**Chapter 7**

**HTTP (Video 7-2)**

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 into the guts of HTTP.

When HTTP came on the scene in the early 1990's, it was used to send static HTML pages. Remember Mosaic and Netscape? Over time, dynamic HTTP came into common use for reading and writing databases and creating HTML on the fly. Many companies built big teams of people to develop and deploy HTTP based applications. This included all the computers, networks, software, disk drives, databases etc….

As the IoT emerged, it was only natural and financially advantageous for companies to extend their existing 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 has 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 it still serves the bulk (>50%) of the web traffic. HTTP 2.0, which was released in 2015, brings many performance benefits but has seen slow uptake in the market (as of 2017 only ~30% of web browsers support it).

Both versions are supported by WICED. And unfortunately, as is common in computers HTTP 1.1 and HTTP 2.0 are exactly the same except for everything is different. Well… that isn’t exactly true, but they are different enough that I will cover them separately. Starting 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 is a mouthful…. All this really means is that a client - like your WICED device - opens a connection to a TCP Server in the Cloud and sends an ASCII text request. The Server then processes the request and then replies with an ASCII text response, and the connection is either left open for more messages or is closed.

Let me say that again. HTTP is just a simple text message. Both HTTP requests and responses are made up of a start line, an optional group of headers, and an optional content body. That’s it. Now, let’s talk about each of the 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 always be HTTP/1.1 for us
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. The "\r\n" is not shown since the library will take care of it for you, but it is there – trust me.

When you are using a web browser, all it does when you type an http:// line is open an 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 TRACE . It turns out that not all servers support all 9 methods. And, unfortunately, what the method does on a particular server and which ones change data are 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, gets data from a server, and PUT and POST sends 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 /ask. That means that you are requesting to GET the information from a server located at /ask.

If you want to add some options to the request you might have a start line 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 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. The headers are used to send metadata between the client and server such as the type of file is being sent, how many bytes of data are in the file, etc.

For a Client Request, there must be a “Host” header. And, If the request has a body, there are two other headers required which specify the type of content in the body and the length. For example, here are headers that say we are connecting to a host called example.com, we are 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 "\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 need to use. The IANA – Internet Assigned Number Authority has a standard list of headers, but you can also define your own as long as your client and server agree on what they mean. Custom headers usually start with "X-" such as "X-MyCustomHeader".

After the last header is an additional line that has just "\r\n". This is the indication 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, you must specify the Content-type in the headers. Again, the IANA has a list of content types which are also known as MIME types. The ones that are most useful for 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, zero 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 request –has a start line, optional headers, and optional content body. Sound familiar?

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 a web browser looks like this:

HTTP/1.1 404 NOT FOUND

The Status Code is always 3 digits and it is defined by the Internet Engineering Task Force, or IETF. If the first digit is a 2 it generally means success. 200 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 is server dependent, so you should always use the Status Code in the firmware – don't rely on the Status Message. For example, 404 might return "NOT FOUND", "SERVER NOT FOUND", "PAGE NOT FOUND", or something else.

After the Status Message is a "\r\n" which - you guessed it – 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 and Content-body. In addition you will typically get other 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.

So that's it for the HTTP 1.1 format. 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 command line tool that is very useful in experimenting with HTTP requests and responses, REST is a design philosophy that most web sites adhere to, and Web APIs are collections of useful functions 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 exercises.

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

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