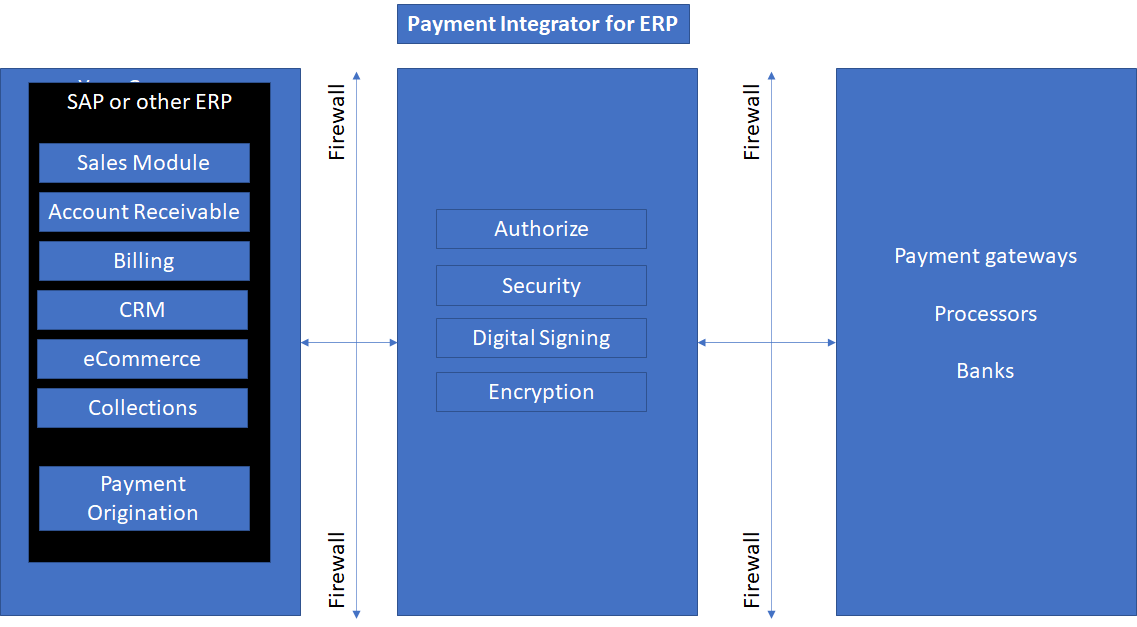
# Integrate ERP Payment processing with banks payment gateways!



How to integrate SAP invoice processing to bank’s payment gateway (NEFT, RTGS, IMPS, A2A) so as to securely automate direct credit to vendors bank account?

Generic solution that I implemented (Given below in order of execution to process payload):

1. Receive disbursement payload from SAP
2. Standardize & validate payload.
3. Generate OAuth security token, that will be used as authentication for allowing us entry into payment gateway
4. Implement two-way SSL authentication for payload.
5. Implement XML digital signature for payload using customer’s private key
6. Base64 encode of the encrypted (IV + digitally signed request) using 32-byte symmetric key
7. Base 64 encode of the encrypted 32-byte symmetric key using banks public key
8. Now we execute the payment gateway end point API (NEFT or RTGS or IMPS or A2A) along with the payload that we secured above
9. The API response is decrypted and its digital signature is verified for authenticity
10. Return the validated response to SAP
11. Based on the response there may be retry attempt from SAP via above solution

# Technical Specifications

1. Input Parameters to web method
2. string hexString: Payment request payload from caller. This is converted to HEX string so as to receive payload of any size.
3. string TransactionId: Dynamically generated by caller to uniquely identify each payment request sent.
4. string debuglog: caller passes either ‘true’ or ‘false’ as string. Given a true value application will write to log file named as ‘transactionid.txt’
5. string pfxcertificatepassword: Password of the .PFX certificate file
6. string pfxcertificatefileName: Name of the .PFX certificate file that exists in the same folder as application.
7. string pemcertificatefileName: Name of the .PEM certificate file that exists in the same folder as application
8. string keyfileName: Name of the .KEY file that exists in the same folder as application.
9. string leafcertificatefileName: Name of the “xxxxLEAF.TXT” file that exists in the same folder as application.
10. string clientid: An id procured by the caller as a part of registration with bank to be passed as “Username” in Authorization headers during bank’s API call
11. string clientsecret: Password associated with clientid given to the caller during registration with bank to be passed as “Password” in the Authorization headers during bank’s API call
12. string scope: A bank specific text field given to the caller as part of registration. This value is added as the query parameter during the call to bank’s API
13. string granttype: This will contain text value that is passed to Bank’s api as query parameter. It will be given as part of registration. Typically for oAuth 2.0 authorization it will contain value of “client\_credentials”
14. string iduser: Passed by the caller to add to the payment payload. This is optional parameter and if this already exists as part of the payload itself then the value passed in the parameter is ignored
15. string groupid: Passed by the caller to add to the payment payload. This is optional parameter and if this already exists as part of the payload itself then the value passed in the parameter is ignored
16. string apikey: The values passed by the caller is obtained during the registration process and is passed as “apikey” in the header during bank’s api call
17. string signaturemethod: Value contains name of the signature method that is used during the digital signing of the payment request. This is provided by the bank as part of registration. For example it can contain “<http://www.w3.org/2001/04/xmldsig-more#rsa-sha256> to indicate RSA-SHA256 algorithm for digital signature
18. string canonicalizationmethod: Value contains name of the canonicalization method that is used during the digital signing of the payment request. This is provided by the bank as part of registration. For example, "http://www.w3.org/TR/2001/REC-xml-c14n-20010315"
19. string oauthtokenurl: URL of the bank’s API to get oAuth token
20. string nefttransferurl: URL of the bank's API for financial transaction- NEFT, RTGS or A2A, to post the payment request to
21. out String responseStatus: This value indicates the status of the call to banks API and extracted from the decrypted response received by the application from bank’s API. Typically it will be either “SUCCESS” or “FAILUER”
22. out String responseStr: This is the decrypted value in its entirety of the encrypted response received by the application from bank’s financial API
23. out String codstatus: This value has the code for the status of the call to banks API and extracted from the decrypted response received by the application from bank’s API. It will be one of the status code provided by the bank during registration
24. out String txtstatus: This value indicates if the payment request is accepted or rejected by the bank’s API and is extracted from the decrypted response received by the application from bank’s API.
25. Converting HEX payment input to simple text -
26. Method Name: ConvertHexStringToByteToString
27. Parameter
    1. string hex: Payment request sent by the caller in HEX format
28. Purpose: This method will convert HEX input string to normal text input string
29. How
    1. First we will need to convert the HEX input to byte array
    2. Find the length of input HEX string
    3. Create a byte array which half the size of length of input HEX string
    4. Fill in each element of the byte array by converting the substring of HEX input string

for (int i = 0; i < bytes.Length; i++)

{

bytes[i] = Convert.ToByte(hex.Substring(i \* 2, 2), 16);

}

* 1. Now encode the byte array with UTF 8 encoding to convert the byte array to plain text

1. Output: returns the converted plain text payment request to the caller
2. Digitally signing the payload
3. Method Name: NewSignPayloadXML
4. Parameters
   1. RuntimeConfigurations cfg: This holds all the parameter values passed to web method
   2. string inputXml: This is the XML string which is converted from HEX input
5. Purpose: Bank;s API requires that payment request payload to be digitally signed so as to ensure that no one has tampered with (even looked at) request payload. The logic uses signature hash using RSA SHA-256 along with customers private key and also adds the certificate to the signature under X509Data tag
6. How
   1. First add the “id” attribute to faml or faxml tag
   2. Make sure “iduser” and “groupid” element tags exist in the payload under “faml” or “faxml”. If they don’t then please add those elements and to assign the respective value from “cfg” object
   3. Creating X509Certificate2
      1. Read PEM text from the .PEM certificate file
      2. Read KEY from the .KEY file. This is the RSA private key
      3. Read the bytes between the “BEGIN CERTIFICATE” and “END CERTIFICATE” tags from the variable that holds certificate file content
      4. Read the bytes between the “BEGIN RSA PRIVATE KEY” and “END RSA PRIVATE KEY” tags from the variable that holds key file content
      5. Use the certificate bytes to create X509Certificate2 object
      6. Now we have to parse the key byte read above using ASN.1 format to get the RSA service provider
      7. Assign the RSP service provider object created above as PrivateKey to the X509Certificate2 object
   4. Signing of XML
      1. Create SignedXml object using modified payload xml
      2. Assign RSA Private key from the X509Certificate2 object to signed xml object
      3. Create Signature object using signed xml object’s signature
      4. Assign Signature method value from cfg object to Signature object’s Signedinfo.SignatureMethod variable
      5. Assign Canonicalization method value from cfg object to Signature object’s SignedInfo.CanonicaliazationMethod variable
      6. Get the signature variable value from the signed xml object
      7. Add “#” + “iduser” to a reference object
      8. Add the reference object to the signature variable
      9. Create KeyInfoX509Data object by passing X509Certificate2 object
      10. Use the SubjectName.Name from X509Certificate2 object and assign it to KeyInfoX509Data object
      11. Create a KeyInfo object and assign KeyInfoX509Data object as a clause
      12. Assign the KeyInfo object to the Signature objects KeyInfo variable
      13. Compute the signature of signed xml object
      14. Get XML from signed xml object
   5. Import the above XML into the XML payload
   6. Return the updated XML
7. Generate 32 byte symmetric private key
8. Method Name: GenerateSymmetricKey
9. Parameters:
   1. out byte[] encryptionKeyBytes: The key generated in this method is returned
10. Purpose: Genarating private 32 byte key to be used to encrypt the signed xml
11. How
    1. We use the Random number generator to create the key
    2. Character Range allowed: ((character <= Z && character >= A) || (character <= z && character >= a) || (character <= NINE && character >= ZERO))
    3. No special characters are allowed

Random RNG = new Random();

int length = StringLength;

var rString = "";

for (var i = 0; i < length; i++)

{

rString += ((char)(RNG.Next(1, 26) + 64)).ToString().ToLower();

}

return rString;

* 1. Use UTF8Encoding to get the bytes from above random string
  2. Assign the byte array to the out parameter

1. Encrypt signed XML
2. Method Name: NewEncryptSignedXML
3. Parameters
   1. String toEncrypt: The signed XML to be encrypted
   2. byte[] key: The 32 byte private key generate above
   3. returns the encrypted signed XML as byte array
4. Purpose: To encrypt the signed XML using the generated 32 byte private key
5. How
   1. Use UTF8 encoding to get byte array from input signed XML
   2. Create AesCryptoServiceProvider object
   3. Assign the input 32 byte key to AesCryptoServiceProvider object’s key
   4. Generate random string of 16 bytes and use UTF8Encoding to get byte array
   5. Assign the 16 byte random byte array to AesCryptoServiceProviderobject “IV” member
   6. Assign CipherMode.CBC to AesCryptoServiceProvider object’s mode member
   7. Assign PaddingMode.PKCS7 to AesCryptoServiceProvider object’s padding member
   8. Create symmetric AES encryptor object using AesCryptoServiceProvider object’s key and IV
   9. Create a new memory stream object
   10. Write AesCryptoServiceProvider object’s IV member value to memory stream object
   11. Use symmetric AES encryptor object and memory stream object and create CryptoStream object in write mode
   12. Write the byte array of input signed XML to the CryptoStream object
   13. Return the byte array from the memory stream
6. Base64 encode the Encrypted value
7. Method Name: EncodeByteArrayToBase64String
8. Parameters
   1. byte[] value: byte array to encode
   2. returns a string holding Base64 encoded value of byte array
9. Purpose: Base64 encode the Encrypted value. This is the value that will get sent to banks’ API in the json object
10. How
    1. Use the ToBase64String to convert byte array
11. Encrypt Base 64 encoded 32 byte private key
12. Method Name: NewEncryptionEncodeSymmetricKey
13. Parameters:
    1. RuntimeConfigurations cfg: Configuration object holding all values passed to web method
    2. byte[] encryptionKeyBytes: The generated 32 byte private key that was used to encrypt the signed XML payload
14. Purpose: The bank API will require the 32 byte private key, that we generated & used to encrypt the signed XML payload, to decrypt the signed payload. Hence we need to pass the key to bank API which is base 64 encoded and encrypted before passing
15. How
    1. Create X509Certificate2 object using the “---LEAF.TXT” file
    2. From the X509Certificate2 object get the RSA public key and export the same in RSAParameter object
    3. Create a RSACryptoServiceProvider object and import the RSAParameter object
    4. Use RSACryptoServiceProvider object to encrypt the input 32 byte private key using RSAEncryptionPadding.Pkcs1
    5. Use the EncodeByteArrayToBase64String method to encode the encrypted private key and return the string
16. Generate oAuth token from bank oauth api
17. Method Name: GenerateOAuthToken
18. Parameters:
    1. RuntimeConfigurations cfg: Configuration object holding values passed to the web method
    2. Returns valid oAuth token if successful else null
19. Purpose: Bank api requires oAuth token to get our call to financial api authenticated using oauth token returned by bank’s oauth api. This method will use the oAuth api url to get the token
20. How
    1. Create a string containing clientid and client secret
       1. var authenticationString = $"{cfg.clientid}:{cfg.clientsecret}";
    2. Get byte array of above string and base 64 encode the same
    3. Create X509Certificate2 object using the .PFX certificate file and PFX password from configuration object
    4. Create HTTPClientHandler object and assign it X509Certificate2 object
    5. Create a HTTPClient object using the HTTPClientHandler object
    6. Use base 64 encoded authentication string and “Basic” auth type to create DefaultRequestHeader.Authorization object In the HTTPClient object
    7. Create an empty list of KeyValuePair
       1. var postData = new List<KeyValuePair<String, String>>();
    8. Create HTTPContent object by passing the above list to FormUrlEncodedContent
    9. Set HTTPContent object’s header’s content type to "application/x-www-form-urlencoded" using MediaTypeHeaderValue
    10. Use the HTTPClient object to post to bank api and get the result and get the oauth token json object as below

var responseResult = client.PostAsync(cfg.oauthtokenurl + "?" + "grant\_type=" + cfg.granttype + "&" + "scope=" + cfg.scope, content).Result;

if (!responseResult.IsSuccessStatusCode)

{

WriteToLog(cfg, "oAuth URL Failed: " + responseResult.Content.ReadAsStringAsync().Result, enforce: true);

throw new HttpRequestException(responseResult.Content.ReadAsStringAsync().Result);

}

var jsonContent = responseResult.Content.ReadAsStringAsync().Result;

oAuthToken = JsonConvert.DeserializeObject<OAuthToken>(jsonContent);

* 1. Return the oAuth token json object that will be used during call to bank’s financial API

1. Execute bank’s financial API (NEFT or RTGS or A2A)
2. Method Name: ExecuteTransactionAPI
3. Parameters:
   1. RuntimeConfigurations cfg: Configuration object that holds the passed value to web method
   2. String encodedData: The signed & encrypted XML contained payment request
   3. String encryptedencodedKey: The encrypted and base 64 encoded 32 byte private key generated and used to encrypt the signed payload
   4. OAuthToken oAuthToken: oAuth token object received from bank’s oAuth API
   5. string TransactionId: Caller supplied transaction id
   6. Returns the response received from bank’s financial API
4. Purpose: After signing and encrypting payload and receiving the oAuth token we call the respective bank’s financial API, so that bank can add the payment request payload to its queue
5. How
   1. Create a new json object containing all the required information

var requestPayload = new RequestPayload

{

RequestSignatureEncryptedValue = encodedData,

SymmetricKeyEncryptedValue = encryptedencodedKey,

Scope = cfg.scope,

TransactionId = transactionId,

OAuthTokenValue = oAuthToken.AccessToken

};

var stringPayload = JsonConvert.SerializeObject(requestPayload);

* 1. Create HTTPContent object using StringContent by passing serialized payload created above
  2. Add "application/json"content as content type to the header of HTTPContent object using MediaTypeHeaderValue
  3. Create X509Certificate2 object using .PFX file and PFX password from configuration object
  4. Create a HTTPClientHandler object and the X509Certificate2 object to its certificate property
  5. Create HTTPClient object using the HTTPClientHandler object
  6. Add key from configuration object to HTTPClient object’s default headers using name as “apikey”
  7. Now call the respective bank’s financial API and get the results

var httpResponse = client.PostAsync(cfg.targeturl, httpContent).Result;

* 1. Read the response as string and deserialize the response data and return the same

if (httpResponse.Content != null)

{

var responseContent = httpResponse.Content.ReadAsStringAsync();

// From here on you could deserialize the ResponseContent back again to a concrete C# type using Json.Net

var jsonContent = httpResponse.Content.ReadAsStringAsync().Result;

WriteToLog(cfg, "Result from API URL: " + jsonContent);

responseData = JsonConvert.DeserializeObject<ResponsePayload>(jsonContent);

}

* 1. If the response status has value of “SUCCESS” then we will decrypt the response and return the decrypted response to caller
  2. If the response status does not have value of “SUCCESS” then we will return 0 as the return code along with status value

1. Decrypt the symmetric key received in response received from bank’s financial API
2. Method Name: DecryptDecodeReceivedKey
3. Parameters:
   1. RuntimeConfigurations cfg: Configuration object containing values passed to web method
   2. string receivedKey: The encrypted key received in response from bank;s financial API
   3. Returns the byte array containing decrypted key
4. Purpose: We need to decrypt the received key so that we can decrypt the exact response
5. How:
   1. Convert the received key to a byte array using FromBase64String
   2. Now we create X509Certificate2 object using the .PEM certificate file and .KEY file from configuration object

using (TextReader tr = new StreamReader(AppDomain.CurrentDomain.BaseDirectory + cfg.pemcertificatefile))

{

pemText = tr.ReadToEnd();

WriteToLog(cfg, "PEM-TEXT:" + pemText);

}

WriteToLog(cfg, "Before reading key file:" + AppDomain.CurrentDomain.BaseDirectory + cfg.keyfile);

using (TextReader tr = new StreamReader(AppDomain.CurrentDomain.BaseDirectory + cfg.keyfile))

{

keyText = tr.ReadToEnd();

WriteToLog(cfg, "KEY-TEXT:" + keyText);

}

WriteToLog(cfg, "Before Certificate(pemText, keyText, pass)");

Certificate cert = new Certificate(pemText, keyText, pass);

WriteToLog(cfg, "Before cert.GetCertificateFromPEMstring(false)");

certificate = cert.GetCertificateFromPEMstring(cfg, false);

* 1. Get the RSAPrivateKey from certificate object and export the parameters in RSAParameter object
  2. Create a RSACryptoServiceProvider object and import the parameters of RSAPrivateKey
  3. Use the RSACryptoServiceProvider object to decrypt the decoded key into byte array and return the byte array

1. Decrypt & check the digital signature of the response value received in response from bank’s financial API using decrypted key
2. Method Name: DecryptDecodeReceivedXML
3. Parameters:
   1. RuntimeConfigurations cfg: Configuration object containing values passed to web method
   2. byte[] key: Decrypted key to be used to decrypt response value
   3. string encodedencryptedXML: actual response received from bank API that is encoded and encrypted
   4. out string returnStr: Decoded and decrypted actual response
4. Purpose: Bank’s API returns response that is encoded and encrypted. This has to be decoded and decrypted in order for the caller to understand what happened to the call that was made to bank’s API
5. How:
   1. Decode the encoded & encrypted input into byte array using FromBase64String
   2. Decrypt the decoded array using described below

using (var provider = new AesCryptoServiceProvider())

{

provider.Key = encryptionKey;

provider.Mode = CipherMode.CBC;

provider.Padding = PaddingMode.PKCS7;

using (var ms = new MemoryStream(encryptedString))

{

byte[] buffer = new byte[16];

ms.Read(buffer, 0, 16);

provider.IV = buffer;

using (var decryptor = provider.CreateDecryptor(provider.Key, provider.IV))

{

using (var cs = new CryptoStream(ms, decryptor, CryptoStreamMode.Read))

{

byte[] decrypted = new byte[encryptedString.Length];

var byteCount = cs.Read(decrypted, 0, encryptedString.Length);

//return Encoding.UTF8.GetString(decrypted, 0, byteCount);

return \_encoder.GetString(decrypted, 0, byteCount);

}

}

}

}

* 1. Create a XMLDocument object using the decrypted XML input
  2. Create a SigndXML object using the XMLDocument object
  3. Find the signature element from the SignedXML object by finding the element by tag name “Signature”
  4. Use the signature element’s value of 0th index and load it into SignedXML object
  5. Now get the “faml” or “faxml” node from the XMLDocument object
  6. From the “faml” or “faxml” node get the value of attribute “id” and create a string such as,

string idattrib = "#" + faxmlNode.Attributes["Id"].Value;

* 1. Compare and make sure the URI from SignedXML obejct’s 0th references value matches to id attribute string
  2. Create X509Certificate2 object using the “xxxxLEAF.TXT” file from configuration object
  3. Use the CheckSignature method of SignedXML object and pass X509Certificate2 object to check the validity of the SignedXML object
  4. Remove the id attribute from “faml” or “faxml” node
  5. Return the outer xml from this node to the caller

1. Parse the response and return to the caller
2. Method Name: GetResultsFromNEFTResponse
3. Parameters:
   1. RuntimeConfigurations cfg: Configuration object containing values passed to web method
   2. String responseStr: Decoded & decrypted and validated response value
   3. out String codstatus: Transaction status
   4. out String txtstatus: Status description
4. Purpose: This method will parse the input response to get the numeric status and its description to be returned to the caller
5. How:
   1. Create a XMLDocument object using the response string
   2. Create a XMLNodeList object by finding “codstatus” element tag name
   3. Assign the 0th items value from the node list to codstatus parameter
   4. Create a XMLNodeList object by finding “txtstatus” element tag name
   5. Assign the 0th items value from the node list to txtstatus parameter