2', '3'};
if (obj_signatureAssemblyResponse.isResponseSuccessfull())
{
    obj_signatureAssemblyResponse.writeSignedPDFTo(
                        str_signedDocPath);
    isResponseSuccessfull = true;
}
else
{
    str_errorMessage = obj_signatureAssemblyResponse.getErrorMessage();
}
```

**** End of Document ****