Sign Requests

Version: 20230905

Online Version

Contents

When you make requests to APIs, you must provide a signature to verify your identity and ensure data security. This topic describes how to generate a signature in a request.

The integration of cloud development SDKs can relieve you from the tedious process of generating API signatures. For more information, see Service SDKs.

1 Signature algorithm

HMAC-SHA256 is used in API services to create a message digest. Different signature algorithms are applied to **token management API requests** and **general API requests**, as described in the following table.

API type
Token management API

```
    <th>General business API</td>
</tr>
<tr>
    <th>Scope of application</th>
    <td>Requests that are made to get or refresh tokens</td>
    <td>Requests that are made to manage business rather than tokens.</
       td>
</tr>
<tr>
    <th>Signature algorithm</th>
    <td>
        str = client_id + t + nonce + stringToSign<br>
        sign = HMAC-SHA256(str, secret).toUpperCase()
    </td>
    <td>
        str = client_id + access_token + t + nonce + stringToSign<br>
        sign = HMAC-SHA256(str, secret).toUpperCase()
    </td>
</tr>
 <tr>
    <th>Field description</th>
    <td colspan="2">
        <ul><li>client_id: the key of an authentication key pair.</li>
        <li>t: the 13-digit standard timestamp.</li>
        <li>nonce: the universally unique identifier (UUID) generated
           for each API request. Combined with the timestamp, the UUID
           ensures the uniqueness of API requests. This field is
           optional.</li>
        <li>stringToSign: the signature string. For more information,
           see stringToSign.</li>
        <li>secret: the value of an authentication key pair.</li>
        <li>access_token: the access token. The field value is obtained
            from the authentication key pair.</li></ul>
    </td>
</tr>
<tr>
    <th>Field description</th>
    <td colspan="2">
        Concatenate the value of client_id, the 13-digit standard
           timestamp (t) of the specified request, nonce, and
           stringToSign to create a string named as str. Use HMAC-
           SHA256 to create a hash digest value based on the string str
            and the value of the secret. Capitalize all letters of the
           hash digest value to get a signature.
    </td>
</tr>
```

Different from token management API requests, in the signature algorithm for general API requests, `access_token` is additionally concatenated to generate the string

str.

2 stringToSign signature string

2.1 Structure

stringToSign is created in the following structure:

```
String stringToSign=
HTTPMethod + "\n" +
Content-SHA256 + "\n" +
Headers + "\n" +
URL
```

This structure consists of the following four parts:

- HTTPMethod
- Content-SHA256
- Headers
- URL

You can concatenate the strings of these four parts with line-feed characters (\n) to create a string-to-sign named as stringToSign.

2.2 HTTPMethod

HTTPMethod represents an API method, such as GET, POST, PUT, and DELETE.

2.3 Content-SHA256

Content-SHA256 represents the SHA256 value of a request body. SHA256 is calculated only when the body is not a form.

Calculation:

```
String content-SHA256 = SHA256(bodyStream.getbytes("UTF-8")); // bod
yStream is a byte array.
```

An empty body is still encrypted into e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b85
.

2.4 Headers

Headers represents the string in which all request headers involved in the signature calculation are concatenated with line-feed characters (\n) as described in Request Structure. Each request header is a key-value pair. For example:

1. The request parameters are as follows:

```
client_id: 1KAD46OrT9HafiKd****
sign: C5EFD19AD45E33A060C0BE47AEF65D975D54B2D70CBAA7A1ACA1A7D0E5C0
      **
**
sign_method: HMAC-SHA256
t: 1588925778000
access_token: 3f4eda2bdec17232f67c0b188af3****
nonce: 5138cc3a9033d69856923fd07b49****
...
```

2. To improve data security, add two custom fields area_id and request_id to the signature calculation.

```
area_id: 29a33e8796834b1****
request_id: 8afdb70ab2ed11eb85290242ac13****
```

3. The new request structure as follows:

```
client_id: 1KAD46OrT9HafiKd****
sign: C5EFD19AD45E33A060C0BE47AEF65D975D54B2D70CBAA7A1ACA1A7D0E5C0
      **
**
sign_method: HMAC-SHA256
t: 1588925778000
access_token: 3f4eda2bdec17232f67c0b188af3****
nonce: 5138cc3a9033d69856923fd07b49****
...
  // Two request headers are added
Signature-Headers: area_id:request_id
 area_id: 29a33e8796834b1**** // Subject to your customization
request_id: 8afdb70ab2ed11eb85290242ac13****  // Subject to your
   cus
tomization
```

> All request header fields involved are concatenated with colons (:). As shown in the preceding code block, area_id:request_id is generated as the value of Signature-Headers.

Then, concatenate all key-value pairs included in Signature-Headers to generate the value of Headers in the following structure:

```
String Headers =
HeaderKey1 + ":" + HeaderValue1 + "\n" +
HeaderKey2 + ":" + HeaderValue2 + "\n" +
...
HeaderKeyN + ":" + HeaderValueN + "\n"
```

In this example, the following value of Headers is obtained:

```
String Headers =
area_id + ":" + 29a33e8796834b1**** + "\n" +
request_id + ":" + 8afdb70ab2ed11eb85290242ac13**** + "\n"
```

2.5 URL

A URL represents the request path into which a request path and form parameters are concatenated.

- Concatenate URLs: Sort form parameters by keys in alphabetically ascending order and concatenated in the following way. If query parameters or form parameters are empty, set the URL to the request path that is not suffixed with a question mark (?).

```
String url =
Path +
"?" +
Key1 + "=" + Value1 +
"&" + Key2 + "=" + Value2 +
"&" + Key3 +
...
"&" + KeyN + "=" + ValueN
```

For example:

```
String url = /v1.0/iot-03/devices/87707085bcddc23a5fa3/logs?
    end_time
=1657263936000&event_types=1&start_time=1657160836000
```

- Sort parameters in alphabetically ascending order. If multiple parameters have the same initial letter, sort these parameters by their second letter alphabetically. This way, letters are sequentially compared among different parameters until the parameters can be sorted as expected.

  Take the following parameters as an example:

  - start_time=1657160836000
  - end_time=1657263936000
  - event_types=1

    They are sorted in the following alphabetical order:

  - end_time=1657263936000
  - event_types=1
  - start_time=1657160836000

    In these parameters, end_time and event_types have the same initial letter, and the second letter n of end_time comes earlier in the alphabet than the second letter v of event_types. Therefore, end_time is followed by event_types .

    The first letter of end_time and event_types comes earlier in the alphabet than the first letter of start_time. As a result, end_time and event_types are followed by start_time.

3 Token management API example

In the following example, make an API request to get a token. Set `grant_type` to `1` and leave the body parameter empty.

```
GET:/v1.0/token?grant_type=1
```

The request header structure is as follows:

Parameter	Value
method	GET
client_id	1KAD46OrT9HafiKdsXeg
secret	4OHBOnWOqaEC1mWXOpVL3yV50s0qGSRC
t	1588925778000
sign_method	HMAC-SHA256
nonce	5138cc3a9033d69856923fd07b491173
Signature-Headers	area_id:call_id
area_id (**Customized and used in signature calculation**)	29a33e8796834b1efa6
call_id (**Customized and used in signature calculation**)	8afdb70ab2ed11eb85290242ac130003

3.1 Concatenate a `stringToSign`

Generate the string-to-sign `stringToSign` in the following structure:

```
GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852  b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003
/v1.0/token?grant_type=1
```

The structure is described as follows:

GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003

/v1.0/token?grant_type=1

Description:
The string in red represents HTTPMethod.
The string in orange represents Content-SHA256.
The string in green represents signature header fields customized by you.
The string in blue represents the URL.

3.2 Create a string that includes the string-to-sign

After `stringToSign` is generated, you can use it to create the string `str`:

```
1KAD46OrT9HafiKdsXeg1588925778000 5138cc3a9033d69856923fd07b4  91173GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852  b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003
/v1.0/token?grant_type=1
```

The structure is described as follows:

1KAD46OrT9HafiKdsXeg1588925778000 5138cc3a9033d69856923fd07b491173GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003

/v1.0/token?grant_type=1

Description:
The string in red represents `client_id`.
The string in orange represents timestamp `t`.
The string in green represents `nounce`.
The string in blue represents `stringToSign`.

3.3 Use HMAC-SHA256 to create a message digest

Hash digest value:

```
HMAC-SHA256(1KAD46OrT9HafiKdsXeg1588925778000 5138cc3a9033d69  856923fd
07b491173GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852  b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003
/v1.0/token?grant_type=1,4OHBOnWOqaEC1mWXOpVL3yV50s0qGSRC)
```

- Create a hash digest value and encode the hash digest value into a new string:

  9e48a3e93b302eeecc803c7241985d0a34eb944f40fb573c7b5c2a82158af13e

- Capitalize all letters of the new string:

  9E48A3E93B302EEECC803C7241985D0A34EB944F40FB573C7B5C2A82158AF13E

4 General business API example

In the following example, make an API request to get a list of users. Set `schema` to `apps` and leave the body parameter empty.

Parameter	Value
URL	/v2.0/apps/schema/users
method	GET
client_id	1KAD46OrT9HafiKdsXeg
secret	4OHBOnWOqaEC1mWXOpVL3yV50s0qGSRC
t	1588925778000
access_token	3f4eda2bdec17232f67c0b188af3eec1
sign_method	HMAC-SHA256
nonce	5138cc3a9033d69856923fd07b491173
Signature-Headers	area_id:call_id
area_id (**Customized and used in signature calculation**)	29a33e8796834b1efa6
call_id (**Customized and used in signature calculation**)	8afdb70ab2ed11eb85290242ac130003
page_no	1
page_size	50

4.1 Concatenate a `stringToSign`

Generate the string-to-sign `stringToSign` in the following structure:

```
GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852  b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003
/v2.0/apps/schema/users?page_no=1&page_size=50
```

The structure is described as follows:

GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003

/v2.0/apps/schema/users?page_no=1&page_size=50

Description:
The string in red represents HTTPMethod.
The string in orange represents Content-SHA256.
The string in green represents signature header fields customized by you.
The string in blue represents the URL.

4.2 Create a string that includes the string-to-sign

After `stringToSign` is generated, you can use it to create the string `str`:

```
1KAD46OrT9HafiKdsXeg3f4eda2bdec17232f67c0b188af3eec115889257  78000513
8cc3a9033d69856923fd07b491173GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852  b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003
/v2.0/apps/schema/users?page_no=1&page_size=50
```

The structure is described as follows:

1KAD46OrT9HafiKdsXeg3f4eda2bdec17232f67c0b188af3eec1158892577800051 38cc3a9033d69856923fd07b491173GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003

/v2.0/apps/schema/users?page_no=1&page_size=50

Description:
The string in red represents `client_id`.
The string in purple represents `access_token`
The string in orange represents timestamp `t`.
The string in green represents `nounce`.
The string in blue represents `stringToSign`.

4.3 Use HMAC-SHA256 to create a message digest

Hash digest value:

```
HMAC-SHA256(1KAD46OrT9HafiKdsXeg3f4eda2bdec17232f67c0b188af3  eec11588
9257780005138cc3a9033d69856923fd07b491173GET
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852  b855
area_id:29a33e8796834b1efa6
call_id:8afdb70ab2ed11eb85290242ac130003
/v2.0/apps/schema/users?page_no=1&page_size=50,4OHBOnWOqaEC1  mWXOpVL3
yV50s0qGSRC)
```

- Create a hash digest value and encode the hash digest value into a new string:

  ae4481c692aa80b25f3a7e12c3a5fd9bbf6251539dd78e565a1a72a508a88784

- Capitalize all letters of the new string:

  AE4481C692AA80B25F3A7E12C3A5FD9BBF6251539DD78E565A1A72A508A88784

5 Implement the HMAC-SHA256 authentication scheme

5.1 Sample code for Java

Download the sample code for Java.

5.2 Example for Golang

Download the sample code for Golang.

5.3 Sample code for Node.js

Download the sample code for Node.js.

5.4 Sample code for JavaScript

```
/**
Run the code online with this jsfiddle. Dependent upon an open sourc
e js library calledhttp://code.google.com/p/crypto-js/.
**/
<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.0.0/
enc-base64.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.0.0/
hmac-sha256.min.js"></script>
<script>
  var hash = CryptoJS.HmacSHA256("str", "secret");
  var hashInBase64 = hash.toString().toUpperCase();
  document.write(hashInBase64);
</script>
```

5.5 Sample code for PHP

```
/**
PHP has built-in methods for hash_hmac (PHP 5) and base64_encode (PH
P 4, PHP 5) resulting in no outside dependencies. Say what you want
about PHP but they have the cleanest code for this example.
**/
$s = strtoupper(hash_hmac("sha256", "str", 'secret'));
echo var_dump($s);
```

5.6 Sample code for C

```
using System;
using System.Security.Cryptography;
namespace Test
{
  public class MyHmac
  {
    public static string Encrypt(string str, string secret)
            {
                secret = secret ?? "";
                var encoding = new System.Text.UTF8Encoding();
                byte[] keyByte = encoding.GetBytes(secret);
                byte[] messageBytes = encoding.GetBytes(str);
                using (var hmacsha256 = new HMACSHA256(keyByte))
                {
                    byte[] hashmessage = hmacsha256.ComputeHash(mess
ageBytes);
                    StringBuilder builder = new StringBuilder();
                    for (int i = 0; i < hashmessage.Length; i++)
                    {
                        builder.Append(hashmessage[i].ToString("x2")
);
                    }
                    return builder.ToString().ToUpper();
                }
            }
  }
}
```

6 FAQs

6.1 How do I verify the encrypted signature?

During local development, you can test API requests with Postman to verify the encrypted signature. For more information, see Verify Signature Result.

6.2 Why does a blank line exist in `stringToSign`?

This depends on the structure of `stringToSign`. `stringToSign` consists of `HTTPMethod`, `Content-SHA256`, `Headers`, and `URL`, concatenated with a line-feed character (\n) in between. The structure is as follows:

```
String stringToSign=
HTTPMethod + "\n" +
Content-SHA256 + "\n" +
Headers + "\n" +
URL
```

The following example shows how to calculate `Headers`:

```
String Headers =
HeaderKey1 + ":" + HeaderValue1 + "\n" +
HeaderKey2 + ":" + HeaderValue2 + "\n" +
...
HeaderKeyN + ":" + HeaderValueN + "\n"
```

Therefore, the `Headers` string is followed by two line-feed characters (\n) that result in a blank line.