



## Securing Web Services with Labview UNED, 4<sup>th</sup> December 2014

### WP 4.2 - "Services Adaptation to Connect Remote Labs to VLE"

**WARNING: This tutorial changes the way the Web Services are accessed. This means that the laboratory client application must be updated with the security options described in the last part of the tutorial.**

#### Introduction

This example is built with LabView 2013. It could happen that some screens are different in other versions.

You can use **API keys** to restrict which clients can send HTTP requests to HTTP method VIs. An API key is a string of seemingly random characters that consists of two parts, an *access ID* and a *secret ID*. The access ID works similarly to a username, and the secret ID works similarly to a password intended only for authorized clients.

At runtime, the Web services runtime engine inspects every client request as it arrives. When a request matches a secured URL mapping, the runtime looks for the "**digital signature**" in the request. If this signature is missing or incorrectly formed, the request is rejected. If the signature is present, the runtime looks at the request and calculates what the signature should be for that request. It then compares its calculated signature with the one actually present in the message. If the two do not match, the request is rejected.

When a secured request is rejected, the client receives an HTTP response containing an "**HTTP/1.1 403 Forbidden**" error, indicating that the request has been rejected because the client was not authorized to make that request.

The algorithm for creating signatures includes the time of day of the request, and the runtime is configured to reject any request containing an "**out-of-date**" signature. A signature is considered to be out-of-date if it was created more than a configured number of minutes outside a window from the current time. By default this is plus or minus 15 minutes. This is to minimize the likelihood that an unauthorized application could capture a properly-signed request and "replay" it to perform an unauthorized action at a later time.

Note that this means that the clocks on the server and the client must be correctly set. The signature time uses UTC time, which is timezone-independent, so the client and server may be in different timezones. The digital signature mathematically guarantees that a request has not been altered in transit. Any attempt to change a request after it has been signed automatically invalidates the signature.

The core assumption is that secured Web service requests must be authorized by the presence of a digital signature, and only an application that is authorized has the knowledge to create a proper digital signature. You can configure a single API key that applies to all Web services running on any web server, including the Application Web Server, web servers for embedded applications, and the Web service debugging server.

You first must configure an access ID and secret ID for the Web Server to generate API keys. Then you must enable API key security for each URL mapping that you want to protect.

As example, we have chosen one of the small remote laboratories build with Lego Mindstorms.

#### Creating API keys at Web Service Server

Web Services Server administrator (see Figure 1) can be accessed by [http://Computer\\_IP:3580](http://Computer_IP:3580). It is developed with Silverlight, thus it is recommended a Windows operating system and the browser Explorer.

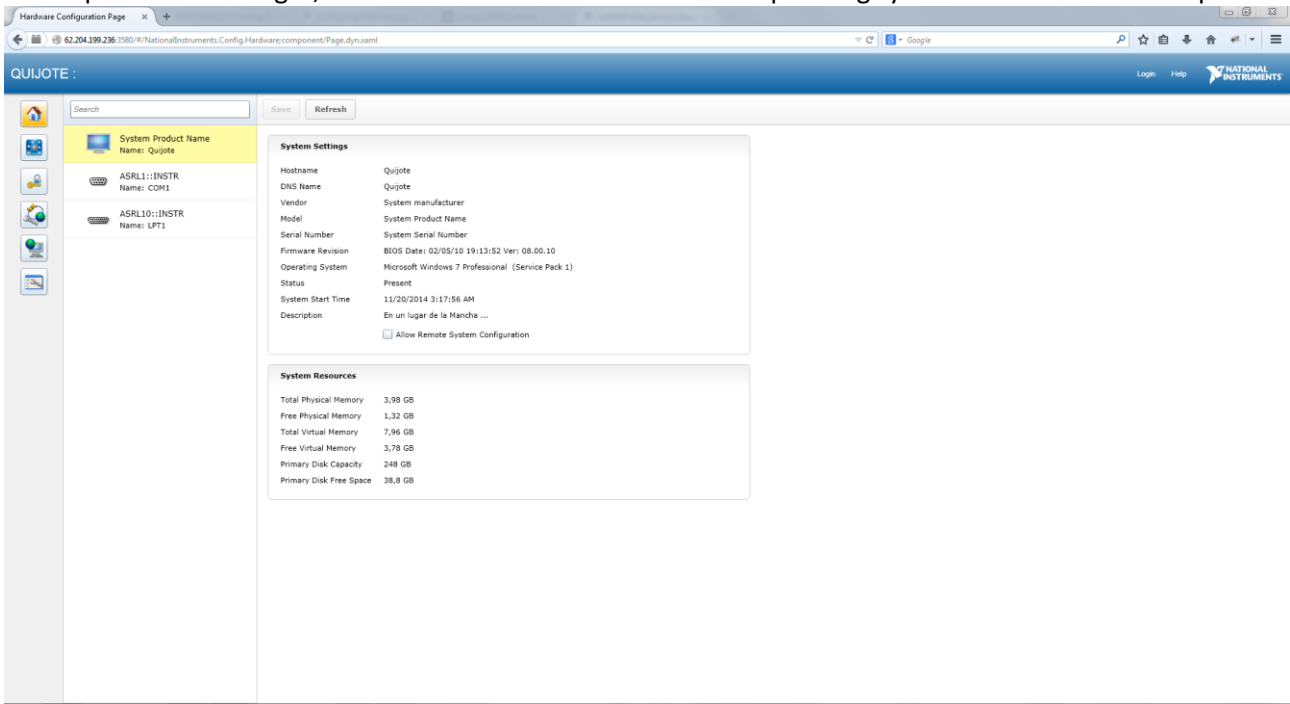


Figure 1. LabView Web Services Server Administrator

On the top and right of the window, there is a **login** link, where we should write our user (Admin as example) and its corresponding password (see Figure 2 ).

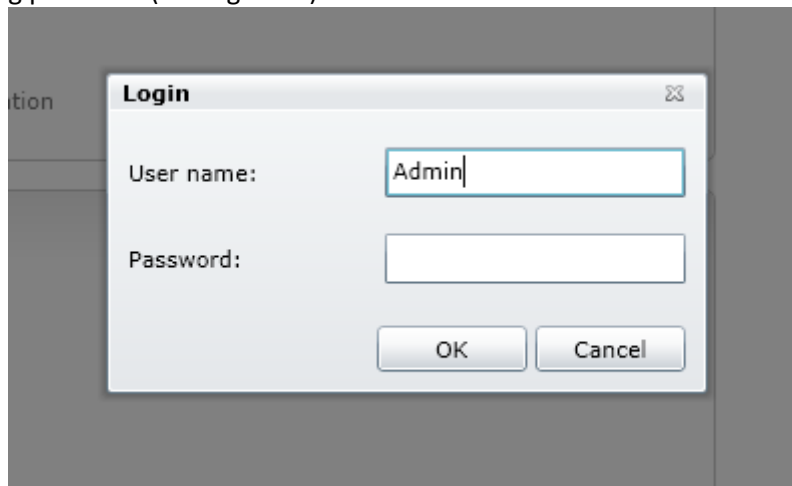

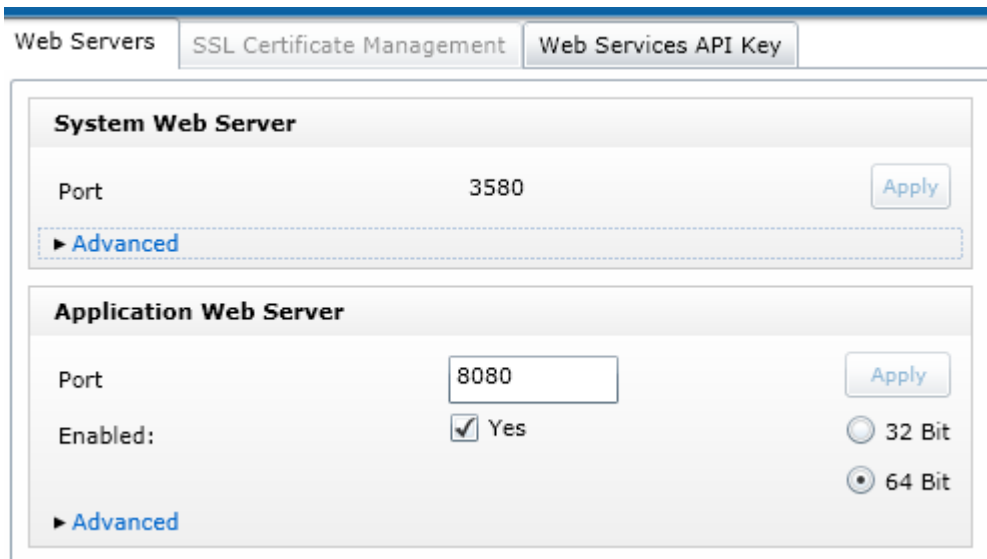




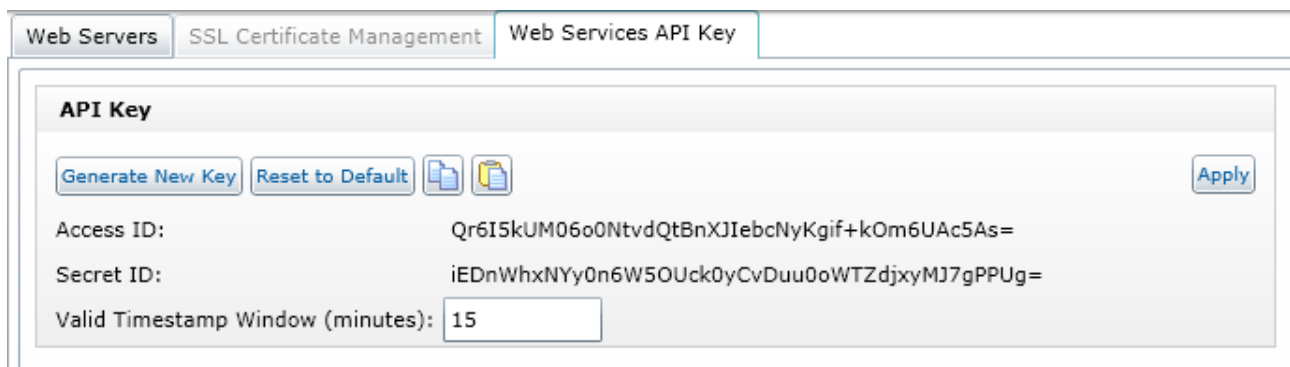
Figure 2. Login dialog

Then, we check the Web Server section clicking in the following icon . From the two available tabs (see Figure 3), we choose “**Web Services API Key**”.



**Figure 3. Web Server screen options**

At the API Key form, there is a button “**Generate New Key**” for generating a new pair of Access ID and Secret ID (see Figure 4). There are also a **copy button**  and **copy to the clipboard button**  so, once the Access ID and the Secret ID are generated, they can be copied into the client application. As it is explained before, each digital signature has valid for a period of time. At the “**Valid Timestamp window**” input box, we can set this parameter. In this case the valid window for a digital signature is 15 minutes. There is also an “**Apply**” button for applying the desired configuration.




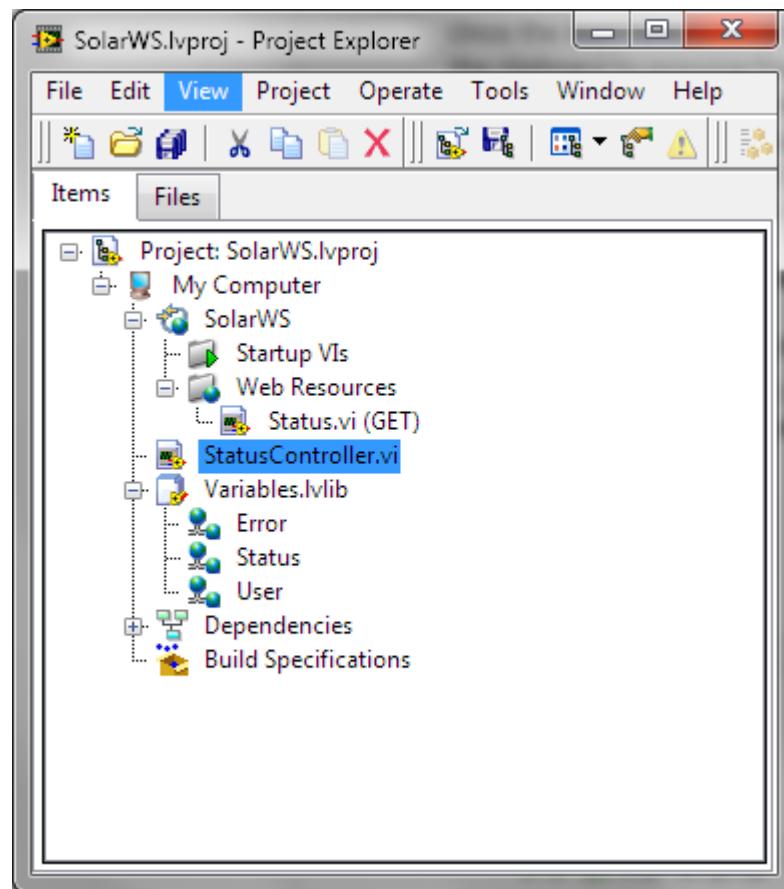
**Figure 4. Web Services API Key configuration options**

Once the keys have been chosen for a particular LabVIEW installation, you can copy the key (both parts) to the clipboard by pressing the **Copy to Clipboard button** in the Security Key dialog, and either paste them into a text document. One common use for copying the Security Key to the clipboard is to be able to paste it into a client application's configuration dialog or its source files, so that those clients may invoke secured Web services on your targets. At the final section of this tutorial there is an explanation about the use of these credentials in a HTML-AJAX client.

### Configuration of Web Services with API keys

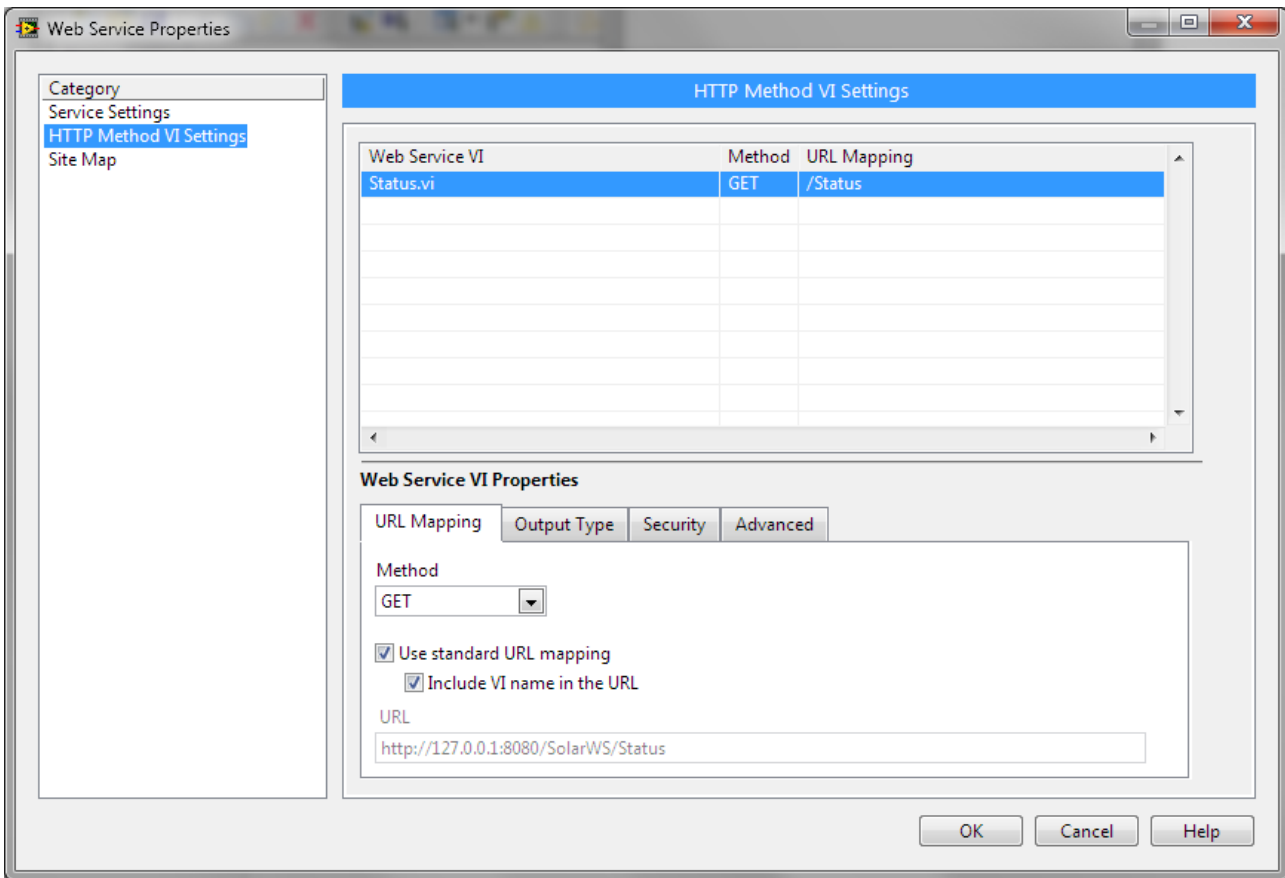
Complete the following steps to enable API key security for an HTTP method VI in a Web service:

- Right-click the Web service project item  in the project tree and select **Properties** to display the **Web Service Properties** dialog box (see Figure 5).



**Figure 5. Project explorer (at Labview 2013 could be different)**

- On the *HTTP Method VI Settings* page (see Figure 6), select a VI from the **Web Service VI** table. In the **Web Service VI Properties** dialog choose “**Security**” tab.



**Figure 6. URL Mappings options at Properties dialog**

- Place a checkmark in the **Require API key** checkbox (see Figure 7).

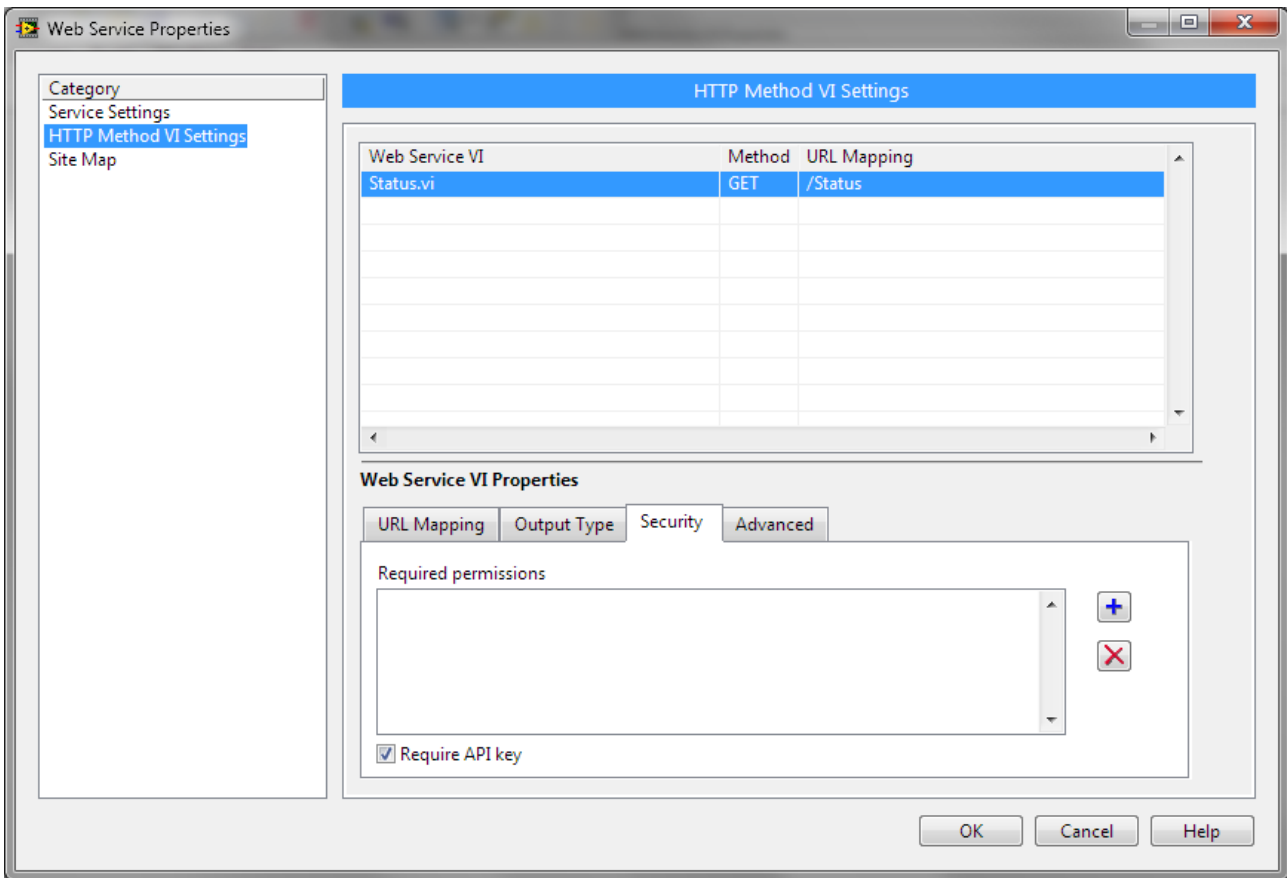


Figure 7. Select "Require API key" option

Once the security parameters are changed, we have to build the project and deploy it as usual (see Figure 8).

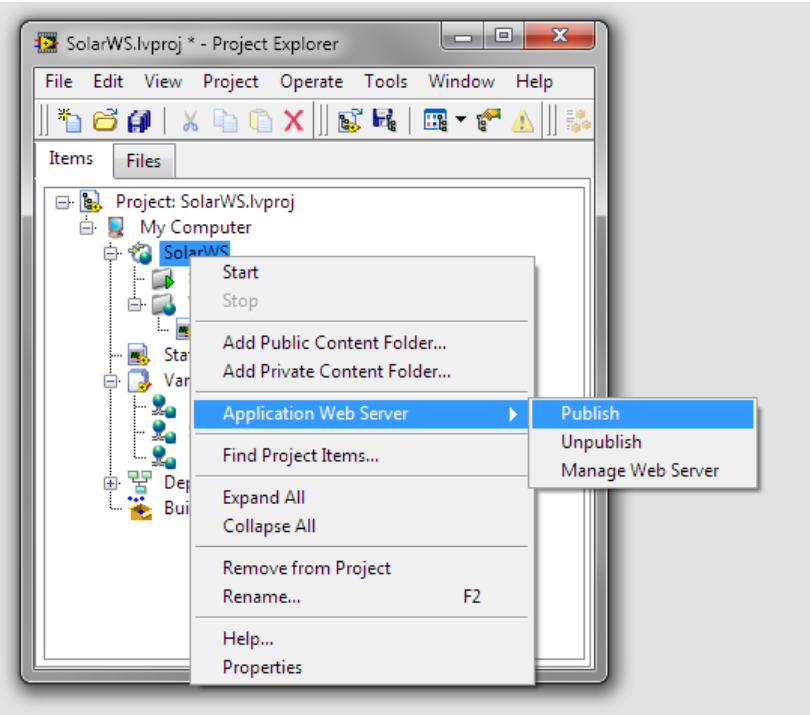
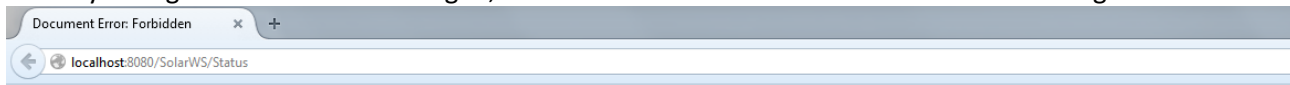


Figure 8. Publishing a Web Service

Due to the need of including a digital signature in each request, if we try directly the status service, which security configuration has been changed, we obtain the 403 access error as it is shown at Figure 9.



**Access Error: 403 -- Forbidden**

**Figure 9. Access error**

## Consuming Web Services attaching API keys

### HTML+AJAX clients

Using AJAX, the main way to consume a Web Service is by means of XMLHttpRequest object. Let's see it with an example:

```
var xmlhttp = new XMLHttpRequest();
var url="http://computer_ip:8080/ProjectID/WSName";

xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
        var data = JSON.parse(xmlhttp.responseText);
        ...
    }
}
```

```
xmlhttp.open("GET", url, true);
xmlhttp.send();
```

The first line is the creation of the XMLHttpRequest object. In the second line, we create a variable called url which represents the URL of the Web Service. In the case of the Lego Laboratory this URL is [http://computer\\_ip:8080/SolarWS/Status](http://computer_ip:8080/SolarWS/Status).

Afterwards we capture the even onreadystatechange for the Web Service. This function is executed when there is a change on the state of the request. The state of a request could be:

- 0: request not initialized
- 1: server connection established
- 2: request received
- 3: processing request
- 4: request finished and response is ready

The variable *readyState* stores each request state. The variable *status* reports about the final status of the request by means of an HTML code (e.g. 404" for "Not Found" or "200" for "OK"). If the status is 200 (everything go well) then the response data is stored at data variable. In this case we assume that the Web Service response contains JSON data. But if the response contains XML data, instead of the var data line, we can do:

```
var data=xmlhttp.responseXML
```

The last two lines are in charge of creating the request to the URL and sending the request to the server.



Our Web Services are secured this way. But, the client applications must change their request adding a digital signature. So this code must change.

Client applications use the **Security key** as part of the digital signing process for Web service requests by combining elements from the message to be sent with both parts of the Security Key via a mathematical algorithm. The resulting signature is for this one instance of this request. It will not work for any other request. Furthermore, it has an expiration date (15 minutes from the time of creation by default, though this can be modified) so that it can't be intercepted and replayed by an unauthorized user. The process of creation a digital signature is where the client application creates a number of individual string elements that will get concatenated together. These elements are:

1. The string for the HTTP method to be used ('GET', 'POST', 'PUT', 'DELETE'), so 'GET' in our example.
2. The URL requested. In our example is the URL of the Web Service Status 'SolarWS/Status'
3. The current time of day (in UTC time using format 'YYYY-MM-DD HH:MM:SSZ')  
'2014-12-01 22:41:02Z'
4. The Access ID portion of the Security Key the default Access ID
5. The hex-encoded MD5 digest of the Secret ID portion of the Security Key  
As an example, given: 'pTe9HRIQuMfJxAG6QCGq7UvoUpJzAzWGKy5SbZ+roSU=' the default Secret ID '4ce83e7d608f70375fd1cda0a6f3ae66' the hex-encoded MD5 digest of the Secret ID string. As it is a static code you can use the following web to compute this code:  
<http://md5sum.org/?plain=pTe9HRIQuMfJxAG6QCGq7UvoUpJzAzWGKy5SbZ%2BroSU%3D> .  
Another good option is the library CriptoJS <https://code.google.com/p/crypto-js/>. This library also allows us to compute the following step.
6. The hex-encoded MD5 digest of the content-body if there is a content-body for the request (Not applicable for this request because there is no content-body for a GET request).

Appended together, these give us a string of:

```
'GET/SolarWS/Status2014-12-01  
22:41:02ZPqVr/ifkAQh+IVrdPIykXIFvg12GhhQFR8H9cUhphgg=4ce83e7d608f70375fd1cda0a6f3ae66'
```

We then calculate the SHA256 digest of that string (which gives us an array of binary data) and encode that array in Base64, to give us:

```
'EB/UfbO60NZrVPkhJ1JrNg8egkK5iwJg9HT6p3zZmbU='
```

To form the final signature, we append together:

1. 'NIWS' (for requests with no content-body, or for requests in which we do not want to include the content-body as part of the signature) or 'NIWS2' (for requests with a content body).
2. A blank space (0x20).
3. The Access ID
4. A colon (0x3a)
5. The Base64-encoded SHA256 digest, as calculated above.

This gives us the signature string:

```
'PqVr/ifkAQh+IVrdPIykXIFvg12GhhQFR8H9cUhphgg=:EB/UfbO60NZrVPkhJ1JrNg8egkK5iwJg9HT6p3zZmbU='
```

This signature string is added to the HTTP request as a header named: **"x-ni-authentication"**.

The timestamp string used to create this signature is also attached to the signature as a header named **"x-ni-date"**.





So a complete signed HTTP request (with '\n' shown to represent the required carriage returns) would look like:

```
GET /SolarWS/Status HTTP/1.1 '\n'  
x-ni-date: 2014-12-01 22:41:02Z '\n'  
x-ni-authentication: NIWS  
PqVr/ifkAQh+lVrdPIykXlFvg12GhhQFR8H9cUhphgg=:EB/UfbO60NZrVPkhJ1JrNg8egkK5iwJg9HT6p3zZmbU=  
\n'  
\n'
```

So the resulting code is the following:

```
<script src="http://crypto-  
js.googlecode.com/svn/tags/3.1.2/build/rollups/sha256.js"></script>  
<script src="http://crypto-  
js.googlecode.com/svn/tags/3.1.2/build/components/enc-base64-min.js"></script>  
<script>  
    var xmlhttp = new XMLHttpRequest();  
    var url="http://localhost:8080/SolarWS/Status";  
    var path='/SolarWS/Status'  
  
    function format(i){  
        if (i< 10) return '0'+i;  
        else return i;  
    }  
    xmlhttp.onreadystatechange = function() {  
        if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {  
            var data = JSON.parse(xmlhttp.responseText);  
            console.log(data);  
        }  
        else console.log(xmlhttp.status);  
    }  
  
    var accessID='PqVr/ifkAQh+lVrdPIykXlFvg12GhhQFR8H9cUhphgg=' ;  
    var secretID='4ce83e7d608f70375fd1cda0a6f3ae66' ;  
    var now=new Date();  
    var timestamp=now.toISOString().replace("T"," ");  
    console.log(timestamp);  
    var token='GET'+path+timestamp+accessID+secretID;  
    var hash =  
CryptoJS.SHA256(token).toString(CryptoJS.enc.Base64);  
    var hsec='NIWS '+accessID+':'+hash;  
    console.log(hsec);  
  
    xmlhttp.open("GET", url, true);  
    xmlhttp.withCredentials = true;  
    xmlhttp.setRequestHeader("x-ni-authentication",hsec);  
    xmlhttp.setRequestHeader("x-ni-date",timestamp);  
    xmlhttp.setRequestHeader('content-type','application/x-www-  
form-urlencoded');  
    xmlhttp.send();
```

</script>

On order that this code works it is needed that the client application is deployed at the same web server that the labview application. If this is not the situation, the requests will be rejected due to CORS. More information about CORS can be found at the following link:

<http://www.html5rocks.com/en/tutorials/cors/#toc-making-a-cors-request>

## Python Client

In addition, this code is equivalent to the previous one in Python language:

```
import requests,hashlib, base64
from datetime import datetime

accessID= 'PqVr/ifkAQh+lVrdPIykXlFvg12GhhQFR8H9cUhphgg='
secretID= 'pTe9HRlQuMfJxAG6QCGq7UvoUpJzAzWGKy5SbZ+roSU='
method='GET'
url= '/SolarWS/Status'

timestamp=datetime.utcnow()
tst=timestamp.strftime("%Y-%m-%d %H:%M:%S")+ 'Z'

secret_digest=hashlib.md5(secretID).hexdigest()

hashed=method+url+tst+accessID+secret_digest

hash_result=hashlib.sha256(hashed).digest()

token='NIWS '+accessID+' :'+base64.b64encode(hash_result)

headers={'x-ni-authentication':token,'x-ni-date':tst,'content-
type':'application/x-www-form-urlencoded'}

r=requests.get('http://localhost:8080/SolarWS/Status',headers=headers)
print r.text
```

## Bibliography

- Labview Tutorial “Securing Web Services”: [http://zone.ni.com/reference/en-XX/help/371361K-01/lvhowto/security\\_ws/](http://zone.ni.com/reference/en-XX/help/371361K-01/lvhowto/security_ws/)
- Labview White Paper about Web Services Security: <http://www.ni.com/white-paper/7749/en/>