Chapter 4b: Cloud Protocol: HTTP

At this end of this chapter, you will understand the HTTP protocol, including:

- The architecture and use model of the ModusToolbox™ for Connectivity *http_client* and *http_server* libraries.
- How to use CURL to test HTTP(S) servers.
- How to use HTTP(S) to read & write data to the Cloud using RESTful APIs.
- How to create an HTTPS connection using TLS.
- How to test your HTTP client using HTTPBIN.

Table of contents

Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	CY_ISR_PROTO(MyISR); make build
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

4b.1 Introduction

When HTTP came on the scene in the early 90's, it was principally used to send static HTML pages. Over time, dynamic HTTP came into common use (reading and writing databases and creating HTML on the fly). Many companies built big teams of people to develop and deploy HTTP based applications internally to their employees and externally to their customers.

As IoT emerged, it was only natural and financially advantageous for companies to extend their existing infrastructure to enable IoT devices to communicate with the existing Web services. Although HTTP has issues which make it less than "perfect" for IoT, it is still the most important standard because of the huge investment that has been made in the existing Internet infrastructure.

There are essentially two versions of HTTP: 1.1 and 2.0. Although conceptually similar, they are materially different in their implementation.

HTTP 1.1 was released in 1997 and as of 2024 it still serves about 40% of web traffic. HTTP 2.0, which was released in 2015, brings many performance benefits but has seen slow uptake in the market.

In this chapter, we will focus on HTTP 1.1 since that's the version that is typically used on IoT devices that use the HTTP protocol.

4b.2 HTTP 1.1 protocol

HTTP 1.1 is an application layer, single transaction, stateless, plain-text, client-server protocol. This means that a client (e.g. your device) opens a connection to a TCP server (in the Cloud), sends an ASCII text request, then the server responds, and the connection is closed. There is no memory in the protocol itself but there might be in an application e.g. Cookies.
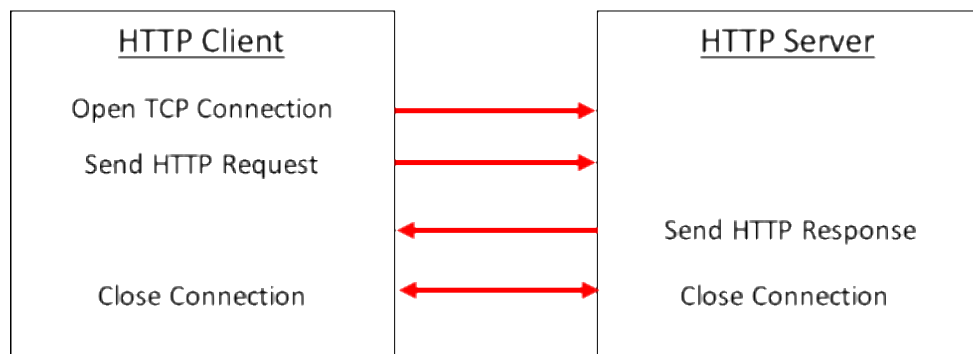
HTTP requests and responses are made up of one mandatory start line, an optional group of HTTP headers (same format for client and server) and one optional content body (same format for client and server).

4b.2.1   Client request message format

An HTTP transaction starts with the client opening a TCP socket to the server (or a TLS TCP socket to the server). The client then sends up to four things:

- Client Request Start Line (4b.2.2)
- Headers (4b.2.6) (one or more strings of the form of "headername:headervalue\r\n")
- A "\r\n" line after the last header
- Optional: Content Body (4b.2.7) (one payload with as many bytes as required e.g. a file or an html page or a JSON document)

4b.2.2   Client request > start line

The client request start line has five elements:

- The HTTP Method (4b.2.3)
- The requested Resource path (4b.2.4)
- Optional: Options (4b.2.5) (a '?' followed by a list of optional arguments separated by '&')
- The version of HTTP (for this chapter it will always be "HTTP/1.1"
- A "\r\n"

An example legal client request start line is:

```
GET /ask HTTP/1.1
```

In the above example, *GET* is the HTTP Method, */ask* is the Resource, and *HTTP/1.1* is the version.

4b.2.3   Client request > start line > HTTP methods

There are 9 HTTP methods which are sometimes called "verbs" because they request a simple action from the server to act upon a Resource. The verbs fit into two categories (Safe/Unsafe, Idempotent/non-Idempotent)

- **Safe** – the method doesn't change anything on the server and can be run without fear of side effects. Any method that changes anything on the server is therefore Unsafe.
- **Idempotent** – no matter how many times you run the method, the state on the server state remains the same. For example, if you "PUT" a document to the server, it will only have one instance on the server no matter how many times you run it. As another example, a DELETE will have the same effect no matter how many times you run it. A non-idempotent method changes the state of the server every time you run it. For example, a POST might insert data in the database every time it is run.

4b.2.3.1   GET (safe, idempotent)

For the GET method (https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET), the server will reply with an HTTP response with the Content Body of the requested Resource (i.e. the file). The server response will include Headers that will tell the client how long the Content Body is "Content-length" and what is the MIME-Type (4b.2.7) of the Content Body "Content-type".

4b.2.3.2   HEAD (safe, idempotent)

The Head method (https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD) performs the same operation as "GET" except it only replies with the Headers and does not return the Content Body.

4b.2.3.3 PUT (unsafe, idempotent)

For the PUT method (https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PUT), the client asks the server to replace the Resource with the Content attached to the message. The server knows the length of the Content based on the Header "Content-length" and the MIME Type based on the Header "Content-type".

4b.2.3.4 PATCH (unsafe, idempotent)

With this method (https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PATCH) the client is requesting a partial PUT e.g. if the Resource is a document that contains a name and an age, then the client could PATCH just a new age by having content with the updated age by sending a JSON document with "{age:49}".

4b.2.3.5 POST (unsafe, non-idempotent)

For the POST method (https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST), the client asks the server to update the Resource based on the Content attached to the message. An example of this method is sending a temperature to the server which will be saved into the database. The server knows the length of the Content based on the Header "Content-length" and the MIME Type based on the Header "Content-type".

4b.2.3.6 DELETE (unsafe, idempotent)

The DELETE method https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/DELETE asks the server to remove the Resource.

4b.2.3.7 OPTIONS (safe, idempotent)

This OPTIONS method https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS asks the server to respond with an HTTP message that has a "Options" header that enumerates the list of legal HTTP Methods for that server.

4b.2.3.8 TRACE (safe, idempotent)

This TRACE method (https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/TRACE) is an infrequently implemented debugging Method that should cause the server to reply with the client message (echo'd back).

4b.2.3.9 CONNECT

The CONNECT method (https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/CONNECT) requests the server to open a tunneling TCP connection. This method is probably never used in an IoT application.

You should be aware that the idempotence and safety of these methods is established by convention. There is no technical reason why a "GET" couldn't delete the resource or a "PUT" couldn't return the resource, but you should never implement a web server like that.

4b.2.4 Client request > start line > resources

When you look at an http web address (sometime known as a Universal Record locator (URL)) you typically see:

- http://server.com/path

or

- http://server.com/path?option=28

These URLs are of the form of:

- "http:" specifies the protocol.
- "server.com" is the DNS name of the HTTP server
- "/path" the location of the resource on the HTTP server.
- "?option=28" is an option that is sent to the server (see next section)

In generic terms, a URL is "protocol://serverName/path?option".

In HTTP 1.1 the 2nd and 3rd elements of the client request line are the resource & options which are the same as the path and options from a URL. For example, you might see an HTTP request that looks like this:

```
GET /resource HTTP/1.1
```

Which is a request to the server to please send the document located at "/resource" as an HTTP response.

4b.2.5 Client request > start line > options

Options are appended to the resource location by placing a "?" at the end of the resource path. You can then specify options by adding "option=value" or just "option". You can specify multiple options by separating them with "&". These options are sometimes used to send commands or other information to the server e.g. "user=arh&password=secret".

An example request with an option to format the response as "simple" (what "simple" means is part of the application semantics) might look like this:

```
GET /resource?format=simple HTTP/1.1
```

4b.2.6 Client request > headers

The HTTP headers are just a list of "name: value" pairs, one per line with the name and the value separated by a ": ". The names are case insensitive. The Headers are used to send metadata between the client and server. The metadata may include the type of file being sent, how many bytes are in the file, what kinds of content can the client or server accept, what is the client user, what is the client password, etc. The Host header is required in all client requests. Other headers may be required depending on the request. Here are a few example Header lines:

```
Host: example.com
Content-type: application/json
Content-length: 129
Accept: application/json
X-Some-Header: 1239asdf
Set-cookie: nsatrack=129
```

Note: *You must send "\r\n" at the end of each header line, but ModusToolbox™ for Connectivity inserts this automatically for you if you use the HTTP library API functions.*

The IANA has a standard list of headers and has developed a registration scheme for people to add more. In addition, you can define your own headers that can mean anything that your server/client can agree on. The names of these Headers are generally in the form of "X-something".

Every request to a server must include the "Host" header. Also, there are two headers for specifying the type and length of the content payload, "Content-type" and "Content-length" which are required for any request that includes a payload.

4b.2.7   Client request > content body

The optional body of the message can be sent by the client. It is just a string of bytes that starts right after the "\r\n" after the headers. The number of bytes sent is specified by the header "Content-length" and the format of the body is specified by the header "Content-type".

The legal values of the "Content-Type" header is also known as a "MIME Type". MIME (an old acronym that means Multipurpose Internet Mail Extension) types are specified by the IANA and can be found on their website. Some of the types that are probably useful for IoT applications include:

- application/json
- application/xml
- text/plain

The list runs to 100's of possible types:

Name	Template	Reference
1d-interleaved-parityfec	application/1d-interleaved-parityfec	[RFC6015]
3gpdash-qoe-report+xml	application/3gpdash-qoe-report+xml	[_3GPP][Ozgur_Oyman]
3gpp-ims+xml	application/3gpp-ims+xml	[John_M_Meredith]
A2L	application/A2L	[ASAM][Thomas_Thomsen]
activemessage	application/activemessage	[Ehud_Shapiro]
activemessage	application/activemessage	[Ehud_Shapiro]
alto-costmap+json	application/alto-costmap+json	[RFC7285]
alto-costmapfilter+json	application/alto-costmapfilter+json	[RFC7285]
alto-directory+json	application/alto-directory+json	[RFC7285]
alto-endpointprop+json	application/alto-endpointprop+json	[RFC7285]
alto-endpointpropparams+json	application/alto-endpointpropparams+json	[RFC7285]
alto-endpointcost+json	application/alto-endpointcost+json	[RFC7285]
alto-endpointcostparams+json	application/alto-endpointcostparams+json	[RFC7285]
alto-error+json	application/alto-error+json	[RFC7285]
alto-networkmapfilter+json	application/alto-networkmapfilter+json	[RFC7285]
alto-networkmap+json	application/alto-networkmap+json	[RFC7285]
AML	application/AML	[ASAM][Thomas_Thomsen]
andrew-inset	application/andrew-inset	[Nathaniel_Borenstein]
applefile	application/applefile	[Patrik_Faltstrom]
ATF	application/ATF	[ASAM][Thomas_Thomsen]
ATFX	application/ATFX	[ASAM][Thomas_Thomsen]
atom+xml	application/atom+xml	[RFC4287][RFC5023]
atomcat+xml	application/atomcat+xml	[RFC5023]
atomdeleted+xml	application/atomdeleted+xml	[RFC6721]
atomicmail	application/atomicmail	[Nathaniel_Borenstein]
atomsvc+xml	application/atomsvc+xml	[RFC5023]
ATXML	application/ATXML	[ASAM][Thomas_Thomsen]
auth-policy+xml	application/auth-policy+xml	[RFC4745]

4b.2.8 Server response message format

Upon receiving a request from a client, the server will respond with:

- **Server Response Start Line**
- Optional **Headers** (same format as the client header)
- Optional **Content Body** (same format as the client Content body)

The client can then:

- Close the connection

or

- Leave the connection open to possibly send another request (the server will eventually close the connection after a timeout of unspecified length. This generally lasts in the range of seconds.

4b.2.9 Server response > start line

The HTTP server response start line will have 4 elements:

- The protocol (probably "HTTP/1.1")
- The Status Code (a number as defined by the IETF)
- The Status Message (a short human readable text version of the status code). This should not be processed by your client to act, use the Status Code instead.
- A line with just "$\backslash r\backslash n$"

An example server response start line (indicating success) is:

```
HTTP/1.1 200 OK
```

Or a failure with the infamous 404 error:

```
HTTP/1.1 404 NOT FOUND
```

4b.2.10 Server response > start line > status codes

The server will respond with a 3-digit status code that is defined by the IETF in [section 10 of RFC2616](). The codes include a wide range of possible things that happened on the server including:

- 200 OK
- 201 Created
- 202 Accepted
- 400 Bad Request
- 404 Not Found

There is a practical discussion of these codes on the Mozilla foundation [website]().

4b.2.11 Server response > start line > status message

In addition to the server status code, the server will respond with a short description of the status code e.g. "OK" or "Created". You should treat the textual response as informational only. You should not parse it and make decisions based on it. For your application logic only use the server status code.

4b.2.12   Server response > headers

The server uses exactly the same Header format scheme as the client.

4b.2.13   Server response > content body

The server uses exactly the same content body format scheme as the client.

4b.3   Client for URLs or "C" URL (CURL)

CURL is a utility for sending and receiving HTTP requests which is built into Unix (Linux, MacOS) and Windows. For Windows, you should use modus-shell or a command terminal (cmd). The PowerShell application in Windows also has a version of CURL, but it is different than what is shown in this example.

CURL is a handy tool to help you figure out what an HTTP website is doing so that you can build your ModusToolbox™ application to do the same thing. CURL will let you create HTTP requests with all the commands (GET, POST, PUT, …), any headers you want, plus any content that you want.

For example, if you want to see what options are available on the "anything" resource on the httpbin.org website you can type the command:
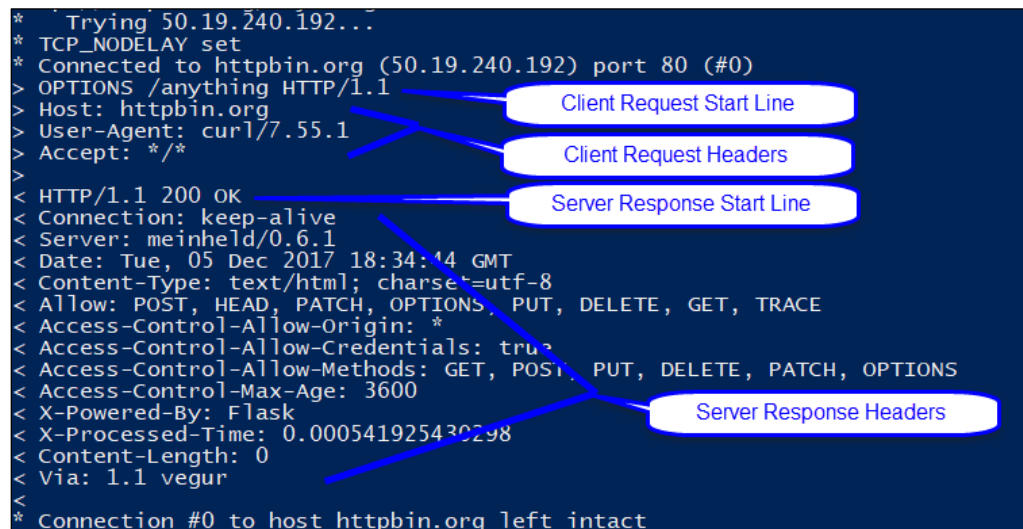
```
curl -v -X OPTIONS http://httpbin.org/anything
```

This example will build an HTTP message that looks like this:

```
OPTIONS /anything HTTP/1.1
Host: httpbin.org
```

The website will then reply with the HTTP options that it supports, and you will see the output on the terminal (because of the -v).

The data sent from your client to the server is shown as lines starting with ">" while the response data from the server is shown as lines starting with "<".

CURL supports both http (non-secure) and https (secure). If you specify the root certificate using the `--cacert` option, CURL will validate the certificate before proceeding with the http transaction. If the server requires the client certificate (e.g. AWS), using the `--cert` option will provide the client's certificate file.

Note that if you specify JSON with the -d option and your JSON has quotes in it, they must be escaped using backslash (\) characters. For example:

```
curl -v -X POST -H "Content-type: application/json" -d "{\"Key1\":\"Value1\"}"
http://httpbin.org/anything
```

If you are using Windows PowerShell to execute the CURL command, replace the double quotes that are <u>not</u> inside the JSON message with single quotes. In that case the above command would be:

```
curl -v -X POST -H 'Content-type: application/json' -d '{\"Key1\":\"Value1\"}'
http://httpbin.org/anything
```

When you specify a body (such as with -d), CURL will automatically calculate and send the Content-length header for you.

In the following table, I show a bunch of CURL commands that are sending requests to httpbin.org which I will talk about in detail in a later section. Some of the useful CURL options are:

Option	Explanation & Example
-v	Verbose: all the http request and response will be echo'd to the screen
	`curl -v http://httpbin.org/get`
| | ```
[Alans-MacBook-Pro:WA101 arh$ curl -v http://httpbin.org/get
*   Trying 54.243.197.181...
* TCP_NODELAY set
* Connected to httpbin.org (54.243.197.181) port 80 (#0)
> GET /get HTTP/1.1
> Host: httpbin.org
> User-Agent: curl/7.51.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Server: meinheld/0.6.1
< Date: Mon, 24 Jul 2017 20:25:44 GMT
< Content-Type: application/json
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Credentials: true
< X-Powered-By: Flask
< X-Processed-Time: 0.000767946243286
< Content-Length: 213
< Via: 1.1 vegur
<
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.51.0"
  },
  "origin": "207.67.13.18",
  "url": "http://httpbin.org/get"
}
* Curl_http_done: called premature == 0
* Connection #0 to host httpbin.org left intact
``` |
-X "command"	CURL will execute the specified HTTP command GET, POST, PUT, DELETE, OPTIONS, TRACE, CONNECT, HEAD.
	If you use PUT, POST you need to specify the content by adding --data
	`curl -v -X OPTIONS http://httpbin.org/get`

Option	Explanation & Example

–H "headername:headervalue"	Adds a header to the HTTP request. You can have multiple –H to add multiple headers. If you specify a header that CURL does automatically e.g. "Content-Type:" it will be overridden by specifying this option.
	
	`curl -v –H 'x-some-custom: someValue' http://httpbin.org`
	
	```[Alans-MacBook-Pro:WA101 arh$ curl -v –H "x-some-custom: someValue" http://httpbin.org/get`
	```* Trying 23.21.145.230...```
	```* TCP_NODELAY set```
	```* Connected to httpbin.org (23.21.145.230) port 80 (#0)```
	```> GET /get HTTP/1.1```
	```> Host: httpbin.org```
	```> User-Agent: curl/7.51.0```
	```> Accept: */*```
	```> x-some-custom: someValue```
	```>```
	```< HTTP/1.1 200 OK```
	```< Connection: keep-alive```
	```< Server: meinheld/0.6.1```
	```< Date: Mon, 24 Jul 2017 20:30:24 GMT```
	```< Content-Type: application/json```
	```< Access-Control-Allow-Origin: *```
	```< Access-Control-Allow-Credentials: true```
	```< X-Powered-By: Flask```
	```< X-Processed-Time: 0.00122499465942```
	```< Content-Length: 248```
	```< Via: 1.1 vegur```
	```<```
	```{```
	``` "args": {},```
	``` "headers": {```
	``` "Accept": "*/*",```
	``` "Connection": "close",```
	``` "Host": "httpbin.org",```
	``` "User-Agent": "curl/7.51.0",```
	``` "X-Some-Custom": "someValue"```
	``` },```
	``` "origin": "207.67.13.18",```
	``` "url": "http://httpbin.org/get"```
	```}```
	```* Curl_http_done: called premature == 0```
	```* Connection #0 to host httpbin.org left intact```

-d "data"
--databinary "data"
	Specifies the data for a PUT, POST. CURL will automatically add the "Content-length:" header.
	
	`curl -v -X PUT –H 'content-type: application/json' -d '{asdf}' http://httpbin.org/put`
	
	```[Alans-MacBook-Pro:WA101 arh$ curl -v -X "PUT" –H "content-type: application/json" -d "{asdf}" http://httpbin.org/put```
	```* Trying 107.20.224.87...```
	```* TCP_NODELAY set```
	```* Connected to httpbin.org (107.20.224.87) port 80 (#0)```
	```> PUT /put HTTP/1.1```
	```> Host: httpbin.org```
	```> User-Agent: curl/7.51.0```
	```> Accept: */*```
	```> content-type: application/json```
	```> Content-Length: 6```
	```>```
	```* upload completely sent off: 6 out of 6 bytes```
	```< HTTP/1.1 200 OK```
	```< Connection: keep-alive```
	```< Server: meinheld/0.6.1```
	```< Date: Mon, 24 Jul 2017 20:33:21 GMT```
	```< Content-Type: application/json```
	```< Access-Control-Allow-Origin: *```
	```< Access-Control-Allow-Credentials: true```
	```< X-Powered-By: Flask```
	```< X-Processed-Time: 0.00127291679382```
	```< Content-Length: 351```
	```< Via: 1.1 vegur```
	```<```
	```{```
	``` "args": {},```
	``` "data": "{asdf}",```
	``` "files": {},```
	``` "form": {},```
	``` "headers": {```
	``` "Accept": "*/*",```
	``` "Connection": "close",```
	``` "Content-Length": "6",```
	``` "Content-Type": "application/json",```
	``` "Host": "httpbin.org",```
	``` "User-Agent": "curl/7.51.0"```
	``` },```
	``` "json": null,```
	``` "origin": "207.67.13.18",```
	``` "url": "http://httpbin.org/put"```
	```}```
	```* Curl_http_done: called premature == 0```
	```* Connection #0 to host httpbin.org left intact```

Option	Explanation & Example

-o filename	Send output to filename. This only sends the content, not the headers to the file
	
	`curl -o blah.json http://httpbin.org/get`
	
	``[Alans-MacBook-Pro:WA101 arh$ curl -o blah.json http://httpbin.org/get
	  % Total % Received % Xferd Average Speed Time Time Time Current
	                                 Dload Upload Total Spent Left Speed
	100 213 100 213 0 0 1543 0 --:--:-- --:--:-- --:--:-- 1554
	[Alans-MacBook-Pro:WA101 arh$ ls blah.json
	blah.json
	[Alans-MacBook-Pro:WA101 arh$ more blah.json
	{
	  "args": {},
	  "headers": {
	    "Accept": "*/*",
	    "Connection": "close",
	    "Host": "httpbin.org",
	    "User-Agent": "curl/7.51.0"
	  },
	  "origin": "207.67.13.18",
	  "url": "http://httpbin.org/get"
	}```

--head	CURL will make the method HEAD. You will need to use the -v to see the headers because there will be no content sent back by the http server
	
	`curl -v --head http://httpbin.org/get`
	
	``[Alans-MacBook-Pro:WA101 arh$ curl -v --head http://httpbin.org/get
	* Trying 75.101.156.200...
	* TCP_NODELAY set
	* Connected to httpbin.org (75.101.156.200) port 80 (#0)
	> HEAD /get HTTP/1.1
	> Host: httpbin.org
	> User-Agent: curl/7.51.0
	> Accept: */*
	>
	< HTTP/1.1 200 OK
	HTTP/1.1 200 OK
	< Connection: keep-alive
	Connection: keep-alive
	< Server: meinheld/0.6.1
	Server: meinheld/0.6.1
	< Date: Mon, 24 Jul 2017 20:35:52 GMT
	Date: Mon, 24 Jul 2017 20:35:52 GMT
	< Content-Type: application/json
	Content-Type: application/json
	< Access-Control-Allow-Origin: *
	Access-Control-Allow-Origin: *
	< Access-Control-Allow-Credentials: true
	Access-Control-Allow-Credentials: true
	< X-Powered-By: Flask
	X-Powered-By: Flask
	< X-Processed-Time: 0.000696897506714
	X-Processed-Time: 0.000696897506714
	< Content-Length: 213
	Content-Length: 213
	< Via: 1.1 vegur
	Via: 1.1 vegur
	
	<
	* Curl_http_done: called premature == 0
	* Connection #0 to host httpbin.org left intact```

Option	Explanation & Example
--cookie "value"	This will add the header "Cookie: value" to your header
| | ```
curl -v --cookie 'name=arh' http://httpbin.org/get
```<br>```
[Alans-MacBook-Pro:WA101 arh$ curl -v --cookie "name=arh" http://httpbin.org/get
*   Trying 23.23.159.159...
* TCP_NODELAY set
* Connected to httpbin.org (23.23.159.159) port 80 (#0)
> GET /get HTTP/1.1
> Host: httpbin.org
> User-Agent: curl/7.51.0
> Accept: */*
> Cookie: name=arh
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Server: meinheld/0.6.1
< Date: Mon, 24 Jul 2017 20:37:33 GMT
< Content-Type: application/json
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Credentials: true
< X-Powered-By: Flask
< X-Processed-Time: 0.000661849975586
< Content-Length: 240
< Via: 1.1 vegur
<
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Connection": "close",
    "Cookie": "name=arh",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.51.0"
  },
  "origin": "207.67.13.18",
  "url": "http://httpbin.org/get"
}
* Curl_http_done: called premature == 0
* Connection #0 to host httpbin.org left intact
``` |
| --cacert server_cert.pem | Verify the certificate of the https connection with the certificate.pem root ca. In the example below, if the httpbin.pem does not match the root certificate received from httpbin.org the connection will fail. |
| | ```
curl --cacert httpbin.pem https://httpbin.org/get
``` |
| --cert client_cert.pem | Send client_cert.pem to the HTTPS server to verify the client identity |
| | ```
curl --cert client_cert.pem https://httpbin.org/
``` |

This link (https://curl.haxx.se/docs/httpscripting.html#The_HTTP_Protocol) takes you to a useful tutorial using CURL with HTTP.

4b.4   Representational State Transfer (REST) & RESTful APIs

REST (https://en.wikipedia.org/wiki/Representational_state_transfer) is a design philosophy developed by Thomas Fielding for his PhD Dissertation. This philosophy has achieved wide acceptance on the Internet, and many people at least pay lip service to supporting it. In Dr. Fielding's thesis he described 7 characteristics: Uniform Interface, Stateless, Cacheable, Client-Server, Layered System, Code-on-demand. If you want to understand more of the philosophy please go read his thesis or google "rest api definition" or "rest api tutorial".

So now what? A RESTful API is a webserver that implements REST. In other words, an HTTP client can interact with a RESTful HTTP server using the principals outlined by REST. Practically and most commonly this means:

- You send and receive JSON documents
- The returned HTTP server status code tells you what happened with your request
- The HTTP resources are nouns such as:
    - /companies returns a list of the companies
    - /companies/ifx is a list of information about Infineon
    - /companies/ifx/products is a list of all Infineon products
- The HTTP client methods are verbs such as:

- GET /companies/cy/products will return a JSON document with a list of all the products
- POST /companies will add a new company to the server (from the attached JSON document)
- DELETE /company/ftdi will delete FTDI from the list of companies

It is common to use options on the resource to perform actions such as:

- Filtering   /companies/cy/products?type=Wi-Fi
- Pagination   /companies?page=27
- Searching   /companies?search=Infineon
- Sorting   /companies?sort=rank_asc

4b.4.1   Web APIs

A Web API is a publicly available RESTful API. On the Internet, there are countless APIs created by both companies and people. There are a bunch of useful ones out there which you can reliably use in your projects. To find APIs, some web directories have been created including:

- https://github.com/APIs-guru/openapi-directory

A few APIs that might be useful include:

- Weather - https://www.wunderground.com/weather/api
- Twitter - https://dev.twitter.com/overview/api
- Google Translate - https://cloud.google.com/translate/docs/translating-text

A vast number of the APIs on the internet use an "API key". This is generally a string of 20ish characters that enable you to access the API. When you register on the website of the API provider, they will tell you the API key. There are two common methods for sending the API keys

- HTTP option   /blah/foo/bar?apikey=1234abcd
- HTTP header   "X-myapikey: 1234abcd"

4b.5   ModusToolbox™ for Connectivity HTTP 1.1 client library

In the previous sections of this chapter I talk about the HTTP Protocol and the way it works and is used. In this section I start the process of explaining how to use ModusToolbox™ for Connectivity to implement HTTP. Fundamentally the ModusToolbox™ for Connectivity http-client library provides you APIs to build the client request lines, headers and content, and parse the output that comes from responses.

ModusToolbox™ for Connectivity has two HTTP libraries, http-client and http-server which provide support for HTTP 1.1 clients and servers respectively. These libraries support both HTTP (non-secure) and HTTPS (secure).

Rather than write all of the code to get a HTTP client running yourself we will provide a template to use in the exercises. That being said, the basic steps to set up an HTTP client using the http-client library are:

1. Include the *wifi-core-freertos-lwip-mbedtls* and *http-client* libraries using the Library manager.
2. Add the following `COMPONENTS` to your *Makefile*:

   ```
   COMPONENTS=FREERTOS MBEDTLS LWIP SECURE_SOCKETS
   ```

3. Add the following `DEFINES` to your *Makefile*:

   ```
   DEFINES+=HTTP_MAX_RESPONSE_HEADERS_SIZE_BYTES=2048
   ```

```
DEFINES += HTTP_USER_AGENT_VALUE="\"mtb-http-client\""
```

Note: *Change "mtb-http-client" to your desired user agent name, this name will be sent with all your requests.*

```
DEFINES += HTTP_DO_NOT_USE_CUSTOM_CONFIG
DEFINES += MQTT_DO_NOT_USE_CUSTOM_CONFIG
```

4. Add the following `#include` to your application code:

    ```
    #include "cy_http_client_api.h"
    ```

5. Initialize the *http-client* library by calling `cy_http_client_init` with no arguments.

6. Optionally, if using TLS:

    a. Declare a structure of type `cy_awsport_ssl_credentials_t` and initialize it to 0:

        ```
        cy_awsport_ssl_credentials_t credentials;
        (void) memset(&credentials, 0, sizeof(credentials));
        ```

    b. Set the client identity, configure the TLS properties, and initialize the server root certificate by populating the following members in the structure created above:
        - `const char *client_cert` – The client's certificate
        - `size_t client_cert_size` – The size of the client's certificate – see notes below
        - `const char *private_key` – The client's private key
        - `size_t private_key_size` – The size of the client's private key – see note
        - `const char *root_ca` – The root certificate with which to verify the server
        - `size_t root_ca_size` – The size of the server's root certificate – see note

Note: *The certificates and keys that you specify must consist of the entire string <u>including</u> the null termination so you must use `sizeof` instead of `strlen` when specifying the 3 sizes.*

Note: *The client's certificate and private key are only required if the server validates the client's identity. If not, those entries (and their size) need not be configured.*

7. Create and configure a structure with the server information:

    a. Create a struct of type `cy_awsport_server_info_t` with the info for the server you want to connect to and initialize it to 0:

        ```
        cy_awsport_server_info_t serverInfo;
        (void) memset(&serverInfo, 0, sizeof(serverInfo));
        ```

    b. Initialize the following data members in the structure:
        - `const char *host_name` – The name of the server you want to connect to e.g. "httpbin.org"
        - `uint16_t port` – The port of the server you want to connect to

8. Register a disconnection callback by setting the value of a `cy_http_disconnect_callback_t` variable to the name of the function you want to be the disconnection callback. For example:

    ```
    cy_http_disconnect_callback_t disconnectCallback = (void*)disconnect_callback;
    ```

9. Create a handle for your HTTP client of type `cy_http_client_t`.

10. Create your HTTP client by calling `cy_http_client_create`. This function takes several arguments:
    - `cy_awsport_ssl_credentials_t*` – A pointer to the struct you populated with TLS information, set this to `NULL` for an unsecure connection
    - `cy_awsport_server_info_t*` – A pointer to the struct you populated with the server information
    - `cy_http_disconnect_callback_t` – The variable you assigned to the disconnect callback function
    - `void*` – User data to be sent while invoking the disconnect callback. Set this to `NULL` if you don't want to send anything
    - `cy_http_client_t*` – A pointer to the handle for your HTTP client

11. Connect to the HTTP server by calling the function `cy_http_client_connect`, which takes the following arguments:

    - `cy_http_client_t` – The handle for your HTTP client (this is <u>not</u> a pointer)
    - `uint32_t` – The timeout value for sending messages in miliseconds
    - `uint32_t` – The timeout value for receiving messages in miliseconds

12. Declare an array of `uint8_t` that is large enough to store all the HTTP messages that you will be sending and receiving.

13. Declare a struct of type `cy_http_client_request_header_t`. You will use this struct to specify what kind of request you want to send. This struct has the following data members that you need to populate:

    - `uint8_t *buffer` – A pointer to the array of `uint8_t` you created previously. This is the buffer that requests you send and responses you receive from the server will be stored in
    - `size_t buffer_len` – The length of your `uint8_t` array
    - `size_t headers_len` – The length of the headers in your request message, this value is updated by the function `cy_http_client_write_header`. You only need to set it if your application generates your headers manually
    - `cy_http_client_method_t method` – The HTTP verb you want to send. Use the following enumeration:

        ```
        typedef enum cy_http_client_method
        {
            CY_HTTP_CLIENT_METHOD_GET,      /**< HTTP Client Method GET string   */
            CY_HTTP_CLIENT_METHOD_PUT,      /**< HTTP Client Method PUT string   */
            CY_HTTP_CLIENT_METHOD_POST,     /**< HTTP Client Method POST string  */
            CY_HTTP_CLIENT_METHOD_HEAD      /**< HTTP Client Method HEAD string  */
        } cy_http_client_method_t;
        ```

    - `int32_t range_start` – Indicates the Start Range from where the server should return. If the range header is not required, set this value to -1
    - `int32_t range_end` – Indicates the End Range until where the data is expected. Set this to -1 if requested range is all bytes from the starting range byte to the end of file or the requested range is for the last N bytes of the file
    - `const char *resource_path` – The path to the resource you want to access

14. Declare a struct of type `cy_http_client_header_t`. This will be the struct you use to specify the headers you want. This struct has the following members:

    - `char *field` – The name of your header

- `size_t field_len` – The length of your header name – see note below
- `char *value` – The value of your header
- `size_t value_len` – The length of your header value – see note

Note: *The field and value are strings and in this case you do NOT want to include the null termination so you must use `strlen` rather than `sizeof` to specify the lengths.*

Note: *If you need more than one header, declare the `cy_http_client_header_t` struct as an array where each element in the array will contain the desired information for one header.*

15. Create the complete HTTP message by calling `cy_http_client_write_header`, which takes the following arguments:

- `cy_http_client_t` – The handle for your HTTP client.
- `cy_http_client_request_header_t*` – A pointer to the struct you created previously.
- `cy_http_client_header_t*` – A pointer to the struct or array you created previously.
- `uint32_t` – The number of elements you created in the header struct array.

Note: *The `cy_http_client_write_header` function automatically includes the Host header, so you don't need to include it. If you don't need any headers besides the host header, you can just specify NULL for the header structure and 0 for the number of headers. For example:*

    `cy_http_client_write_header(clientHandle, &request, NULL, 0);`

16. Declare a variable of type `cy_http_client_response_t`. This will be the location that the server's response to your request is stored in.

17. Send your HTTP message and receive the server's response by calling `cy_http_client_send` with the following arguments:

- `cy_http_client_t` – The handle for your HTTP client
- `cy_http_client_request_header_t*` – A pointer to the struct you created previously
- `uint8_t*` – A pointer to the buffer containing the content body of your message
- `uint32_t` – The length of what you are sending in the content body of your message
- `cy_http_client_response_t*` – A pointer to the variable you created to hold the response

    If you don't want to send a content body, you can use NULL and 0 for the last two arguments.

Note: *When receiving the server's response, `cy_http_client_send` will block until either the buffer you assigned to `cy_http_client_request_header_t.buffer` is full or until the receive timeout you passed into `cy_http_client_connect` is reached.*

18. To parse for specific headers in the server's response message you first need to create another struct or array of structs of type `cy_http_client_header_t`, and then set their `field` members to the header names you want to parse for. Then you can call the function `cy_http_client_read_header`, which takes the following arguments:

- `cy_http_client_t` – The handle for your HTTP client

- `cy_http_client_response_t*` – A pointer to the response from the server
- `cy_http_client_header_t*` – A pointer to the header struct you created in the first part of this step
- `uint32_t` – The number of headers you are parsing for

19. To disconnect your HTTP client from the server you can call the function `cy_http_client_disconnect`, which only takes one argument:

- `cy_http_client_t` – The handle for your HTTP client

20. To delete your HTTP client, you can call the function `cy_http_client_delete` with the only argument being:

- `cy_http_client_t` – The handle for your HTTP client

For more details on using the http-client library, see the http-client library documentation.

4b.6   ModusToolbox™ for Connectivity HTTP 1.1 server library

Rather than write all of the code to get a HTTP server running yourself it is much easier to just create an application using the *HTTPS Server* code example and modify it according to your needs. That being said, the basic steps to set up an HTTP server using the http-server library are:

1. Include the *wifi-core-freertos-lwip-mbedtls* and *http-server* libraries using the Library manager.
2. Add the following COMPONENTS to your *Makefile*:

   ```
   COMPONENTS=FREERTOS MBEDTLS LWIP SECURE_SOCKETS
   ```

3. Add the following DEFINE to your *Makefile*:

   ```
   DEFINES+=MAX_NUMBER_OF_HTTP_SERVER_RESOURCES=<N>
   ```

Note: *Change <N> to be the max number of resources to be supported by your HTTP server. If you don't set it, the default value is 10.*

4. Add the following `#include` to your application code:

   ```
   #include "cy_http_server.h"
   ```

5. Initialize the http-server library by calling `cy_http_server_network_init` with no arguments.

6. Optionally initialize the server identity, configure TLS properties, and initialize the server root certificate by populating a struct of type: `cy_https_server_security_info_t`. This struct has data members:

   - `uint8_t *certificate` – The HTTP server's certificate
   - `uint16_t certificate_length` – The length of the HTTP server's certificate
   - `uint8_t *private_key` – The server's private key
   - `uint16_t key_length` – The length of the server's private key
   - `uint8_t *root_ca_certificate` – A pointer to the root certificate with which to verify clients
   - `uint16_t root_ca_certificate_length` – The length of the root certificate

7. Declare a struct of type `cy_network_interface_t`. Then populate its member data according to the following:

   - `cy_network_interface_type_t type` – The network interface type. Pick this from the enumeration:

     ```
     /** Network interface type */
     typedef enum
     {
         CY_NW_INF_TYPE_WIFI = 0, /**< Wi-Fi network interface */
         CY_NW_INF_TYPE_ETH       /**< Ethernet network interface */
     } cy_network_interface_type_t;
     ```

   - `cy_network_interface_object_t object` – Pointer to the network interface object. You can populate this by creating a `cy_socket_sockaddr_t` variable, populating it with your devices IP address and IP address version, then typecasting it as a `void*`. For example:
     ```
     nw_interface.object = (void *) &https_ip_address;
     nw_interface.type = CY_NW_INF_TYPE_WIFI;
     ```

Note: In the above code nw_interface is of type cy_network_interface_t and https_ip_address is of type cy_socket_sockaddr_t.

8. Create a variable of type `cy_http_server_t`. This will be the handle for your server.

9. Call the function `cy_http_server_create` with the following arguments:

   - `cy_network_interface_t*` – A pointer to the struct you created previously
   - `uint16_t` – The port you want your HTTP server to run on
   - `uint16_t` – The maximum number of client connections that can be accepted
   - `cy_https_server_security_info_t*` – A pointer to the variable you created to hold the security info
   - `cy_http_server_t*` – A pointer to the server handle

10. Create a function to handle requests from clients. You can copy the following example function and modify it according to your needs:

```
/*******************************************************************************
 * Function Name: dynamic_resource_handler
 *******************************************************************************
 * Summary:
 *  Handles HTTPS GET, POST, and PUT requests from the client.
 *  HTTPS GET sends the current CYBSP_USER_LED status as a response to the client.
 *  HTTPS POST toggles the CYBSP_USER_LED and then sends its current state as a
 *  response to the client.
 *  HTTPS PUT sends an error message as a response to the client if the resource
 *  registration is unsuccessful.
 *
 * Parameters:
 *  url_path - Pointer to the HTTPS URL path.
 *  url_parameters - Pointer to the HTTPS URL query string.
 *  stream - Pointer to the HTTPS response stream.
 *  arg - Pointer to the argument passed during HTTPS resource registration.
 *  https_message_body - Pointer to the HTTPS data from the client.
 *
 * Return:
 *  int32_t - Returns HTTPS_REQUEST_HANDLE_SUCCESS if the request from the client
 *  was handled successfully. Otherwise, it returns HTTPS_REQUEST_HANDLE_ERROR.
 *
 *******************************************************************************/
int32_t dynamic_resource_handler(const char* url_path,
                                 const char* url_parameters,
                                 cy_http_response_stream_t* stream,
                                 void* arg,
                                 cy_http_message_body_t* https_message_body)
{
    cy_rslt_t result = CY_RSLT_SUCCESS;
    int32_t status = HTTPS_REQUEST_HANDLE_SUCCESS;
    char https_response[MAX_HTTP_RESPONSE_LENGTH] = {0};
    char *register_new_resource = NULL;

    switch (https_message_body->request_type){
        case CY_HTTP_REQUEST_GET:
            APP_INFO(("Received HTTPS GET request.\n"));

            /* Update the current LED status. This status will be sent
             * in response to the POST request.
             */
            if (CYBSP_LED_STATE_ON == cyhal_gpio_read(CYBSP_USER_LED)){
                led_status = LED_STATUS_ON;
            }
            else{
                led_status = LED_STATUS_OFF;
            }

            sprintf(https_response, HTTPS_STARTUP_WEBPAGE, led_status);
```

```
            /* Send the HTTPS response. */
            result = cy_http_server_response_stream_write_payload(stream, https_response,
sizeof(https_response));

            if (CY_RSLT_SUCCESS != result){
                ERR_INFO(("Failed to send the HTTPS GET response.\n"));
            }
            break;

        case CY_HTTP_REQUEST_POST:
            APP_INFO(("Received HTTPS POST request.\n"));

            /* Send the current LED status before toggling in response to the POST request.
             * The user can then send a GET request to get the latest LED status
             * on the webpage.
             */
            sprintf(https_response, HTTPS_STARTUP_WEBPAGE, led_status);

            /* Toggle the user LED. */
            cyhal_gpio_toggle(CYBSP_USER_LED);

            /* Send the HTTPS response. */
            result = cy_http_server_response_stream_write_payload(stream, https_response,
sizeof(https_response));

            if (CY_RSLT_SUCCESS != result){
                ERR_INFO(("Failed to send the HTTPS POST response.\n"));
            }
            break;

        case CY_HTTP_REQUEST_PUT:
            register_new_resource = (char *)&resource_name[0];
            if (https_message_body->data_length > sizeof(resource_name)){
                /* Report the error response to the client. */
                ERR_INFO(("Resource name length exceeded the limit. Maximum: %d, Received: %d",
sizeof(resource_name), (https_message_body->data_length)));
                sprintf(https_response, HTTPS_RESOURCE_PUT_ERROR, sizeof(resource_name));

                /* Send the HTTPS error response. */
                result = cy_http_server_response_stream_write_payload(stream, https_response,
sizeof(https_response));

                if (CY_RSLT_SUCCESS != result){
                    ERR_INFO(("Failed to send the HTTPS PUT error response.\n"));
                }
            }
            else{
                memcpy(register_new_resource, (char *)https_message_body->data,
https_message_body->data_length);

                /* New resource request need to be queued. */
                if (pdTRUE != xQueueSend(register_resource_queue_handle, (void
*)&register_new_resource, 0)){
                    ERR_INFO(("Failed to send queue message.\n"));
                }
            }
            break;

        default:
            ERR_INFO(("Received invalid HTTP request method. Supported HTTP methods are GET,
POST, and PUT.\n"));
            break;
    }

    if (CY_RSLT_SUCCESS != result){
        status = HTTPS_REQUEST_HANDLE_ERROR;
    }

    return status;
}
```

11. To define a resource, create a struct of type `cy_resource_dynamic_data_t` or `cy_resource_static_data_t` for a dynamic or static resource respectively. For a dynamic resource populate its member data according to the following:

- `url_processor_t resource_handler` – The name of your function that will handle requests for the page specified by this resource
- `void *arg` – An argument to pass into your request handler function. Set this to NULL if you don't want to use it

For a static resource populate its member data according to the following:

- `const void *data` – A pointer to the data for this resource
- `uint32_t length` – The length of the data for this resource

12. To register resources on your HTTP server, call the function `cy_http_server_register_resource` with the following arguments:

- `cy_http_server_t` – The server handle
- `uint8_t*` – A pointer to the URL of the resource
- `uint8_t*` – A pointer to the MIME type of the resource
- `cy_url_resource_type` – The content type of the resource, pick this from the enumeration:

```
typedef enum
{
    CY_STATIC_URL_CONTENT,       /**< Page is constant data in memory-addressable area. */
    CY_DYNAMIC_URL_CONTENT,      /**< Page is dynamically generated by a @ref url_processor_t
        type function. */
    CY_RESOURCE_URL_CONTENT,     /**< Page data is provided by a Resource which may reside off-
        chip. Currently this URL type is not supported. */
    CY_RAW_STATIC_URL_CONTENT,   /**< Same as @ref CY_STATIC_URL_CONTENT, but the HTTP header must
        be supplied as part of the content. */
    CY_RAW_DYNAMIC_URL_CONTENT,  /**< Same as @ref CY_DYNAMIC_URL_CONTENT, but the HTTP header must
        be supplied as part of the content. */
    CY_RAW_RESOURCE_URL_CONTENT  /**< Same as @ref CY_RESOURCE_URL_CONTENT, but the HTTP header
        must be supplied as part of the content. */
} cy_url_resource_type;
```

- `void*` – A pointer to the static or dynamic resource type structure, this is the struct you created previously

Note: **Note** You cannot register a static resource while the server is running. If you have already called `cy_http_server_start` and wish to register a static resource then you must first call `cy_http_server_stop` before registering a new resource.

13. To start the HTTP server, call the function `cy_http_server_start` with the argument:

- `cy_http_server_t` – The server handle

For more details on using the http-server library, see the http-server library documentation.

4b.7   Httpbin.org

Httpbin.org is a website that was created to help people test their HTTP (and HTTPS) requests. You can send PUT, POST, GET etc. and it will respond with something simple, often in JSON format to "echo" what you sent.

You can build HTTP requests in CURL to test your ideas about how to interact with different HTTP endpoints (i.e. resources). You could, for example, go to http://httpbin.org/get in your browser and it will return:

```
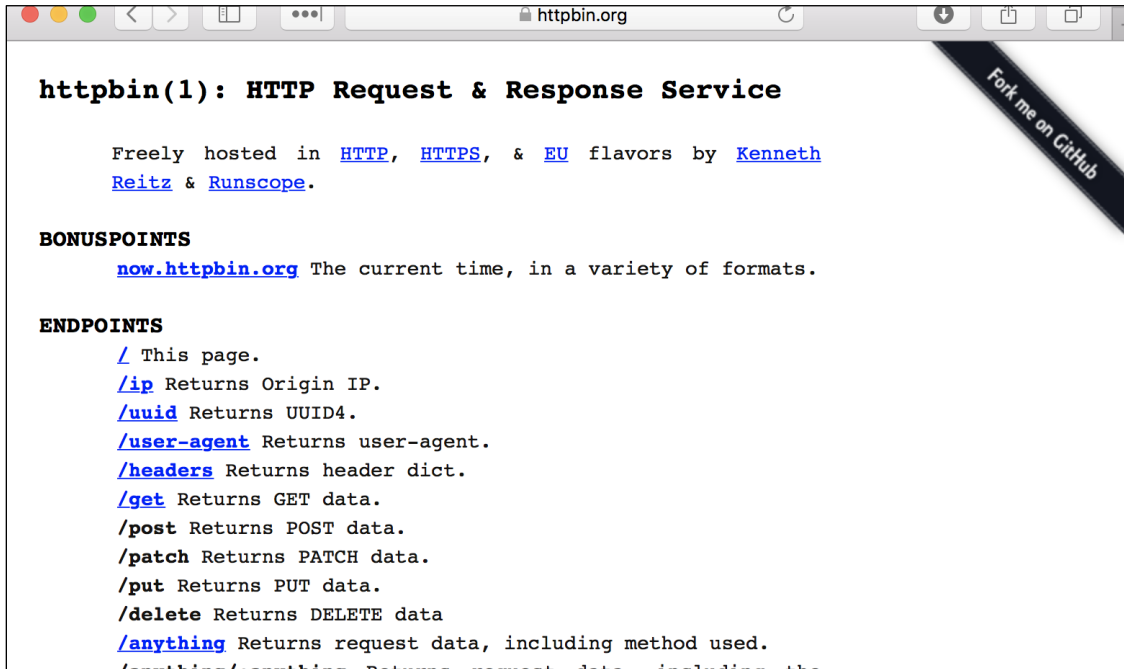{
  "args": {},
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "en-us",
    "Connection": "close",
    "Cookie": "_gauges_unique_day=1; _gauges_unique_month=1; _gauges_unique=1;
_gauges_unique_year=1",
    "Host": "httpbin.org",
    "Referer": "https://httpbin.org/",
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/603.3.8
(KHTML, like Gecko) Version/10.1.2 Safari/603.3.8"
  },
  "origin": "69.23.226.142",
  "url": "https://httpbin.org/get"
```

There are a bunch of endpoints (resources) which allow different HTTP methods to be tested. We will mostly use the /anything resource since (as the name implies) it allows just about anything. It will respond back with an echo of the data that you sent so you can compare to what you intended to send.

```
httpbin(1): HTTP Request & Response Service

    Freely hosted in HTTP, HTTPS, & EU flavors by Kenneth
    Reitz & Runscope.

BONUSPOINTS
    now.httpbin.org The current time, in a variety of formats.

ENDPOINTS
    / This page.
    /ip Returns Origin IP.
    /uuid Returns UUID4.
    /user-agent Returns user-agent.
    /headers Returns header dict.
    /get Returns GET data.
    /post Returns POST data.
    /patch Returns PATCH data.
    /put Returns PUT data.
    /delete Returns DELETE data
    /anything Returns request data, including method used.
    /anything/:anything Returns request data, including the
```

4b.8   Exercises

Exercise 1: Use CURL to access http://httpbin.org

The website httpbin.org is a public server HTTP debugging utility. It will let you make requests and then tell you what is happening by echoing back what you sent in JSON format.

☐   1.   Use a web browser to view http://httpbin.org

☐   2.   From a command terminal run: `CURL -v -X GET http://httpbin.org/anything`

Note: *If your Wi-Fi AP has a proxy server, CURL will not be able to connect (unless you specify the proxy server).*

Note: *If you are using Windows, you will have to download CURL. To use it:*

a.   Go to https://curl.se/windows/ and download CURL. Unzip the file you downloaded and navigate to *curl-version/bin/* using the file explorer.

b.   **Shift-Right-Click** in the File Explorer window and select either **Open command window here** or **Open PowerShell window here**.

c.   Alternately, you can open a command window (cmd) or a modus-shell and manually change directory (cd) to the location listed above.

d.   From the new window, run the command as:
```
curl <other arguments> (for Command window or modus-shell)
./curl <other arguments> (for PowerShell)
```

Note: *The leading dot and slash are required in Windows PowerShell because by default it aliases "curl" to a different function that is similar to CURL but is not the same.*

e.   If you are using PowerShell, remember to use single quotes instead of double quotes (except inside a JSON string).

☐   3.   Use CURL to do a POST to the resource /anything

☐   4.   Use CURL to do a GET from the resource /html.

Exercise 2: Use CURL to access https://httpbin.org using TLS

Use a TLS connection to access httpbin.org. The steps are:

☐   1.   Use a web browser to save the certificate for https://httpbin.org.

Note: *The steps to do this were covered in Chapter 6B.*

a.   You must save the root certificate to use in CURL. It will not work with the intermediate certificate or the httpbin.org certificate.

Note: *Make sure you are viewing the root certificate before you save it.*

b.   Put the certificate in the same folder as the CURL executable to simplify specifying the path.

☐   2.   Use CURL to do a GET from https://httpbin.org/anything
You will need to use the `--cacert` option in CURL to provide the certificate file.

☐   3.   Look at the log file to see the TLS handshaking occurring.

Exercise 3: Run the HTTPS Server example application

☐   1.   Create a new application using the **HTTPS Server** code example. Name it **ch4b_ex03_https_server**.

☐   2.   Update the Wi-Fi configuration parameters in *secure_http_server.h* files.

☐   3.   If you are taking this course in a classroom setting, update the macro `HTTPS_SERVER_NAME` in *secure_http_server.h* to be something unique so that other groups don't connect to your server by mistake. You could use your initials for this.

☐   4.   Follow the instructions in the README.md file in the code example to generate and update the required certificates.

Note: *You will need to use the data in mysercurehttpserver.local.crt, mysercurehttpserver.local.key and rootCA.crt files in the file secure_http_server.h. Use the utility referenced in the README.md file to convert the PEM format to strings.*

☐   5.   Program the server application to your kit.

☐   6.   Once the server starts, use cURL to connect to the server and send messages.

Note: *You will need to provide the files rootCA.crt, mysercurehttpclient.crt, and mysecurehttpclient.key as command line arguments to cURL. See the operation instructions in the README.md file for details.*

Note: *Instead of using cURL, you can connect to the server using a web browser. If you want to try this approach, see the README.md file in the   **HTTPS Server** application. In order for this to work, you will need to install the certificates in your browser.*

Exercise 4: Use your kit to GET data from httpbin.org

Create and then run an application to get data from httpbin.org using your development kit. The application will perform a GET from the /html resource and then from the /anything resource. The steps are:

☐   1.   Create an application using the template in the class files under *Templates/ch04b_ex04_httpbin_get*.

☐   2.   Use the Library Manager to include the http-client library.

     Note: *The Makefile has already been updated with the required* COMPONENTS *and* DEFINES*.*

☐   3.   Update the Wi-Fi configuration parameters in *http_client.h*.

☐   4.   Search for "TODO" in *http_client.c* and *http_client.h* and implement the required functionality.

☐   5.   Open a serial terminal connection to your kit.

☐   6.   Build, program, and run the project.

☐   7.   Observe the results in the terminal window.

Questions:

☐   1.   Which server port is used for HTTP (non-secure)?

☐   2.   What header(s) is/are sent with each request?

☐   3.   What is the variable "connected" used for? Why is it needed?

☐   4.   Uncomment the section of code to wait for the server to disconnect between requests. How long does the server wait before closing the connection?

Exercise 5: Use your kit to GET data from httpbin.org using TLS

Create and then run an application to GET data from httpbin.org using your development kit using a TLS connection. The steps are:

☐   1.   Create an application using the template from *Templates/ch04b_ex05_httpbin_get_secure*.

> *Note:*   *The http-library has already been included in the application and the Makefile has been updated with the required* `COMPONENTS` *and* `DEFINES`.

☐   2.   Update the Wi-Fi configuration parameters.

☐   3.   Search for TODO in *http_client.c* and *http_client.h* and implement the required functionality.

☐   4.   Open a serial terminal connection to your kit.

☐   5.   Build, program, and run the project.

☐   6.   Observe the results in the terminal window.

Questions:

☐   1.   Which server port is used for HTTPS (secure)?

☐   2.   What function call and parameter specifies that the connection should use TLS?

Exercise 6: Use your kit to POST data to httpbin.org

Create and then run an application to POST JSON data to httpbin.org using your development kit. The resource is /anything. The data you send will be echoed back to you by the server. The steps are:

1. Create an application using the template from *Templates/ch04b_ex06_httpbin_post*.

   Note: *The http-library has already been included in the application and the Makefile has been updated with the required* COMPONENTS *and* DEFINES.

2. Update the Wi-Fi configuration parameters.

3. Search for TODO in *http_client.c* and implement the required functionality.

   Use a short JSON message as the body of the message. For example, you could use the key/value pair {"having_fun":"yes"}.

4. Open a serial terminal connection to your kit.

5. Build, program, and run the project. Observe the results.

Questions:

1. What headers are sent with the POST request?

2. What is the JSON content that is posted?

Exercise 7: Use your kit to POST data to httpbin.org using TLS

Create and then run an application to POST data to httpbin.org using your development kit using a TLS connection. The steps are:

☐
1. Create an application using the template from *Templates/ch04b_ex07_httpbin_post_secure*.

Note: *The http-library has already been included in the application and the Makefile has been updated with the required* COMPONENTS *and* DEFINES.

☐
2. Update the Wi-Fi configuration parameters.

☐
3. Search for TODO in *http_client.c* and implement the required functionality.

Use a short JSON message as the body of the message. For example, you could use the key/value pair `{"having_fun":"yes"}`.

☐
4. Open a serial terminal connection to your kit.

☐
5. Build, program, and run the project. Observe the results.

4b.9   Appendix

Answers to the questions asked in the exercises above are provided here.

4b.9.1   Exercise 4 Answers

1. Which server port is used for HTTP (non-secure)?

    Port 80.

2. What header(s) is/are sent with each request?

    In this case, three headers are sent:
    ```
    Host: httpbin.org
    User-Agent: mtb-http-client
    X-Amzn-Trace-Id: Root=...
    ```

3. What is the variable "connected" used for? Why is it needed?

    The variable "connected" is used to determine if the connection to the server is still active. It is needed because the server can disconnect at any time. Therefore, before sending another request, we need to see if the connection is still there. If not, we need to restart everything.

4. Uncomment the section of code to wait for the server to disconnect between requests. How long does the server wait before closing the connection?

    The server disconnects after about 60 seconds of inactivity.

4b.9.2   Exercise 5 Answers

1. Which server port is used for HTTPS (secure)?

    Port 443.

2. What function call and parameter specifies that the connection should use TLS?

    The 1st parameter to `cy_http_client_connect` is of type `cy_awsport_ssl_credentials_t` and has all of the TLS information.

4b.9.3   Exercise 6 Answers

1. What headers are sent with the POST request?

    There are 5 headers:
    ```
    Host: httpbin.org
    Content-Type: application/json
    Content-Length: 18  (back-slashes don't count in the content length)
    User-Agent: mtb-http-client
    X-Amzn-Trace-Id: Root=...
    ```

2. What is the JSON content that is posted?

    The JSON is a key-value pair of `{"having_fun":"yes"}`

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.