The HTTP Module

Introduction

This blurb is about using QDL's HTTP module. This module allows you to do basic operations to a website using the HTTP protocol. It is intended not to be a high-level tool, but a simple library that lets you write one.

Loading the module

This is a Java module and is included in the standard distribution, but is not loaded, so to use it load it by issuing

```
http := j_load('http')
```

Supported functions quick reference

Name	Description	Comment
close()	close the connection	All requests will fail until open() is called
configure	Query/set the configuration	
delete()	DELETE	Use current host
delete(parameters.)	DELETE	Use current host, add parameters
delete(uri_path, parameters.)	DELETE	Append uri_path to host, add parameters
echo_response	Echo all responses to standard out.	May be *enormously* chatty
echo_request	Echo all requests to standard out.	ditto
get()	GET	Use current host
get(uri_path)	GET	Add add uri path to host, no parameters.
get(parameters.)	GET	Use host, parameters. is a stem of query parameters to be encoded and added to the path
get(uri_path, parameters.)	GET	Append path_uri to host, use parameters
headers()	list the current default headers	
headers(arg.)	set the default headers	

host()	get the current host	
host(host_name)	set the current host	Returns previous host
is_open()	is the connection open?	
is_json(response.)	Is the response content of type JSON?	This and is_text accept either the response or just the headers.
is_text(response.)	Is the response content of type text?	This does not include JSON since there is a separate method for that.
open()	open a new connection	
open(insecure)	open a new, insecure connection	insecure is a boolean, which if **true** will turn off security for SSL. Default is **false**.
post(arg | arg.)	POST	Payload may be a string or a stem.
post(uri_path, arg | arg.)	POST	Append uri_path to host, send payload
put(arg | arg.)	PUT	Payload may be a string or stem
put(uri_path, arg | arg.)	PUT	Append uri_path to host, send payload

Get returns a stem with entries for status, content and any returned headers.

Nota Bene: You may set parameter values in **`get, put`** and **`post`** as lists or sets too. These will be sent as a list of parameters to the server in the form **`key[]=v0&key[]=v1&...`** This is easy to do, but be sure the server supports multiple values. Nested structures and general stems will be rejected with an error since there is no standard way to encode them. If you need an alternate format for lists, see **`configure`** and the section below on A*ggregates as parameters.*

Paths, parameters and requests

If the host is set, then you may add an optional uri path to it. The path is appended to the host on a per request basis. This lets you set a base address for the service then specialize calls. Additionally, you may send a stem which will be converted to parameters. Let us say we set up our HTTP connection with

```
http#host('https://some_service.com');
```

Then we can issue the following

```
http#get('logon', {'id' : 'Knútr.Forkbeard', 'password' : '123 foo'})
```

The resulting request constructed would be

https://some_service.com/logon?id=Kn%C3%BAtr.Forkbeard&password=123%20foo

Aggregates as parameters

Stems should be simple, in that entries can be scalars, lists or sets, but not stems. If the entries are lists or sets they will be encoded as lists. There is no standard for encoding lists, so this module lets you choose the various popular options via the **`configure`** call. The default is to send them as array elements, so if you have a parameter of

```
   {'my_param' : ['a','b','c']}
```

Then this is encoded as

```
my_param[]=a&my_param[]=b&my_param[]=c
```

Many servers accept this and it corresponds to having the parameter encoding set to 'array' which is the default. You may set it to 'parameter' which then simply repeats that parameters (so no []'s):

```
my_param=a&my_param=b&my_param=c
```

Finally, if the encoding type is 'value', then each value is comma separated:

```
my_param=a,b,c
```

You may set the list separator to be something other than the comma if your sever really needs it.

The response. A sample session.

Assuming you have loaded the above, open up a connection and get

```
    http:= j_load('http')
    http#host('https://didact-patto.dev.umccr.org/api/visa') ;
    http#open();
true
    z. := http#get({'sub':'https://nagim.dev/p/wjaha-ppqrg-10000'});
    z.
{
 headers: {
  Connection:keep-alive,
  etag:W/"e6-suhkGbMm3fkbNhOR6bOIwIgkh8A",
  Apigw-Requestid:Gv9mdhv4SwMEMHw=,
  Content-Length:230,
  Date:Tue, 05 Oct 2021 19:37:04 GMT,
  Content-Type:application/json; charset=utf-8,
  X-Powered-By:Express
 },
 content: [
    {

s:XnKFkl4RTXtB2DDOf5f4yLtfcTaCGyqMxIV8Q42zX_XR1p9Cnxeqg2KI_4UCzcJZ2XGv_hlqVGOW5_3FE
9ZHCQ,
   v:c:8XZF4195109CIIERC35P577HAM et:1633549022 iu:https://nagim.dev/p/wjaha-ppqrg-
10000 iv:2f69e2650aed4f0e,
   k:rfc8032-7.1-test1
```

```
  }
  ],
 status: {
  code:200,
  message:OK
 }
}
   http#is_json(z.)
true
```

So we see the various components of the response, z.:

- `headers.` - A stem of the headers, where the key is the name of the header and the value is its value (as a string, so `Content-Length` is not a number).
- `content.` - The exact content. Here, it is an array with a single element and note that `headers.Content-Type` contains `application/json` and hence was in JSON format. If the content type is anything else, it will be returned as a stem of lines. The **`is_json`** function tell you if the content type was JSON. In this example, it was a JSON blob with 3 entries.
- `status.` - The http status, which includes the status code and the message from the server. Anything other than something in the 200 range is an error.

Note that there are other options for `content.` but it will always be an array. For instance, from other servers it may be the lines in the body of the response if the `Content-Type` is `form_encoding`. In that case, you will have to loop through the lines and process each of them in turn. Here

```
   body. := response.'content'.0
```

gets the content as a stem from the server.

A GET example

In this case, we have several pages to be fetched

```
   http := j_load('http');
   http#open();
true
   http#host('https://cilogon.org/.well-known/openid-configuration');

   pages. := ['','fermilab','ligo'];
   while[vi ∈ pages.][say(http#get(vi).'content'.'issuer');];
https://cilogon.org
https://cilogon.org/fermilab
https://cilogon.org/ligo
   http#close();
true
```

In this case, it goes to the Cilogon server and grabs some well known pages in turn, constructing the paths for various VIs (virtual issuers), showing the issuer entry for each page. This is quite basic.

Normally you might enclose the call in a try – catch block, check return codes, etc. but this is to show how **`get()`** works, naught else.

Echoing requests and responses

You may enable echoing raw HTTP requests and responses to standard out using the **`echo_request`** and **`echo_http_response`** resp. This dumps what goes over the wire to the console and is intended to be a very low-level debugging tool. Since this works for every call, the sheer volume may be quite large. Generally you toggle these on by passing **true** and off by passing **false**. You may query the current state by issuing the calls with no arguments, or setting it with a boolean:

```
   http#echo_http_request(true)
false
```

meaning in this case no echoing was on (previous value returned) and is now enabled.

And a typical output for echoing everything is

```
----- Echo request -----
GET https://registry-test.hassi-aai.ardc.edu.au/registry/api/co/2/core/v1/people?
identifier=http%3A%2F%2Fcilogon.org%2FserverT%2Fusers%2F17776568 HTTP/1.1
----- End echo request -----


----- Echo response -----
Content Type: application/json; charset=UTF-8
Status: 404
Raw Response:
{"error":"CoreApiPerson \"\" Not Found"}
----- End echo response -----
```

Function Reference

close

Description

Close the connection. If the connection has not been opened, this has no effect.
See also: **open**

Usage

close()

Arguments

None.

Output

Boolean **true** if closing the connection succeeded. If called, all requests will fail until **open** is called again.

Examples

```
   http#close()
true
```

configure

Description

Set the configuration for the HTTP module. This allows you to turn on echoing requests and responses (to standard out), as well as setting the format for lists.

Usage

configure({arg.})

Arguments

(none) – query current settings, nothing is changed

arg. – stem with new values

Output

The current values if no argument, or the previous values if there was a change. Note that if there is an error updating the values, then no values are altered. The structure of the stem is

```
key0  key1      value
---------------------------
echo  request   true|false
      response  true|false
                true|false
list  encode    'array' | 'parameter' | 'value'
list  separator ',' | other
```

Note that if the echo entry is a boolean, then both requests and responses will be echoed to the console. You may also set them individually with the **`echo_request`** and **`echo_response`** calls.

Examples

```
   http#configure()
{echo:false, list:{encode:array, separator:,}}
```

This is the default. Echo is off, array encoding for multiple parameters is used.

credentials

Description

Create the correct credential for the basic auth header. You then can use this to set a header.

Usage

```
dcredentials(username, password)
```

Arguments

username – the name of the user

password – their password

Output

The standard is to return

```
encode_b64(url_escape(username) + ':' + url_escape(password))
```

Examples

Just the call itself:

```
   http#credentials('bob@bgsu.edu','mairzy doats')
Ym9iJTQwYmdzdS5lZHU6bWFpcnp5K2RvYXRz
```

And for reference

```
   decode(Ym9iJTQwYmdzdS5lZHU6bWFpcnp5K2RvYXRz)
bob%40bgsu.edu:mairzy+doats
```

A typical use of this for setting headers

```
   my_credential := http#credentials(username, password);
   my_headers.'Authorization' := 'Basic' + ' ' + my_credential;
   my_headers.'Content-Type'  := 'application/json';

   http#headers(my_headers.);
```

delete

Description

Issue an HTTP DELETE to the server

Usage

`delete({uri_path}, {parameters.})`

Arguments

(none) – issue the DELETE

`uri_path` – optional

`parameters.` – use the stem to add parameters to the DELETE.

Output

Boolean true if the operation succeeded.

Examples

download

Description

Download a file or zipped archive, decompressing it.

Usage

`download(url, target_file)`

`download(url, target_directory, is_archive)`

Arguments

`url` – the web address of the file

`target` – the target of the download.

`is_archive` – If the source is a zipped archive, specifically a jar

The dyadic form downloads to a file, the triadic form to a directory. If **`is_archive`** is true in that case, the entire archive is unzipped to the target directory.

Output

The number of bytes processed or a -1 if the operation did not work (assuming no other error is raised). For instance, the source is an empty file, or you may get 0 bytes processed if you specify that the file is a zipped archive, but it is not (so the system finds zero files to process).

Note: You do not need to set a host or open a connection for this call. It is self-contained. Also, gzipped archives are not supported directly, only jars, so if you need to get a gzipped archive, download it locally, then use tools to unpack it.

Examples

```
http#download('https://github.com/ncsa/oa4mp/v6.2.1/jwt-scripts.tar',
              '/tmp/http_test',
              false);
32548
```

This will download jwt-scripts.tar to the given directory. In this case, 32548 byte were processed.

```
http#download('https://github.com/ncsa/oa4mp/v6.2.1/jwt-scripts.tar',
              '/tmp/http_test',
              true);
0
```

Since the tar file is *not* an archive, nothing was downloaded.

```
http#download('https://github.com/ncsa/oa4mp/v6.2.1/jwt-scripts.tar',
              '/tmp/http_test/scripts.tar');
32548
```

Again 32548 bytes are processed and the remote file jwt-scripts.tar is now locally at /tmp/http_test/scripts.tar.

echo_request

Description

Echo all HTTP requests to the console before each request.
See also: **`echo_response`**

Usage

`echo_request({arg})`

Arguments

(none) – query current state

arg - **`true | false`** that sets the state.

Output

Returns previous value. Be warned that this may be *very* chatty. It is intended as a lower-level debugging tool.

Examples

echo_response

Description

Echo each raw response from the server before processing it.
See also: **`echo_request`**

Usage

`echo_response({arg})`

Arguments

(none) – query current state.

Arg = **`true | false`** that sets the state

Output

If no argument, the current state, otherwise the previous state. This may be *very* chatty. It is intended as a lower-level debugging tool.

Examples

get

Description

Execute an HTTP GET .

Usage

```
get({url_path},{parameters.})
```

Arguments

(none) – issue request with just the host

url_path – append the path to the current host

parameters. – a stem of key/values that are converted to query parameters in the request.

Output

A response. See the section above on the format of the response.

Examples

headers

Description

Query or set the headers. These will be used for all subsequent requests.

Usage

```
headers({headers.})
```

Arguments

(none) – query current headers.

`headers.` – Set the headers.

Output

If no argument, the current headers, if you set them, the previous headers are returned.

Examples

```
http#headers({'x-rems-api-key': ini.'x_rems_api_key',
              'accept': 'application/json',
              'x-rems-user-id': ini.'x_rems_user_id'});
[]
```

This sets the header and the response shows that there were none previously set.

host

Description

Query/set the host for all requests

Usage

```
host({address})
```

Arguments

(none) – query the host

address – the address of the host. All requests will be sent to this address.

Output

If no argument, the current host (if any). If the address is set, the previous host address is returned.

Examples

```
 http#host(config_params.'comanage_registry_host_url');
```

Note that the response was an empty string, hence no host had been previously set.

is_json

Description

Check if the content of a response is JSON.
See also: is_text

Usage

```
is_json(response.)
```

Arguments

response. – the response.

Output

A boolean **true** if the response content type is JSON and false otherwise

Examples

```
response. := http#get();
body. := is_json(response.) ? from_json(response.content) : null;
```

In this case, get a response and if it is json, turn it into a stem, otherwise leaving the as a null;

is_open

Description

Is there a connection opened to the current host?
See also: **`open`**, **`close`**

Usage

`is_open()`

Arguments

(none)

Output

A boolean true if there is a connection open, false otherwise.

Examples

is_text

Description

Check if a response is

Usage

`is_test(response.)`

Arguments

response. -- The response from a request or just the headers

Output

This returns a boolean **true** if the content is text (includes values like text, html, java, javascript etc.) and false otherwise.

Examples

```
   http#is_text(response.'headers');
true
```

open

Description

Open the connection to a server.
See also: **`close`**, **`is_open`**

Usage

`open({unverified})`

Arguments

(none) – opens a connection, using SSL if needed.

`unverified` – a boolean that if true will turn of SSL verification.

Output

A boolean **true** if the connection was opened successfully. Note that turning off SSL does not increase speed or performance and removes essential security. However, in testing environments, *e.g.* against a test server with a self-signed cert, it is needed.

Examples

post

Description

Issue an HTTP POST to the server.

Usage

`post(body | parameters.)`

`post(url_path, body | parameters.)`

Arguments

`body` – a string that will be ebcoded as the body of the post.

`parameters.` – a stem that will be converted to parameters and encoded as the body of the post.

`url_path` – a path appended to the host

Output

The response from the server.

Examples

```
  http#post('We are trying to contact you about your car\'s extended warranty');
```

Posts with the single sentence as the body

```
  http#host('https://some_service.com');
  http#post('logon', {'id' : 'Knútr.Forkbeard', 'password' : '123 foo'})
```

which would send the request to https://some_service.com/logon but with the body of the post as

id=Kn%C3%Batr.Forkbeard&password=123%20foo

put

Description

Issue an HTTP PUT to the server
See also: **`get`**, **`post`**

Usage

`put(body | parameters.)`

`put(url_path, body | parameters.)`

Arguments

`body` – a string that will be ebcoded as the body of the post.

`parameters.` – a stem that will be converted to parameters and encoded as the body of the post.

`url_path` – a path appended to the host

Output

The response from the server.

Examples

Same as POST, except for the call. The difference between POST and PUT relates to whether the state on the server changes, but the semantics of the call are the same. Normally POST is used to create a new resource, PUT is used to update/replace an existing one.

Description

Usage

d

Arguments

Output

Examples