**Basic HTTP Authentication**

## **Basic Authentication**

from requests.auth import HTTPBasicAuth  
requests.get('https://api.github.com/user', auth=HTTPBasicAuth('user', 'pass'))  
<Response [200]>

In fact, HTTP Basic Auth is so common that **Requests provides a handy shorthand** for using it:

requests.get('https://api.github.com/user', auth=('user', 'pass'))  
<Response [200]>

Providing the credentials in a tuple like this is exactly the same as the HTTPBasicAuth example above.

This document defines the "Basic" Hypertext Transfer Protocol (HTTP)

authentication scheme, which transmits credentials as **user-id/**

**password pairs**, **encoded using Base64**

This scheme is not considered to be a secure method of user

authentication unless used in conjunction with some external secure

system such as **TLS** (**Transport Layer Security**, [[RFC5246](https://tools.ietf.org/html/rfc5246)]), as the

user-id and password are passed over the network as cleartext.

To receive authorization, the client

1. obtains the user-id and password from the user,

2. constructs the user-pass by concatenating the user-id, a single

colon (":") character, and the password,

3. encodes the user-pass into an octet sequence (see below for a

discussion of character encoding schemes),

4. and obtains the basic-credentials by encoding this octet sequence

using Base64 ([[RFC4648], Section 4](https://tools.ietf.org/html/rfc4648#section-4)) into a sequence of US-ASCII

characters ([[RFC0020](https://tools.ietf.org/html/rfc0020)]).

**rest API sends basic auth credentials through header Authorization**

If the user agent wishes to send the user-id "test" and password

"123", it would use the **header field**:

**Authorization: Basic** **dGVzdDoxMjPCow==**

The **user's name is "test"**,

and the **password** is the string **"123"**

**followed by the Unicode character U+00A3 (POUND SIGN)**.

* Using the character **encoding scheme UTF-8**, the user-pass becomes:

**'t' 'e' 's' 't' ':' '1' '2' '3' #**

74 65 73 74 3A 31 32 33 C2 A3

* **Encoding this octet sequence in Base64** ([[RFC4648], Section 4](https://tools.ietf.org/html/rfc4648#section-4)) yields:

**dGVzdDoxMjPCow==**

Thus, **the Authorization header** field would be:

**Authorization: Basic** **dGVzdDoxMjPCow==**

Or, for proxy authentication:

Proxy-Authorization: Basic dGVzdDoxMjPCow==

The Basic authentication scheme is not a secure method of user

authentication, nor does it in any way protect the entity, which is

**transmitted in cleartext** across the physical network used as the

carrier.

**The most serious flaw of Basic authentication is that it results in**

**the cleartext transmission** of the user's password over the physical

network. Many other authentication schemes address this problem.

Because Basic authentication involves the cleartext transmission of

passwords, it SHOULD NOT be used (without enhancements such as HTTPS

[[RFC2818](https://tools.ietf.org/html/rfc2818)]) to protect sensitive or valuable information.

In HTTP Basic Auth, the application expects a header that contains a username and a password.

If it doesn't receive it, it returns an HTTP 401 "Unauthorized" error.

And returns a header WWW-Authenticate with a value of Basic, and an optional realm parameter.

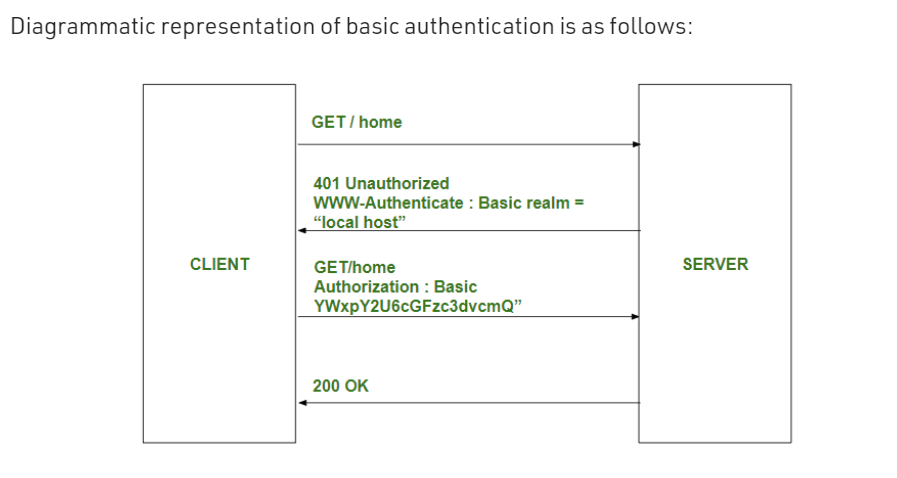
That tells the browser to show the integrated prompt for a username and password.

Then, when you type that username and password, the browser sends them in the header automatically.

As per section 4.1 of RFC-7235, when an HTTP server returns a 401 response, it must also return a WWW-Authenticate header :

1. A server generating a 401 (Unauthorized) response MUST send a
2. WWW-Authenticate **header** field containing at least one challenge.

The **HTTP WWW-Authenticate** response header defines the authentication method that ought to be wont to gain access to a resource. As discussed earlier, the **WWW-Authenticate header** is sent along with a 401 Unauthorized response



401 Response

You can also define the 401 “Unauthorized” response returned for requests with missing or incorrect credentials. This response includes the WWW-Authenticate header, which you may want to mention.

**Basic Auth with cookie using requests**

s = requests.Session()  
data = {"login":"my\_login", "password":"my\_password"}  
url = "http://example.net/login"  
r = s.post(url, data=data)  
  
# After the first log in , get cookies for auth in future  
s.cookies

import requests  
import json  
url = "http://172.125.169.21/api/user/login"  
payload = {"name": "test5", "password": "Hello123"}  
headers = {'Content-Type': "application/json"}  
response = requests.request("POST", url, json=payload, headers=headers)  
# the json parameter should handle encoding for you  
print(response.text)  
print(response.headers)  
  
print(json.dumps(response.cookies, separators=(",",":"), indent = 4))  
  
cookies = response.cookies  
auth\_check\_url = "172.125.169.21/api/user/checkLogin"  
response = requests.get(auth\_check\_url, cookies=cookies)  
print(response.text)

* SSL error in python request

File "C:\Python27\lib\site-packages\requests\adapters.py", line 497, in send

    raise SSLError(e, request=request)

requests.exceptions.SSLError: [SSL: CERTIFICATE\_VERIFY\_FAILED] certificate verify failed (\_ssl.c:590)

import requests  
  
loginurl = 'https://jira01.devtools.mycompany.com/rest/auth/1/session'  
  
resp = requests.post(loginurl, data={'username': 'myusername','password': 'mypassword'})  
  
print(resp.status\_code, resp.reason)  
  
File "C:\Python27\lib\site-packages\requests\adapters.py", line 497, in send  
 raise SSLError(e, request=request)   
requests.exceptions.SSLError: [SSL: CERTIFICATE\_VERIFY\_FAILED] certificate verify failed (\_ssl.c:590)

The error says that the python client does not trust the SSL certificate of the server. In your browser it works, because usually browsers have a broader list of trusted **CA** (**certificate authorities** )or due to a company policy, the CA certificate has already been preinstalled on your PC.

What you need to do is

* **export the CA certificate to a pem format** (you can do this in the browser - look in Google) and use the "verify" parameter when doing the POST request - see [documentation here](http://docs.python-requests.org/en/master/user/advanced/) (section "SSL Cert Verification").
* **Disable certificate verification** altogether by passing **verify=False** to the requests.post method.

I did look around on my computer and found that there are several **.pem files** like **'cacert.pem'** here **C:\Python27\Lib\site-packages\certifi**.  Certifi documentation says python will automatically use this if certify is available which it is. The cacert.pem did not contain any reference to 'jira' however so I'm assuming I will be adding a CA certificate(s) for JIRA here?

import requests  
  
loginurl = 'https://jira01.devtools.mycompany.com/rest/auth/1/session'  
  
filepath = 'C:\Path\To\My\JIRA.pem'  
  
loginArgs = {'username': 'myusername', 'password': 'mypassword'}  
  
resp = requests.post(loginurl, json=loginArgs, verify=filepath)  
  
print(resp.status\_code, resp.reason)

>> (200, ‘OK’)

**Ex: cookie authentication with Jira**

[Cookie-based auth for REST APIs (atlassian.com)](https://developer.atlassian.com/cloud/jira/platform/jira-rest-api-cookie-based-authentication/)

Jira's REST API is protected by the same restrictions which are provided via Jira's standard web interface. This means that if you do not log in, you are accessing Jira anonymously. Furthermore, if you log in and do not have permission to view something in Jira, you will not be able to view it using the Jira REST API either.

In most cases, the first step in using the Jira REST API is to authenticate a user account with your Jira site. Any authentication that works against Jira will work against the REST API. In this tutorial, we will use cookie-based (session) authentication.

This is how cookie-based authentication works in Jira at a high level:

1. The client creates a new session for the user, via the Jira REST API .
2. Jira returns a session object, which has information about the session including the session cookie. The client stores this session object.
3. The client can now set the cookie in the header for all subsequent requests to the Jira REST API.

## **Step 1. Create a new session using the Jira REST API**

1. We need to get a session cookie from Jira, so the first thing we need to do is create a new session using the sessionresource in the Jira REST API. Tip: You can also use the *session* resource to get information about the currently authenticated user in the current session (GET), or log the current user out of Jira (DELETE).
2. To do this, just POST the desired user credentials (as JSON) to the session resource:

|  |  |
| --- | --- |
| Example resource | http://jira.example.com:8090/jira/rest/auth/1/session |
| Example credentials | { "username": "myuser", "password": "mypassword" } |

1. This will create a new session and return the requested session information, which will look similar to the following:

{

"session":

{

"name":"example.cookie.name",

"value":"6E3487971234567896704A9EB4AE501F"

}

}

1. More importantly, you will get the session cookie (in the header of the response) from the server, which you can use in subsequent requests. You can see an example of this below. You'll notice that the cookie name and value are the same as the cookie name and value in the session response above

Set-Cookie: example.cookie.name=6E3487971234567896704A9EB4AE501F; Path=/; HttpOnly

## **Step 2. Use the session cookie in a request**

Now that you've created a session, it's just a matter of setting the cookie in all subsequent requests to the server.

1. Store the session object on the client. The way that you do this will depend on how your client is implemented.
2. When you want to make a request, take cookie name and value from the session and use them to set the 'cookie' field in the header of your request. You can see an example of this below:

headers: {cookie: example.cookie.name=6E3487971234567896704A9EB4AE501F}

That's it! Now, when you submit the request, the session cookie will be used to authenticate you to the Jira server until the cookie expires.

**Cookie expiration**

One disadvantage of using cookies compared to basic authorization is that they expire. You have probably noticed this when accessing Jira through a web browser. Every once in a while, especially if you have not used Jira in a while, you need to log in again because your cookie has expired. The same phenomenon occurs when using REST. If you are writing a script or code which involves REST API calls and:

* Only runs for a few minutes, then you should not have to worry about cookies expiring.
* Runs for a longer period of time due to more complex integration activities, then expiring cookies may cause problems.

If you use REST with a cookie that has expired, you will receive a 401 error response from Jira. The response body will contain a message telling you that your cookie is invalid. At that point, you will need to re-authenticate to the session resource on the "auth" API.