**Chapter 7**

**HTTP (Video 7-1)**

Hi, I’m Alan Hawse, welcome back to WICED Wifi 101. In the previous video I gave you an overview of the Cloud and the four most commonly used Cloud protocols – remember… HTTP, MQTT, AMQP and CoAP. In this video we'll get down to the guts and the glory of HTTP.

When HTTP came on the scene during the early 1990's, it was used to send static HTML pages. Remember Mosaic and Netscape? Maybe not, but I’m old so there it is. Over time, dynamic HTTP came into common use for reading and writing databases and creating HTML on the fly. Many companies built giant teams of people to deploy and develop these HTTP based applications. This included all of the computers, networks, software, disk drives, databases, languages, etc.

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

There are two versions of HTTP in use today - 1.1 and 2.0. HTTP 1.1 was released in 1999 and as of 2017 still serves the bulk (>50%) of the web traffic. HTTP 2.0 was released in 2015, and it brings many performance benefits, but it's seen a very slow uptake in the market (as of 2017 only ~30% of web browsers support it).

Both versions are supported by WICED. Unfortunately, as is common in computers, HTTP 1.1 and HTTP 2.0 are exactly the same except for everything – they made them really different. Well… that isn’t exactly true, but they're different enough that I will cover them separately in two different chapters. We'll start with 1.1 in this chapter.

All righty… HTTP 1.1 is an application layer, single transaction, stateless, plain-text, client-server protocol. Wow… that quite a bit… alright… all this really means is that a client - like your WICED device - opens up a connection to a TCP Server inside of the Cloud and then it sends it an ASCII text request. The Server then processes that request and replies with an ASCII text response, and then connection is either left open or it's maybe closed depending on some configuration settings.

Let me say that again. HTTP is just a simple text message. Both HTTP requests and HTTP responses are made up of a start line, an optional group of headers, and finally an optional content body. That’s it. Now, let’s talk about each of these three sections.

**Client Request**

An HTTP client request begins with a start line which is made up of 5 elements. They are:

1. The Method

2. The Requested Resource

3. Optional Options

4. The version of HTTP that will be used, and for HTTP 1.1 that will always be HTTP/1.1

5. A "\r\n" which indicates the end of the start line

Here is an example of a legal client request start line:

GET /ask HTTP/1.1

In this example, the method is GET, the resource is /ask, and the version is HTTP/1.1. In this case there are no options and the "\r\n" – I didn't show it to you since the library will take care of that for you automatically, but it's there – trust me.

When you're using a web browser, all it does when you type in the http:// line at the top of the browser is open up a TCP connection to the server, and then send a GET request.

As I mentioned in the last video, GET is one of the 9 HTTP methods or verbs. The others are PUT, POST, HEAD, PATCH, DELETE, CONNET, OPTIONS, and finally TRACE. It turns out that not all of the servers support all of the 9 methods. And, unfortunately, what each method does on a particular server and which ones change the data – all of that is done by convention, so you need to make sure you understand what your HTTP calls do on a given server. For IoT devices, GET, PUT, and POST are the only methods that are commonly used. The names reflect what they do. GET just gets data from a server, and PUT just sends data to the server, and POST just sends a little bit of data to the server.

The resource is specified as a path that you want to access on the server. In the example I showed a minute ago, the path is just /ask. That means you are requesting to GET the information from the server that is located at /ask.

If you want to add some options to the request – remember that optional options thing that I said before that was really awkward – well, you might build a start line just like this:

GET /ask?user=myname&password=secret HTTP/1.1

For this example, I'm also asking to GET a resource located at /ask, but in this case, I'm providing a username and a password as options in the start line. Options are separated from the resource by a question mark and multiple options are separated by an ampersand.

After the HTTP start line comes one or more headers which are just name/value pairs specified one per line separated by a colon – in other words, the key, colon, the value. The headers are used to send metadata between the client and the server such as the type of the file that's being sent, or how many bytes of data are in the file, or one of a whole bunch of different options.

For a Client Request, there must be a “Host” header. And, if the request has a body, there are two other headers which are required that specify the type of content in the body as well as the length. For example, here are headers that say we are connecting to a host called example.com, we're sending a JSON document and that document is 129 bytes long.

* Host: example.com
* Content-type: application/json
* Content-length: 129

As with the end of the start line, each header must have a "\r\n" at the end of it, but again the WICED HTTP API library takes care of that for you.

There are a bunch of other legal headers, and it will depend on what your server is doing which ones you might need to use. The IANA, also known as the Internet Assigned Number Authority, has a standard list of headers, but you can also define your own header as long as your client and your server agree on what the header means. Custom headers usually start with "X-" such as "X-MyCustomHeader" or "X-WICED\_WIFI\_is\_awesome".

After the last header is an additional line that just has a "\r\n". This is the indication to the server that the headers are finished. And… as you probably guessed, the library will take care of that for you when you send a request.

After the start line and the headers comes the optional content body. The body is just a stream of bytes that starts right after the final \r\n in the headers. If your request has a body, remember you must specify the Content-type in the headers and the Content-length. Again, the IANA has a list of content types which are also known as MIME types. The ones that are most useful for the IoT and IoT applications are:

* application/json
* application/xml
* text/plan

The number of characters you send in the body must match the Content-length header.

In summary, a client request has three parts, a request line, one or more headers, and an optional body of bytes.

After you send the request, the Server will respond with a similar message called “the server response”

**Server Response**

The server response has a start line, optional headers, and optional content body. Does that sound familiar? Well, it should because it's very much like the client request.

The Server Response begins with a start line which is made up of 4 elements. They are:

1. The Protocol

2. A Status Code

3. A Status Message

4. A "\r\n" which indicates the end of the start line

A Server Response might look like this:

HTTP/1.1 200 OK

The protocol is HTTP/1.1, 200 is the Status Code, and OK is the status message.

Another one that you have likely seen from your web browser looks like this:

HTTP/1.1 404 NOT FOUND

The famous 404 error.

The Status Code is always 3 digits and it is also defined by the Internet Engineering Task Force, or the IETF. If the first digit is a 2 it generally means success. 200 commonly means OK, 201 means Created, 202 means Accepted. Other leading digits – like 4 – mean something probably went wrong.

The exact wording of the Status Message that the server sends you is dependent on the server's implementation, so you should always use the Status Code in the firmware, and you should never rely on the Status Message. For example, 404 might return "NOT FOUND", or "SERVER NOT FOUND", or "PAGE NOT FOUND", or some other ridiculous thing, and it doesn't matter because you shouldn't parse it – you should use the 404 to know what happened.

After the Status Message is a "\r\n" which – as you guessed – is included for you by the library.

After the Start Line comes the Response headers. These use the exact same format as the Request headers. If the response has a content body, then the headers will include a Content-type to tell you the format of the file that's attached and a Content-length to tell you how long it is. In addition, you will typically get other important informational headers.

After the headers comes the body – if there is one. It is the same format as the request body.

Once the response has been received from the server, the client can close the connection or leave it open to possibly send another request. The server will eventually close the connection after a timeout which is generally in the range of seconds. It is pretty common for every request/response to use a new connection and then close it right away, but it doesn't have to happen that way so before you send a request you should make sure your connection is still open.

**Wrap-Up**

So that's it for HTTP 1.1 format. It's not really that difficult to understand, is it?

In chapter 7B of the manual – which is available on the Cypress Video tutorial website - I have sections that discuss CURL, REST, and Web APIs. CURL is a very commonly used command line tool that's useful in experimenting with HTTP requests and responses.

REST is a design philosophy that most web sites adhere to.

Web APIs are collections of useful functions that are available on the Web.

I won't talk about them in this video, but please read about them in the manual and try out the rest of the exercises.

That’s it for the introduction. In the next video, I'll show you how to use our WICED library that's built into our software that will help you format HTTP requests (remember all those \r\n's?) and they'll help you parse out the information from HTTP responses.

As always, you can post your comments and questions in our Wifi developer community or you're welcome to email me at alan\_hawse@cypress.com or tweet me @askioexpert with your comments, suggestions, criticisms and questions. Thank you!