Chapter 7C: Cloud Connectivity using HTTP 1.1

At this end of Chapter 7C you will understand:

- The HTTP 1.1 protocol
- The architecture and use model of the WICED http_client library
- 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 in WICED using TLS
- How to test your HTTP client using HTTPBIN
- How to use the IoT Cloud Provider InitialState.com

7C.1 INTRODUCTION 2
7C.2 HTTP 1.1 PROTOCOL 2
7C.2.1 Client Request Message Format 3
7C.2.2 Client Request → Start Line 3
7C.2.3 Client Request → Start Line → HTTP Methods 3
7C.2.4 Client Request → Start Line → Resources 4
7C.2.5 Client Request → Start Line → Options 5
7C.2.6 Client Request → Headers 5
7C.2.7 Client Request → Content Body 6
7C.2.8 Server Response Message Format 6
7C.2.9 Server Response → Start Line 7
7C.2.10 Server Response → Start Line → Status Codes 7
7C.2.11 Server Response → Start Line → Status Message 7
7C.2.12 Server Response → Headers 8
7C.2.13 Server Response → Content Body 8
7C.3 CLIENT FOR URLS OR "C" URL (CURL) 8
7C.4 REPRESENTATIONAL STATE TRANSFER (REST) & RESTFUL APIS 13
7C.4.1 Web APIs 14
7C.5 WICED HTTP 1.1 CLIENT LIBRARY 15
7C.6 HTTPBIN.ORG 17
7C.7 INITIAL STATE 18
7C.7.1 Introduction 18
7C.7.2 Using Initial State 20
7C.8 AMAZON WEB SERVICES IOT 26
7C.9 EXERCISE(S) 27
Exercise - 7C.1 Httpbin.org 27
Exercise - 7C.2 Initial State – Virtual LED Controlled by APIARY and CURL 28
Exercise - 7C.3 Initial State – LED State Controlled by Hardware 28
Exercise - 7C.4 (Advanced) Initial State – Temperature & Humidity 29
Exercise - 7C.5 (Advanced) Initial State – Graphing Temperature & Humidity 29
Exercise - 7C.6 (Advanced) Use Web APIs for Temperature Conversion 29
Exercise - 7C.7 (Advanced) Find and Use Twitter Web APIs 29
Exercise - 7C.8 (Advanced) Example.com 30
7C.10 RELATED EXAMPLE "APPS" 30
7C.11 KNOWN ERRATA + ENHANCEMENTS + COMMENTS 30

7C.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.

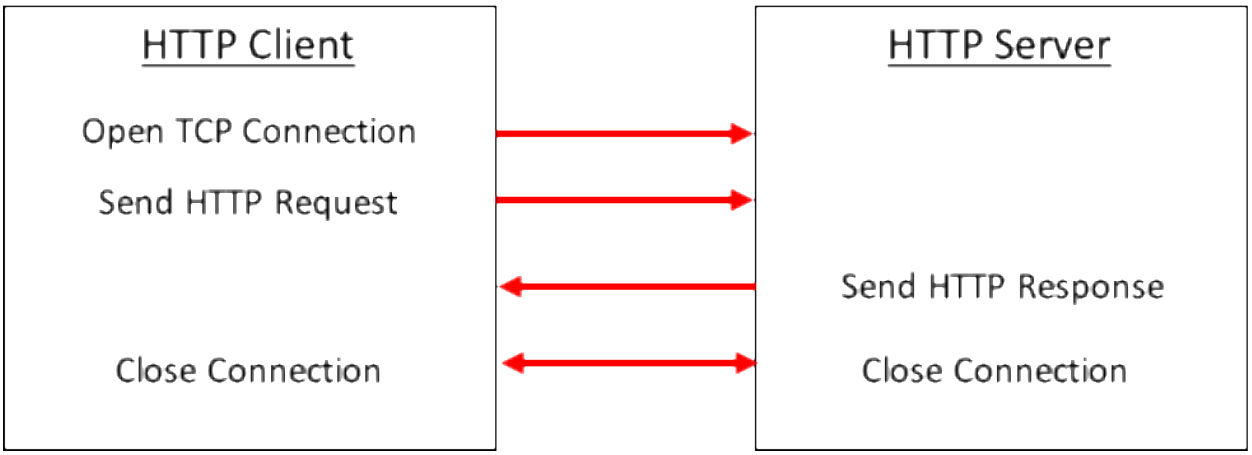
There are essentially two versions of HTTP, 1.1 and 2.0. Although conceptually similar, they are materially different in their implementation and as such are treated as two separate chapters in this class.

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).

WICED supports both protocols, but with two separate libraries.

7C.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 WICED 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. Cookie.



HTTP Requests (both Client and Server) are made up of one mandatory request line, an optional group of HTTP headers (same format for Client and Server) and one optional Content body (same format for Client and Server).

7C.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 (7C.2.2)**
- **Headers (7C.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 (7C.2.7)** (one payload with as many bytes as required e.g. a file or an html page or a JSON document)

7C.2.2 Client Request → Start Line

The Client Request Start Line has five elements

- The **HTTP Method (7C.2.3)**
- The requested **Resource (7C.2.4)** path
- Optional**: Options (7C.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\r\n
```

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

7C.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.

GET (safe, idempotent) – 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 (7C.2.7) of the Content Body "Content-type".

HEAD (safe, idempotent) – This Method performs the same operation as "GET" except it only replies with the Headers and does not return the Content Body.

PUT (unsafe, idempotent) – 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".

PATCH (unsafe, idempotent) – With this method 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}".

POST (unsafe, non-idempotent) – 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".

DELETE (unsafe, idempotent) – This Methods asks the Server to remove the Resource.

OPTIONS (safe, idempotent) – This Method 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.

TRACE (safe, idempotent) – This Method is an infrequently implemented debugging Method that should cause the server to reply with the Client Message (echo'd back).

CONNECT – This method 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 people who implement web servers like that should be beaten for the dogs that they are.

7C.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.

7C.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". These options must be encoded with "url encoding".

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
```

7C.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 … and on and on and on. 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
```

You must send "\r\n" at the end of each header line, but WICED inserts this automatically for you if you use the WICED 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.

7C.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 last header. 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]

7C.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 … generally in the range of seconds.

7C.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 blank line of exactly "\r\n"

An example server response line (indicating success) is:

```
HTTP/1.1 200 OK
```

Or a failure with the infamous 404 error:

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

7C.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.

7C.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.

7C.2.12 Server Response → Headers

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

7C.2.13 Server Response → Content Body

The server uses exactly the same Content Body format scheme as the client.

7C.3 Client for URLs or "C" URL (CURL)

CURL is a utility for sending and receiving HTTP requests which built into Unix (Linux, MacOS) and is also available for Windows (but not built in). CURL is a handy tool to help you figure out what an HTTP website is doing so that you can build your WICED program 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. As with most Unix utilities it is completely out of control with regards to the number of options.

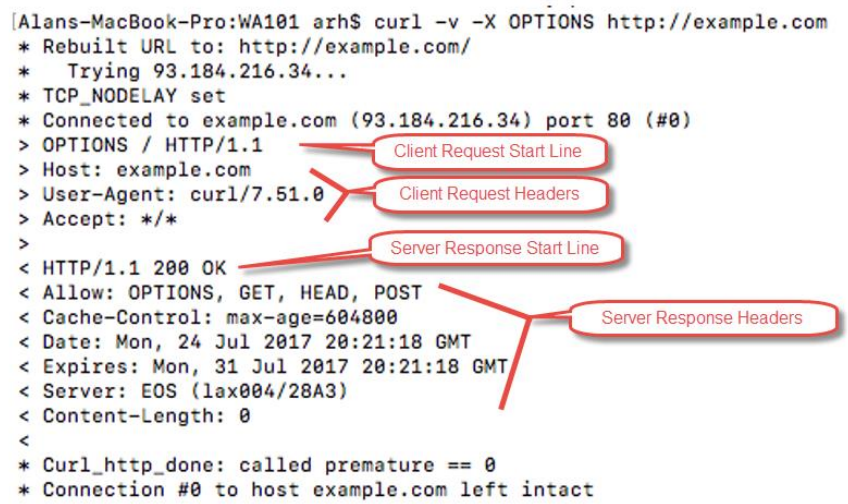
For example, if you want to see what options are available on a website you can type the command

curl -v -X "OPTIONS" http://example.com

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

OPTIONS / http/1.1

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). In this case, the server supports OPTIONS, GET, HEAD, and POST.



CURL supports both http and https. If you specify the root certificate using the --cacert option, CURL will validate the certificate before proceeding with the http transaction.

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://example.com

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

In the table below, 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:

<table>
<tr><th>Option</th><th>Explanation & Example</th></tr>
<tr><td rowspan="2">-v</td><td>Verbose: all the http request and response will be echo'd to the screen</td></tr>
<tr><td>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</td></tr>
<tr><td rowspan="2">-X "command"</td><td>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</td></tr>
<tr><td>curl –v –X "OPTIONS" http://httpbin.org/get</td></tr>
<tr><td rowspan="2">-H "headername:headervalue"</td><td>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.</td></tr>
<tr><td>curl -v -H "x-some-custom: someValue" http://httpbin.org</td></tr>
</table>

	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

<table>
<tr><td></td><td>

```
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
```

</td></tr>
<tr><td rowspan="2">-o filename</td><td>Send output to filename. This only sends the content, not the headers to the file</td></tr>
<tr><td>

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"
}
```

</td></tr>
<tr><td rowspan="2">--head</td><td>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</td></tr>
<tr><td>curl –v --head http://httpbin.org/get</td></tr>
</table>

	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

--cookie “value”	This will add the header “Cookie: value” to your header
	
	curl –v --cookie “name=arh” http://httpbin.org/get
	
	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.
	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.

7C.4 Representational State Transfer (REST) & RESTful APIs

REST 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" and you will find more lunacy, militancy and religion than you probably care to read about.

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 return a list of the companies
  - /companies/cy is a list of the information about Cypress
  - /companies/cy/products a list of all of Cypress 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=wifi
- Pagination /companies?page=27
- Searching /companies?search=Cypress
- Sorting /companies?sort=rank_asc

7C.4.1 Web APIs

A Web API is a publicly available RESTful API. On the Internet, there is a wild-wild-west of APIs. Companies and people open their APIs for all the normal reasons including profit, fame, ego, altruism etc. 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:

- http://www.apiforthat.com
- https://www.programmableweb.com/category/all/apis
- 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”

7C.5 WICED 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 WICED to implement HTTP. Fundamentally WICED provides you APIs to build the Client Request Line, Headers and Content, then parse the output that comes back in the form of a server response.

The WICED SDK has several built-in HTTP libraries including protocols/HTTP_Client which provides support for HTTP 1.1 Clients. You can find the documentation for this library under "Components→ IP Communication → HTTP → HTTP Client". This library supports both HTTP and HTTPS.

To make the HTTP_Client library work you:

- Initialize the http client — http_client_init
- Optionally initialize the client identity — wiced_tls_identity
- Optionally configure the TLS properties — http_client_configuration_info_t
- Optionally initialize the HTTP Server root cert — wiced_tls_init_root_ca_certificates
- Make a connection to the HTTP server — http_client_connect
- Initialize the HTTP request — http_request_init
- Initialize an array of HTTP headers — http_header_field_t[]
- Write the headers — http_request_write_header
- Write the end (the blank line "\r\n") — http_request_write_end_header
- Optionally write the content body — http_request_write
- Flush the writes — http_request_flush

Then you wait for the callback. In the callback function (which your registered when you created the client) you will get the arguments, http_client_t *, http_event_t and http_response_t.

http_client_t: The same callback function can be used to process requests from multiple http_client_t structures, so when the callback runs, the callback will tell you which http_client_t is calling you back.

http_event_t: The http_event_t is an enumerated datatype of the following events:

- HTTP_CONNECTED — Notification of successful connection including TLS
- HTTP_DISCONNECTED — Perform error recovery or cleanup
- HTTP_NOEVENT — Nothing
- HTTP_DATA_RECEIVED — Process the http_response_t

http_response_t: For every http_request_t, WICED automatically creates an http_response_t. The response is a structure:

```
typedef struct
{
    http_request_t* request;
    uint8_t*        response_hdr;                /* this contains HTTP response header including status line */
    uint16_t        response_hdr_length;         /* response_hdr_length is the length of HTTP header. */
    uint8_t*        payload;                     /* This contains Payload received in response */
    uint16_t        payload_data_length;         /* this length indicates only payload length */
    uint16_t        remaining_length;            /* Remaining data for this response. */
} http_response_t;
```

The WICED SDK provides you with a function called http_parse_header which will search through the HTTP header, which is an array of bytes, and will find all the headers that you tell it to look for and parse their values.

The structure has a pointer to the payload and the number of bytes. You are responsible for parsing (or whatever) that data. Don't forget the WICED cJSON library may help you.

All this data is freed when you call wiced_request_deinit. Typically, you will call wiced_request_deinit from the HTTP callback for the event HTTP_DATA_RECEIVED when the value of remaining_length in the response structure is equal to zero.

There is also a function http_client_deinit that you should call after the server disconnects. This function must NOT be done inside the HTTP callback, because you would be removing a thread that is currently running. Therefore, in the HTTP callback, when you get the HTTP_DISCONNECTED event you should set a flag and then call wiced_client_deinit from a different thread (like application_start) when the flag is set.

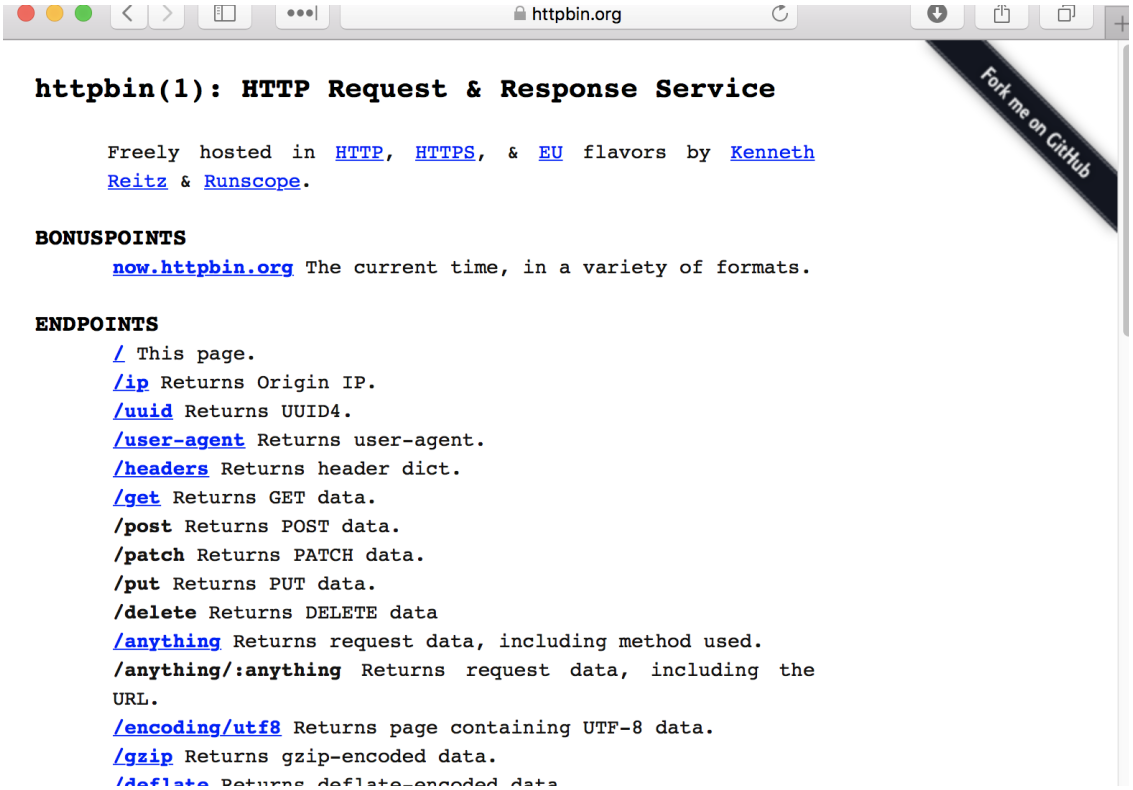
7C.6 Httpbin.org

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

You can build HTTP requests in CURL to test your ideas about how to interact with different HTTP endpoints. 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 including:



7C.7 Initial State

7C.7.1 Introduction

Initial State is a Web API based IoT analysis platform. In other words, they are setup to let you stream data from your IoT devices into their cloud. Then, you can log into their web platform and display and analyze your data with their extensive library of graphical web based tools.

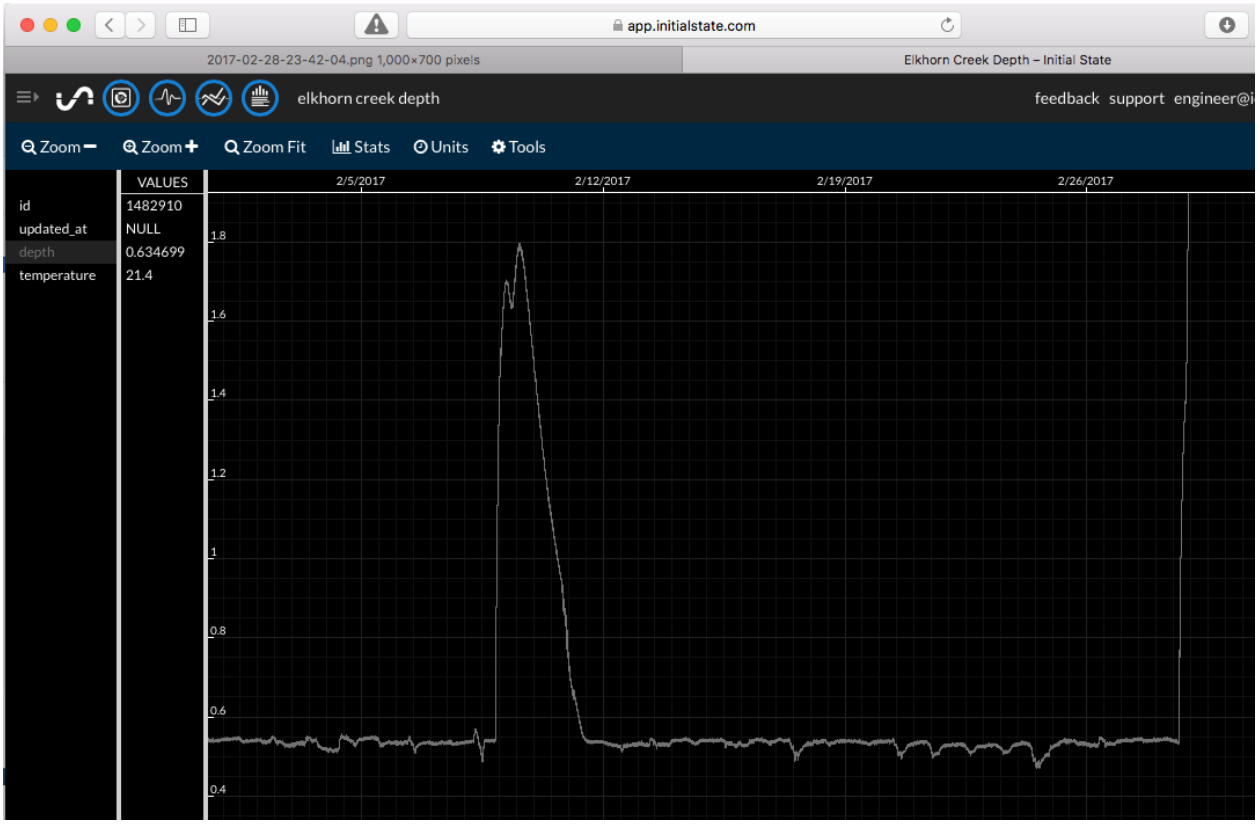
Data that you send to the Initial State cloud is organized into a Stream of time stamped key/value pairs. All Streams of data are grouped together into Buckets (which can hold one or more Streams). Each key/value data point that you send can have a time attached with it, or Initial State can automatically attach the timestamp of your upload to the data point.

The key in the key/value pair should be a textual description of what you are recording e.g. temperature, humidity, LED State, etc. The value can be a real number, a text string, an emoji of the form ":code:" e.g. ":smile:".

Data can be displayed using one or more "Tiles". Each tile can display a summary (such as "on" or "OFF"), a line graph of the values over time, a bar graph, etc. The figure below shows four value tiles and four line graph tiles. You can move Tiles around on the screen or resize them by right clicking on a Tile and selecting "Unlock Layout". You can zoom in on the time scale by clicking in the time bar along the top edge of the Tile screen and dragging the circles to the time range that you want to see.

There is also a Waves App, Line Graph App, and File Source Viewer that can be used to show data in different ways.

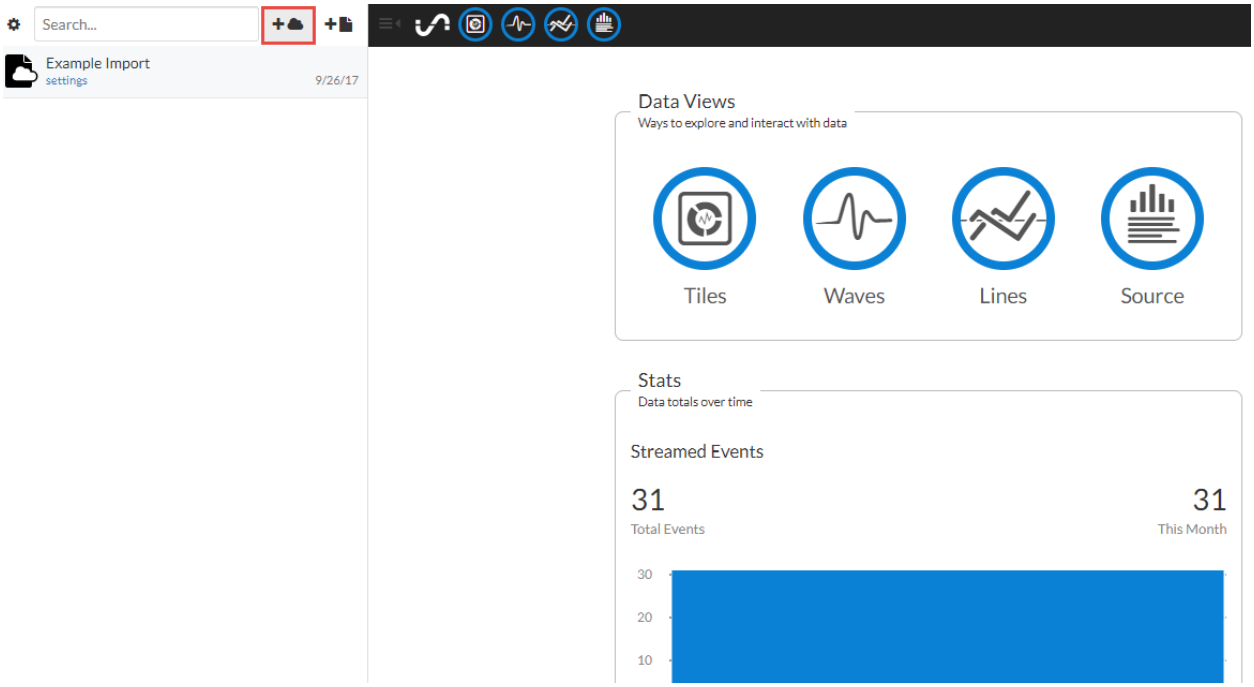
For example, if you lived in Kentucky near the Elkhorn Creek you could create a Bucket called “Elkhorn Creek” that has two Streams of data: one with the depth of the water in the creek and another with the temperature in the barn. You could then use the Line Graph App to display the water depth as shown in the figure below.



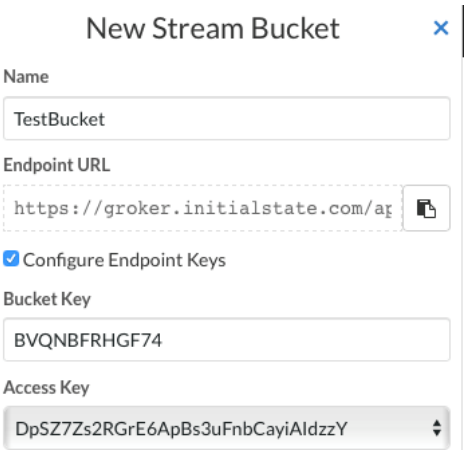
7C.7.2 Using Initial State

Setting up an Account, a Bucket, and a Tile

- Create a free account at www.initialstate.com.
- Create a new bucket by pressing the little "+" in the top left part of the screen next to the cloud icon.

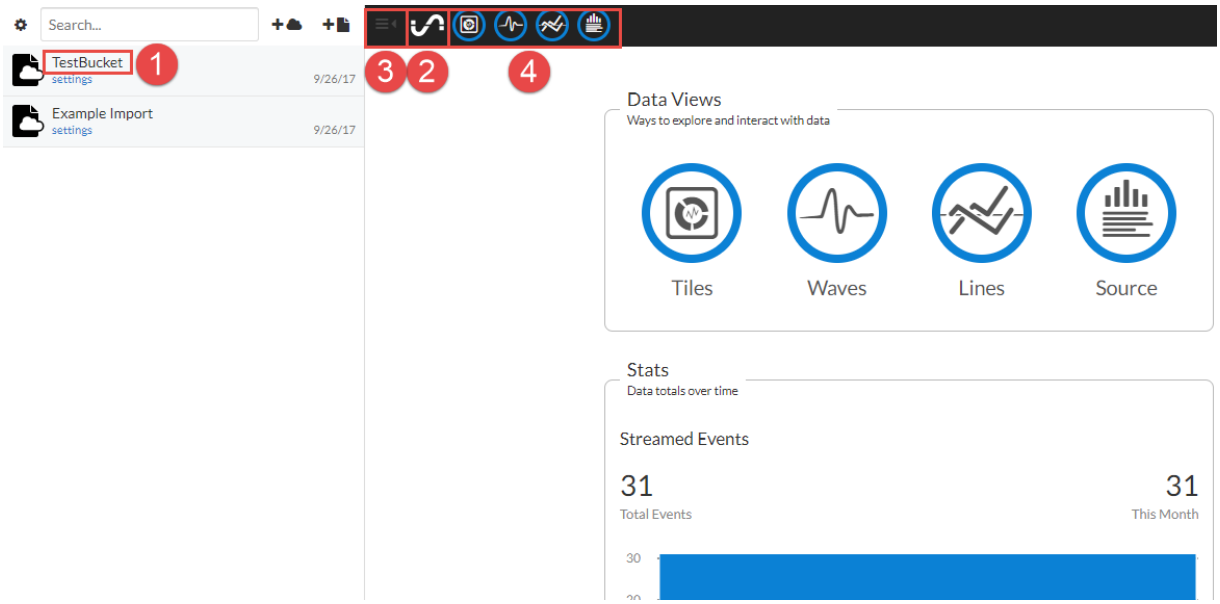


Enter a name for the bucket, check the *Configure Endpoint Keys* box to let the server create a Bucket Key and an Access Key, and click the *Create* button at the bottom of the window

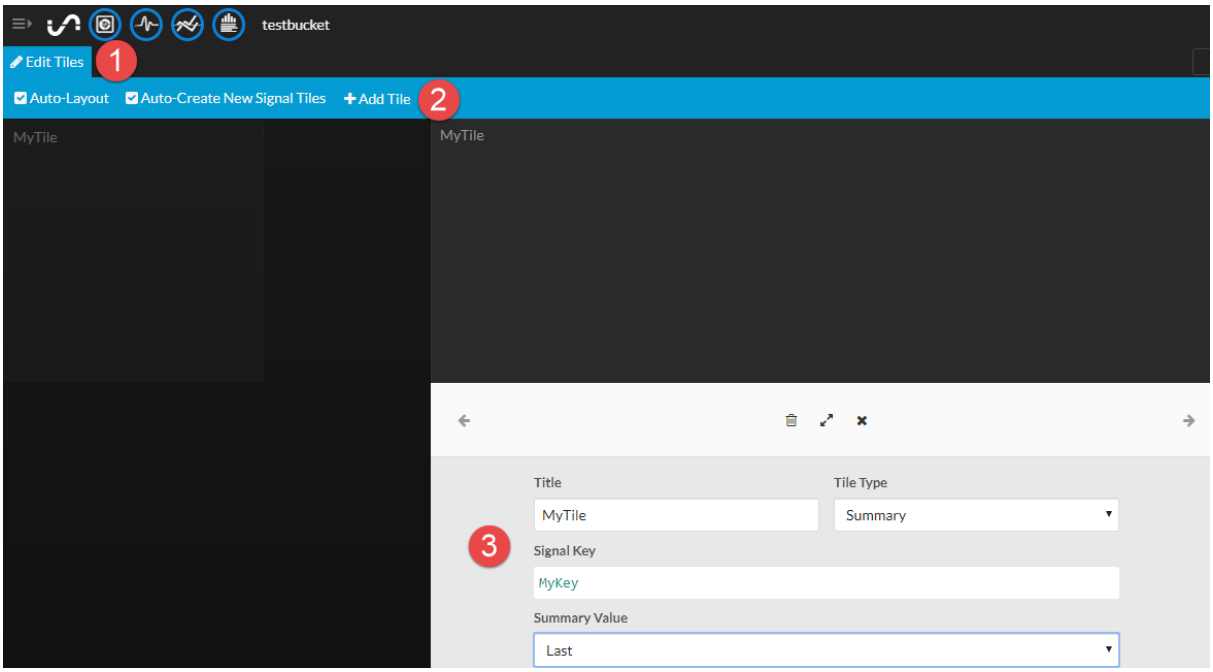


Once you have the Bucket setup, you should see the App Launcher as shown below. Click on the bucket name from the "shelf" window on the left (1) to make sure you are looking at the correct bucket and

then click the App Launcher button (2). Use the three lines with an arrow to open/close the shelf (3). You can also use the buttons along the top to switch to the Tile App, Waves App, Line Graph App, and File Source views (4).



To add a new tile, first select the Tiles App, click Edit Tiles (1), and then Add Tile (2). Enter the information that you want to display (3) and the tile will be created. Click outside the Add Tile window to close it. You will now see your tile, but it will be empty until you POST data to the bucket for the key that you created.



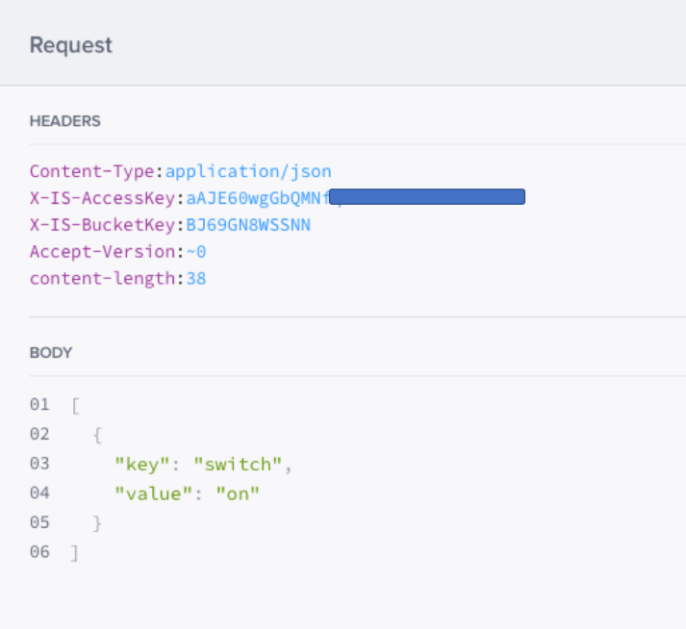
Sending HTTP Data

You can send HTTP POST requests to Initial State using one of two methods:

A JSON document

- Server = groker.intialstate.com
- Resource = /api/events
- HTTP Header for “X-IS-AccessKey: ”
- HTTP Header for “X-IS-BucketKey: ”
- HTTP Header for “Content-Type: application/json”
- JSON Document with an Array of Keymaps with keys of “key”, “value”, “epoch”, “iso8601”

For example, you could send an HTTP request that looks like this:



HTTP Options

- accessKey
- bucketKey
- eventKey0
- eventValue0

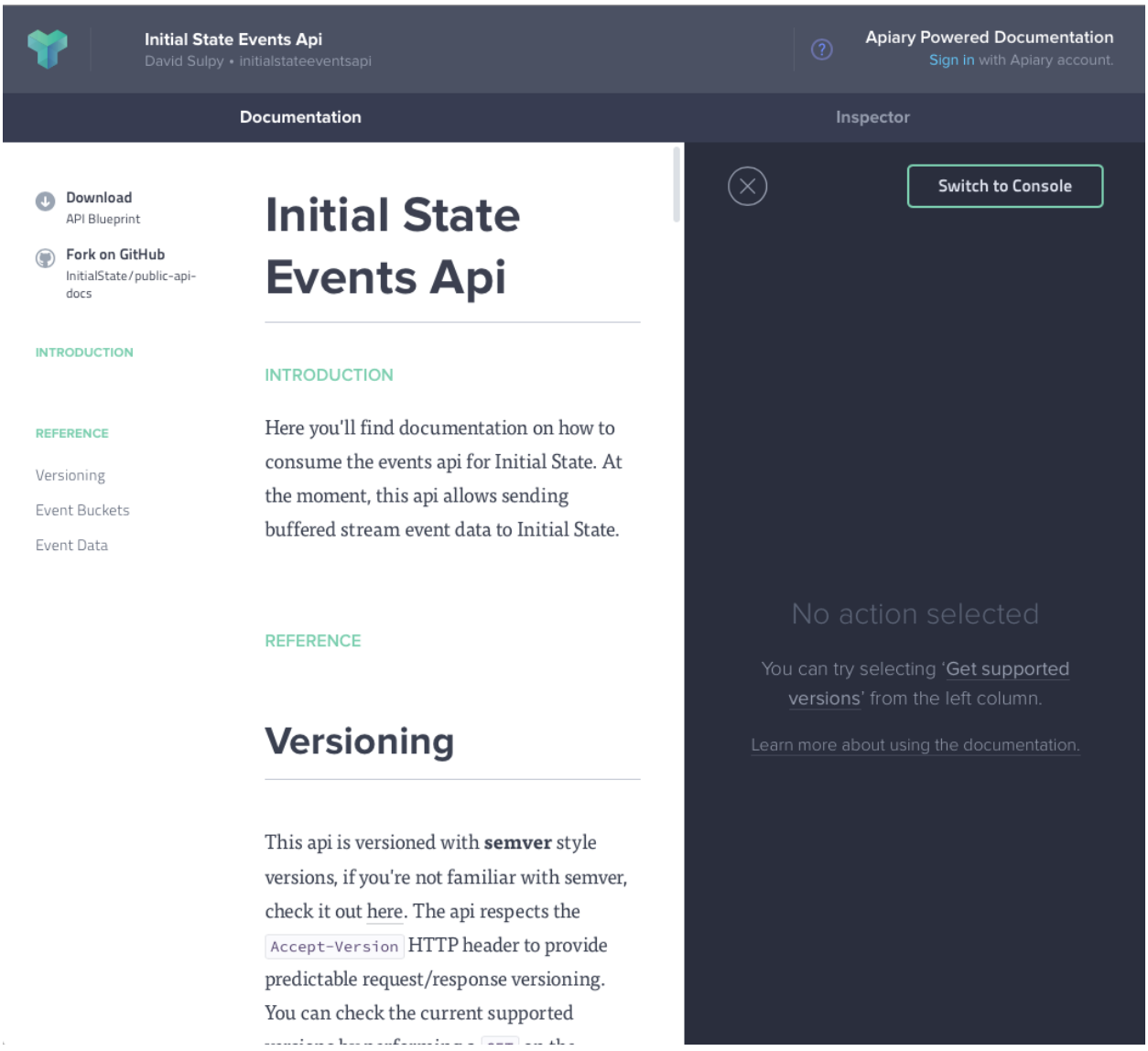
For example, you could send the event “switch on” by sending an HTTP POST request with the following settings:

- Server = https://groker.initialstate.com
- Resource = /api/events
- Options = accessKey=12345&bucketKey=987&eventKey0=switch&eventValue0=on

Using APIARY

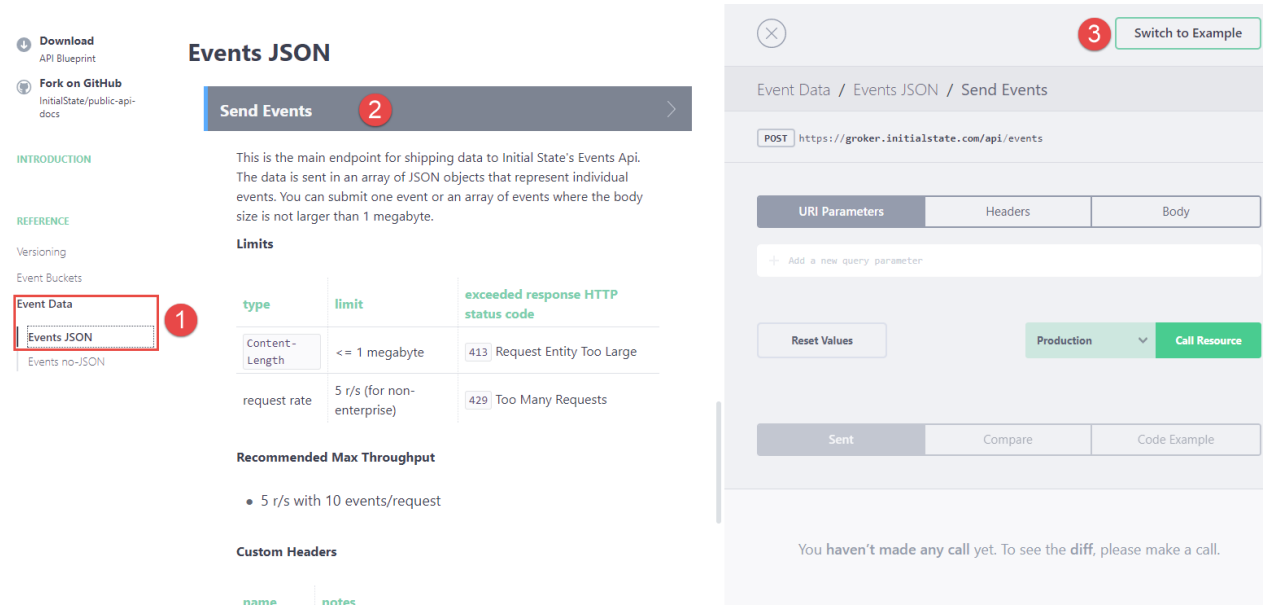
Initial State has documented their Web API with a tool called "APIARY". This is a web based tool which shows all the APIs and how to use them with examples. It can also switch to "console" mode where you can fill in the boxes in HTTP requests and it will send them to the Initial State Web Server.

You can access the APIARY documentation from InitialState by clicking on the link "View The Events API Docs" from the right side of the App Launcher. You can also access it by clicking "support" at the top right corner of the window, selecting "Streaming -> Using the Events API", and then clicking on "documented and testable on apiary". The initial APIARY window looks like this:



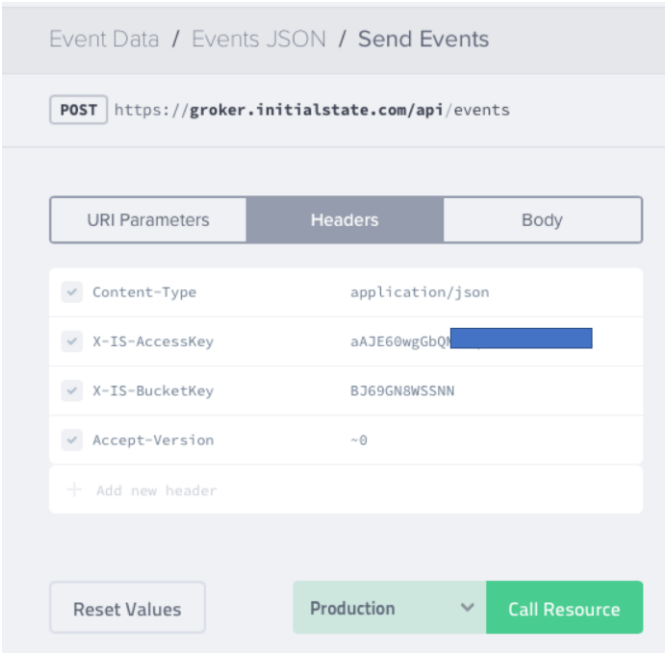
From the left panel, click on "Event Data -> Events JSON" and then click the banner "Send Events" from the center panel. Next, click the button that says "Switch to Console" on the right panel. Once you do

that, you will see the following window that includes documentation on sending events (center panel) and a test console that you can use to send POST requests (right panel).

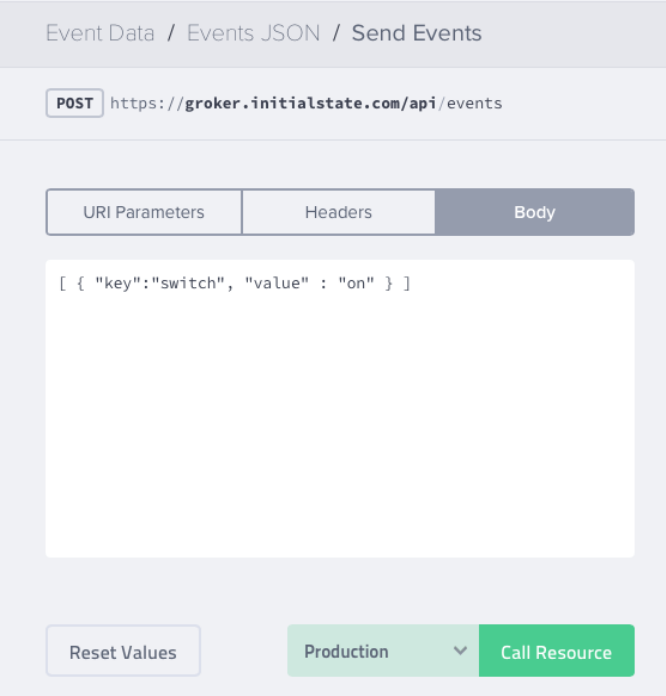


From the console, you can create and send HTTP requests. For example, to send the value “on” to a key called “switch” from the console, you would do the following:

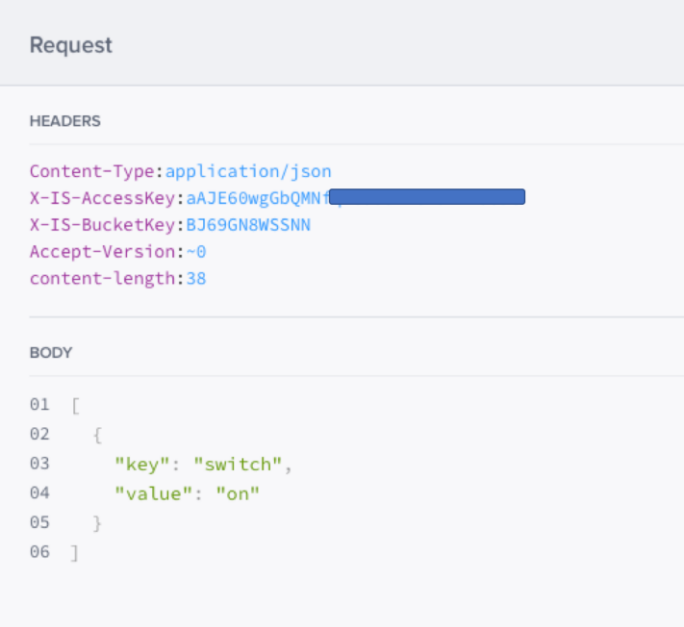
First click on “Header” and fill in your API Key and Bucket Key. The header for Content-Type is already filled in for you.



Then click on “Body” and type in the JSON document for the message.



When you press “Call Resource” it will show you the HTTP 1.1 document and the result.



If you have a Tile set up to monitor the state of the key “switch” you will see its value change to “on” once the message is received.

7C.8 Amazon Web Services IoT

In addition to MQTT, AWS supports a REST API interface to their cloud. The REST API Endpoint is:

https://<your_endpoint>:8443/things/<your_thing_name>/shadow

The connection must have a client verified connection (you need to provide your certificate and private key - hint: wiced_tls_init_identity). After you have a connection you can GET, POST and DELETE the document which is in JSON format.

Here is an example of a CURL connection to AWS:

```
CURL -v --cert 6fb5d874d6-certificate.pem --key 6fb5d874d6-private.pem --cacert rootca.cer -X GET
https://amk6m51qrxr2u.iot.us-east-1.amazonaws.com:8443/things/ww101_39/shadow
```

7C.9 Exercise(s)

Exercise - 7C.1 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.

Look at the website then:

- Use CURL to do a GET from the resource /get.
    - Hint: If your WiFi AP has a proxy server, CURL will not be able to connect (unless you specify the proxy server). You can connect WW101WPA AP, which doesn't require a proxy.
    - Hint: If you are using Windows, CURL is provided in class material. To use it:
        - Go to Software_tools/curl-<version>-win32-mingw/bin
        - Shift-Right-Click in the window and select either "Open command window here" or "Open PowerShell window here"
        - From the new window, run the command as *.\curl <other arguments>*.
            - The leading dot and slash are required because in some cases Windows aliases "curl" to a different function that is similar to CURL but is not the same.
- Use CURL to do a GET from the resource /html.
- Copy the snip.httpbin_org project to 07c/01_httpbin_org.
- Update the files as necessary and run it.
    - Hint: you will need to create a wifi_config_dct.h file and reference it from the make file.
    - Hint: you will see a JSON parse error message. This is an error in the example project – you can ignore it.
- Compare the output to what you got using CURL.
- Modify the project to perform a POST of a JSON string of your choice.
    - Hint: You can just create an array in the firmware containing a JSON string. Quotes inside the JSON string need to be escaped with a backslash.
    - Hint: httpbin.org has an endpoint called /post that will return the POST data.
    - Hint: You will need 3 headers – Host, Content-type, and Content-length.
    - Hint: Don't forget to write the content body with http_request_write.
    - Hint: Use can use CURL to try this out first. Use -H to specify "Content-Type: application/json" and use -d "<JSON>" (replace <JSON> with your JSON data).
    - Hint: If you want to put all the HTTP requests sequentially in a single project you should:
        - Deinit the request and set a semaphore in the callback when each request completes (i.e. when *response->remaining_length* equals 0).
        - Get the semaphore between requests.
        - The above guarantees that requests won't interfere with one another. If you don't do this, you may have the streams from multiple requests sending data over the same socket at the same time. Another alternative is to use a separate

HTTP client for each request which means you would have a separate socket for each one.

- Modify the project to perform a PUT of a JSON document of your choice.
  - Hint: httpbin.org has an endpoint called /put that will return the PUT data.
  - Hint: You will need 3 headers – Host, Content-type, and Content-length.
- Modify the project to perform an OPTIONS.
  - Hint: Use the endpoint "/" to see the options available for the top page.

Exercise - 7C.2 Initial State – Virtual LED Controlled by APIARY and CURL

- Go to www.initialstate.com and signup for an individual trial account.
- Create a new Bucket called "TestBucket". See the section earlier on Initial State for detailed instructions.
- Use the APIARY interface to write ON and OFF to the LED and see that the state changes in the Tile.
  - Hint: You will need to enter the AccessKey and BucketKey for you bucket either as headers or as URI Parameters. Placeholder headers are provided with default values. You can find the values for your bucket by going to the bucket settings on the shelf.
  - Hint: The body should be a JSON document that sends "key":"LED_State" with "value":"ON" or "value":"OFF".
  - Hint: The first time you write to a key from APIARY, it will create a summary tile for you automatically so you won't need to create it manually.
- What status code is returned when you send the message? Look up the code to see what it means.
- Use CURL to send HTTP POST messages to turn ON and OFF the virtual LED. Verify that it changes in the Tile.

Exercise - 7C.3 Initial State – LED State Controlled by Hardware

Create a WICED project to turn on and off the virtual LED on Initial State when the button on your AFE Shield is pressed.

- Hint: Start with the project that sends data to httpbin.org.
- Hint: You will need to get the certificate for initialstate.com or one of its CAs from a web browser and include it in the firmware (use one of the three methods discussed in the TCP/IP Sockets chapter.
- Hint: The HTTP Bin project has the server name in 2 places. Search for it and replace the 2nd one with SERVER_HOST.
- Hint: Don't forget to escape the quotes inside the JSON message with backslashes (\").
- Hint: Since the server can disconnect at any time, you should open the connection each time the button is pressed, send the request, and then close the connection. You can set a flag (or a semaphore) inside the callback function when *response->remaining_length* equals 0 to let you know when to close the connection.

Exercise - 7C.4 (Advanced) Initial State – Temperature & Humidity

Create a WICED project to write the temperature and humidity to Initial State each time you press the button.

Print the values to the terminal to compare with the values on Initial State.

- Hint: Refer to the I2C read project for reading the sensor values.

Exercise - 7C.5 (Advanced) Initial State – Graphing Temperature & Humidity

Create a WICED project that polls the temperature and humidity from the Shield once every second and sends them to initial state each time temperature changes more than 1 degree or humidity changes by more than 1%. Display the data on a graph on Initial State.

Exercise - 7C.6 (Advanced) Use Web APIs for Temperature Conversion

Create a WICED project that will read the temperature from the AFE shield, call the Web API to convert it from C to F, then display it on the LCD screen. We have setup an account with Neutrino API which will provide the conversion. The site and resource path for the conversion are:

https://neutrinoapi.com/convert

The options required are:

from-value=<the value to convert>
from-type=C
to-type=F
user-id=wicedwifi101
api-key=kyM2OWa22SZ1B5PGE7DvjSi67sPMXHTNXXENVut8JvmjkjMo

- Hint: Go to https://neutrinoapi.com/api/convert to see the documentation. Click on "Test API -> API Tools -> Convert" to try it out on the web.
- Hint: Use CURL to test the message and see the response that you will get back.
- Hint: You will need to get the certificate for neutrinoapi.com or one of its CAs from a web browser and include it in the firmware (use one of the three methods discussed in the TCP/IP Sockets chapter.
- Hint: When the response is received, you will need to parse the JSON from the body to get the converted temperature value. Refer to the JSON parser examples from the library chapter.
    - The value is returned as string, not a number.
- Hint: Refer to the Sensor Data project in the Library chapter for an example of reading the values from the shield and displaying them on the screen.

Exercise - 7C.7 (Advanced) Find and Use Twitter Web APIs

Create a WICED project to send a Tweet with the temperature each time the button on the board is pressed.

- Hint: Look in the Web API catalog or do a Google search to find the Twitter Web APIs.

Exercise - 7C.8 (Advanced) Example.com

Write a TCP socket program to send an HTTP request to example.com and print the resulting HTML to the debug UART. In this case, we will just use text strings to build up the request manually instead of using the HTTP library functions. Note how the HTTP communication is just text based – there is nothing fancy about the request/response.

You should:

- Open a Stream Socket to example.com port 80.
- snprintf an HTTP GET request including headers into a buffer. Don't forget the \n\r at the end of the headers.
- Send the buffer via TCP.
- Flush the TCP stream buffer.
- Read the TCP stream and print it onto the screen.

Hint: You may want to use Chapter 6, exercise 1 as a starting point.
Hint: Use CURL to try out the request to compare with what you get from your project.

7C.10 Related Example "Apps"

App Name	Function
http_sever_sent_events	starts, pings gateway, then starts AP
httpbin_org	Use HTTPS to get data from httpbin.org
https_client	Use HTTPS to get data from google HTTPS server and print it to the screen
http_server	WICED Station with an HTTP Server running

7C.11 Known Errata + Enhancements + Comments