Revenue
Cáin agus Custaim na hÉireann
Irish Tax and Customs

Customs & Excise

REST Web Service Integration Guide

The information in this document is provided as a guide only and is not professional advice, including legal advice. It should not be assumed that the guidance is comprehensive or that it provides a definitive answer in every case.

Contents

Version Control		
Version	**Date**	**Change**
0.1	04/10/2019	Initial document published

Document References

Reference
1. Customs & Excise Web Services Specifications (incl AIS) Operational November 2020
2. Customs & Excise SOAP Web Service Integration Guide
3. Customs & Excise REST Web Service Integration Guide

1. Introduction

This document details the REST/XML web services specification provided by Revenue Customs and Excise division.

The Documents Homepage specified in Document References is the home to all technical documentation, specification, and examples for the above web services which has been made available to enable software developers to update their software packages to be compatible with REST web service specification.

Please note: The AIS web services are made available via REST/XML and the request message needs to be digitally signed as described in Digital Signatures section.

This document assumes familiarity with the REST web services. A full description of each of these can be found in the "*Customs & Excise Web Services Specifications (incl. AIS) Operational November 2020"* document.

2. Digital Signatures

Any ROS web service request that either returns confidential information or accepts submission of information must be digitally signed. This must be done using a digital certificate that has been previously retrieved from ROS.

The digital signature must be applied to the message in accordance with the HTTP Signatures specification.

The digital signature ensures the integrity of the document. By signing the document, we can ensure that no malicious intruder has altered the document in any way. It can also be used for non-repudiation purposes.

If a valid digital signature is not attached, a HTTP status code of 401 (Unauthorised) will be returned. The message body will provide more information on the details of the problem.

2.1 HTTP Signatures

The HTTP signatures protocol is intended to provide a simple and standard way for clients to sign HTTP requests. A summary of the structure of a HTTP Signature is outlined below. This is a simplified explanation of the HTTP Signatures specification. The full specification can be found at Signing HTTP Messages and should be read in full. The specification defines two approaches to building a HTTP signature, "The 'Signature' HTTP Authentication Scheme" and "The 'Signature' HTTP Header", Revenue uses the latter.

At a high level, a HTTP Signature is a HTTP header that is added to a HTTP request. It is comprised of a set of components that were used to generate a digital signature and the digital signature itself.

2.1.1 HTTP Signature Sample

Below is a sample HTTP Signature header.

```
Signature: keyId="MIICfzCCAeigAwIBAgIJ... // truncated",
           algorithm="rsa-sha512",
           headers="(request-target) host date digest",
           signature="GdUqDgy94Z8mSYUjr/rL6qrLX/jmudS... // truncated"
```

2.1.2 HTTP Signature Components

The Signature HTTP header contains four components, ***keyId***, ***algorithm***, ***headers*** and ***signature***. Below is a description of each.

keyId: The keyId field must contain a Base64 encoded version of the X509 certificate that accompanies the private key used to sign the message. This field is required.

algorithm: The `algorithm` parameter is used to specify the digital signature algorithm to use when generating the signature. Revenue expects this to be '***rsa-sha512***'. This field is required.

headers: The `headers` parameter specifies the list of headers used when generating the signature for the message. The parameter must be a lowercased, quoted list of HTTP header fields, separated by a single space character. The list order is important, and MUST be specified in the order the HTTP header field-value pairs are concatenated together during signing.

signature: The signature component is a base 64 encoded digital signature. The implementer uses the `algorithm` and `headers` field to form a canonicalized `signing string`. This `signing string` is then signed with the private key that accompanies the X509 certificate associated with the `keyId` field and the algorithm corresponding to the `algorithm` field. The `signature` field is then base 64 encoded.

2.1.3 Signature String Construction

In order to generate the string to be signed, the implementer MUST use the values of each HTTP header defined in the `headers` signature field, to build the signature string. Values must be in the order they appear in the `headers` signature field. If the associated HTTP header does not exist, it should be added to the HTTP request BEFORE attempting to construct this string.

Allowable values in the headers field are outlined in the table below.

Value	Mandatory
* (request-target)	Yes
host	Yes
date	Yes
** x-date	Yes, if date header cannot be added.
*** digest	Yes, if HTTP method is of type POST
content-type	No
content-length	No
x-http-method-override	If HTTP method is of type POST, HTTP header *'X-HTTP-Method-Override'* exists and *'Content-Type=application/x-www-form-urlencoded'*. See Section 2.1.1 for more detail.

* The `(request-target)` header field is a special header field in that its value is comprised of 2 HTTP headers. It is generated by concatenating the lowercase HTTP method, an ASCII space, and the request path headers.

** The 'x-date' headers field value should ONLY be used in conjunction with the X-Date HTTP header if a Date HTTP header cannot be added to the HTTP request programmatically. The Date header has a limitation when using JavaScript in a browser to build and send a HTTP signature. The limitation is that you cannot add a 'Date' HTTP header when executing JavaScript in a browser. The native XmlHttpRequest object prohibits addition of a 'Date' HTTP header. Building the signature string that will be signed with an 'x-date' header instead of a 'date' header removes this restriction.

*** The 'Digest' HTTP header is created using the POST body/payload. The payload should be converted to a byte array, hashed using the SHA-512 algorithm and finally base64 encoded before adding it as a HTTP header.

All other header field values are created by concatenating the lowercase header field name followed by an ASCII colon `:`, an ASCII space ` `, and the header field value. Leading and trailing whitespace in the header field value MUST be omitted. If the header field is not the last value defined in the `headers` signature field, then append an ASCII newline `\n`

2.1.4 Signature Creation

The signature component is a base 64 encoded digital signature. The implementer uses the `algorithm` and constructed Signature String. The Signature String is signed with the private key that accompanies the X509 certificate associated with the `keyId` field and the algorithm corresponding to the `algorithm` field. The `signature` field is then base 64 encoded.

Appendix A – Extracting from a .p12 File

Each customer of ROS will have a digital certificate and private key stored in an industry standard PKCS#12 file.

In order to create a digital signature, the private key of the customer must be accessed. A password is required to retrieve the private key from the P12 file. This password can be obtained by prompting the user for their password.

The password on the P12 is not the same as the password entered by the customer. It is in fact the MD5 hash of that password, followed by the Base64-encoding of the resultant bytes.

To calculate the hashed password, follow these steps:

1. First get the bytes of the original password, assuming a "Latin-1" encoding. For the password "Password123", these bytes are: 80 97 115 115 119 111 114 100 49 50 51(i.e. the value of "P" is 80, "a" is 97, etc.).
2. Then get the MD5 hash of these bytes. MD5 is a standard, public algorithm. Once again, for the password "Password123" these bytes work out as: 66 -9 73 -83 -25 -7 -31 -107 -65 71 95 55 -92 76 -81 -53.
3. Finally, create the new password by Base64-encoding the bytes from the previous step. For example, the password, "Password123" this is “QvdJref54ZW/R183pEyvyw==”.

This new password can then be used to open a standard ROS P12 file.