Chapter 7C: Cloud Connectivity using HTTP 1.1

Time: 4 Hours

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 HTTP(s) to read & write data to the Cloud using RESTful APIs
* How to use CURL to test HTTP(s) servers
* How to create an HTTPS connections using TLS
* How to test your HTTP client using HTTPBIN
* How to use the IoT Cloud Provider InitialState.com

7C.1. Introduction 3

7C.2. HTTP 1.1 Protocol 3

7C.1.1. Client Request Message Format 4

7C.1.2. Server Response Message Format 4

7C.1.3. Client Request Start Line 4

7C.1.4. Server Response Start Line 4

7C.1.5. Client Request: HTTP Methods 5

7C.1.6. Client Request: Resources 6

7C.1.7. Client Request: Resource Options 6

7C.1.8. Headers 6

7C.1.9. Content Body 7

7C.3. Curl 8

7C.4. Representational State Transfer (REST) & RESTful APIs 13

7C.1.10. Web APIs 14

7C.5. WICED HTTP 1.1 Client Library 14

7C.6. Httpbin.org 15

7C.7. Initial State 15

7C.8. Amazon Web Services IoT 22

7C.9. Exercise(s) 22

Exercise - 7C.1 (Advanced) Example.com 22

Exercise - 7C.2 HTTP Bin 23

Exercise - 7C.3 RequestBin 23

Exercise - 7C.4 Use Webapis 23

Exercise - 7C.5 Initial State – Virtual LED 23

Exercise - 7C.6 Initial State – Real LED 23

Exercise - 7C.7 Initial State – Temperature & Humidity 23

Exercise - 7C.8 Initial State – Graphing Temperature 23

Exercise - 7C.9 Initial State – WebAPI 23

Exercise - 7C.10 AWS IoT 23

7C.10. Related Example “Apps” 24

7C.11. Known Errata + Enhancements + Comments 24

# 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 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 browser support it).

WICED supports both protocols, but with two separate libraries.

# HTTP 1.1 Protocol

HTTP 1.1 is an application layer, single transaction, stateless, plain-text, client-server protocol. This means that a client (your WICED device) opens a connection to a TCP Server (in the cloud), then sends an ASCII text request, the Server then responds, and the connection is closed. There is no memory in the protocol (there might be in 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 an optional body (same format for Client and Server)

## Client Request Message Format

A transaction starts with the client opening a TCP socket to the server (or a TLS TCP socket to the server). The client then sends three things:

* **Client Request Start Line**
* Optional **Headers** (one or more strings in the form of “headername: headervalue\r\n”)
* Optional **Content Body**, which is one optional content payload with as many bytes as required e.g. a file or an html page or a json document

## Server Response Message Format

The Server will then respond with:

* **Server Response Line**
* Optional **Headers** (same format as the client header)
* Optional **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.

## Client Request Start Line

The client request start line has four elements

* The HTTP Method
* The requested Resource path & optional arguments
* The version of HTTP (for this chapter it will always be “HTTP/1.1”
* A “\r\n”

An example legal client request line is

GET /ask HTTP/1.1\r\n

## Server Response Start Line

The HTTP Server response line will have 4 elements

* The protocol (probably “HTTP/1.1”)
* The [Status Code](https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml) (a number as defined by the IANA)
* 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

## Client Request: HTTP Methods

There are 9 [HTTP methods](https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages) 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/not-Idempotent)

Safe – the method doesn’t change anything on the Server and can be run without fear of side effects.

Idempotent – no matter how many time you run the method the state on the server state remains the same e.g. 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. An example of idempotent is a method that changes sometime on the server, but makes the same change no matter how many times you run it e.g. a DELETE. A non-idempotent method changes the state of the server every time you run it e.g. a POST which might insert data in the database every time it is run.

[GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET) (safe, idempotent) – The Server will reply with an HTTP message that with the Content of the requested Resource. The server will reply with Headers that will tell the Client how long the Content is “Content-length” and what is the MIME-Type of the Content “Content-type”.

[HEAD](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD) (safe, idempotent) – This Method performs the same operation as “GET” except it only replies with the Headers and does not return the Content.

[PUT](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/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](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PATCH) (unsafe,idempotent) – With this method Client is requesting a partial PUT e.g. If the Resource is a document that contains name, age then the Client could PATCH a new age by having content with the updated age by sending a JSON document with “{age:49}”

[POST](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/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 an 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](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/DELETE) (unsafe, idempotent) – This Methods asks the Server to remove the Resource

[OPTIONS](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/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.

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](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/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.

## Client Request: Resources

When you look at an http web address (sometime known as a 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 this is “protocol://serverName/path?option”.

In HTTP/1.1 the 2nd element of the Client Request line is the Resource & Options. 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.

Another request might look like

GET /resource?format=simple HTTP/1.1

Which is a request to the server to please send the document located at “/resource” as an HTTP response in the format of “simple” (what “simple” means is part of the application semantics)

## Client Request: Resource Options

Resource options are appended to the resource location by placing a “?” at the end of the resource. 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”

## Headers

The [HTTP Headers](https://developer.mozilla.org/en-US/docs/Web/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 meta data between the client and server. The meta data may include, what type of file is 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. Here are a few example Header lines

Content-type: application/json

Content-type: text/plain

Content-type: application/xml

Content-length: 129

Accept: application/json

Accept-Language: en-US, de

X-Some-Header: 1239asdf

Set-cookie: nsatrack=129

The IANA has a [standard list](https://www.iana.org/assignments/message-headers/message-headers.xhtml) 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”.

There are two special headers for specifying the length and type of the content payload, “Content-type” and “Content-length”

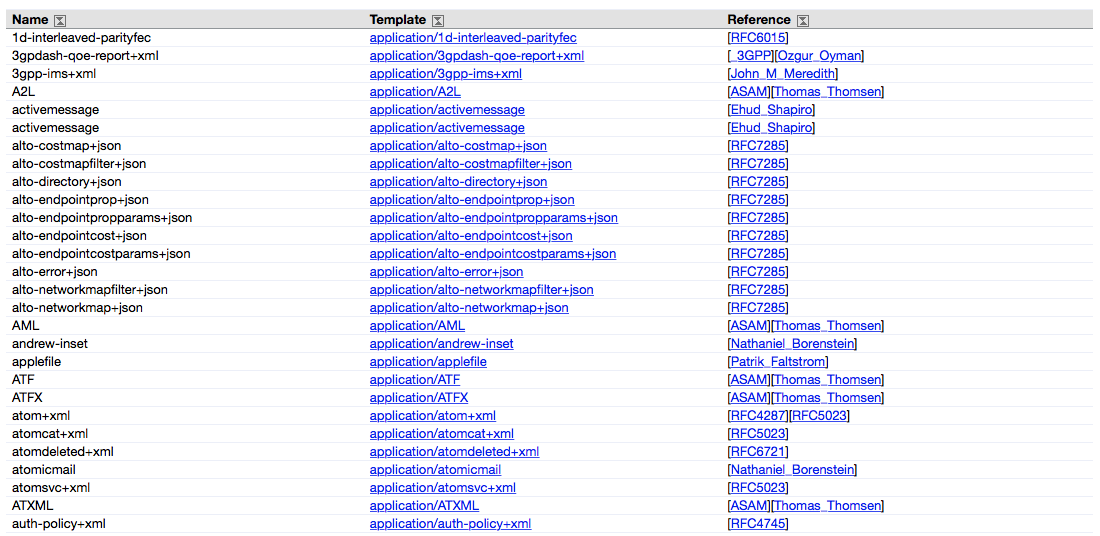
## Content Body

The optional body of the message is either sent (by the client or server). It is just a string of bytes that starts right after the “\r\n”. 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](https://www.iana.org/) and can be found on their [website](https://www.iana.org/assignments/media-types/media-types.xhtml). 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



# Curl

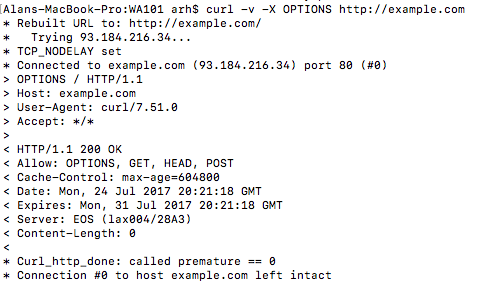
Curl (Client for URLs or See URL) is a Unix (Linux, MacOS, Windows w/cygwin) utility for sending and receiving HTTP requests. Curl is a handy tool to help you figure out what an HTTP website is doing so that you can build your WICED program to do the same thing. Curl will let you create HTTP requests 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

HOST somewebsite.com

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)



Curl support 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.

Some of the useful 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 |
| -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 |
| -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 |
| -d “data”  --databinary “data” | Specifies the data for a PUT, POST. CURL will automatically add the “Content-length:” header and “Content-Type: application/x-www-form-urlencoded” header. |
| curl -v -X "PUT" -H "content-type: application/json" -d "{asdf}" http://httpbin.org/put |
| -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 |
| --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 |
| --cookie “value” | This will add the header “Cookie: value” to your header |
| curl –v --cookie “name=arh” <http://httpbin.org/get> |
| --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.

# Representational State Transfer ([REST](https://en.wikipedia.org/wiki/Representational_state_transfer)) & RESTful APIs

REST is a design philosophy what developed by Thomas Fielding for his [PhD Dissertation](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). 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 Restful HTTP Server using the principals outline 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 is a noun e.g.
  + /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
  + 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 e.g.

* Filtering /companies/cy/products?type=wifi
* Pagination /companies?page=27
* Searching /companies?search=Cypress
* Sorting /companies?sort=rank\_asc

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”

## 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 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>

# WICED HTTP 1.1 Client Library

The WICED SDK several built-in HTTP libraries including protocols/HTTP\_Client 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 header (the blank line “/r/n”) http\_request\_write\_end\_header
* Optionally write the document http\_request\_write
* Flush the writes http\_request\_flush

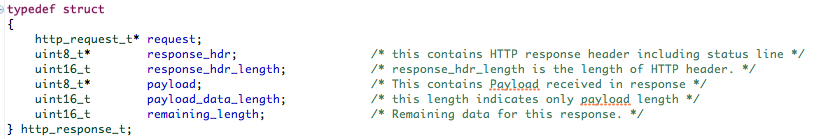
Then you wait for the callback. In the callback function (which your registered when you created the client) you will get the arguments, http\_client\_t \*, http\_event\_t and http\_response\_t.

**http\_client\_t**: The same callback function can be used to process requests from multiple http\_client\_t, 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 there WICED automatically creates a http\_response\_t. The response is a structure:



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 the 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.

# Httpbin.org

# Initial State

[Initial State](http://www.initialstate.com/) 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.

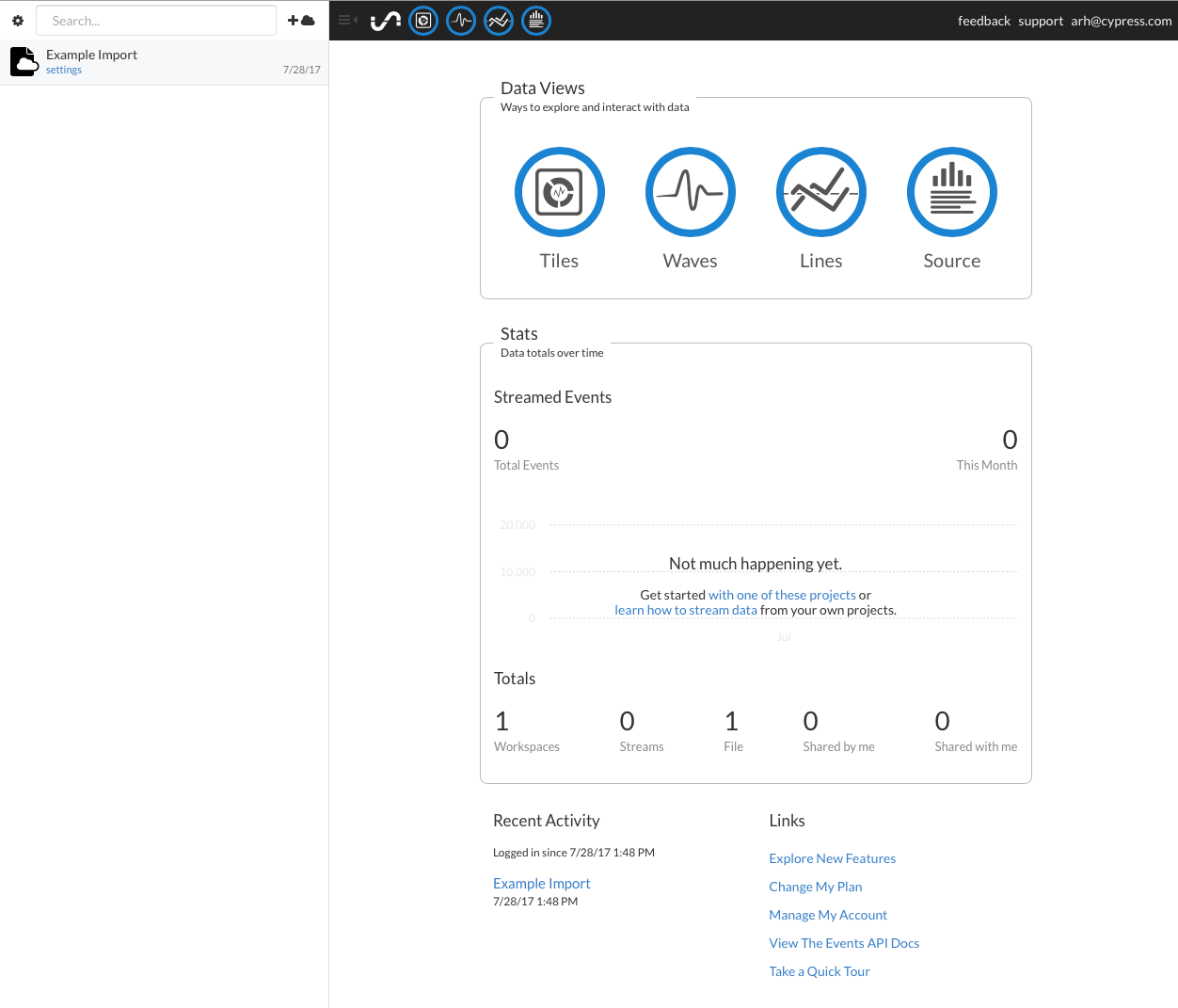
The key in the key/value pair should be a textual description of what you are recording e.g. temperature, humidity, LED State, etc. The value can be a real number, a text string, an [emoji](https://emoji.codes/) of the form “:code:” e.g. “:smile:”.

For example, if you lived in Kentucky near the Elkhorn Creek could create a Bucket called “Elkhorn Creek” that has two Streams of data, one, with the depth of the water in the creek and two, the temperature in the barn.

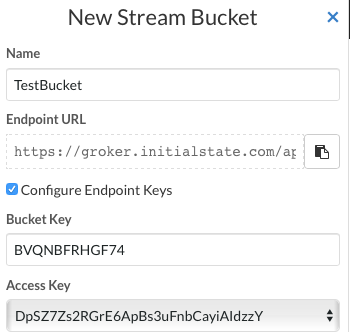
Picture

To use Initial State, you need to

* Create a free account at [www.initialstate.com](http://www.initialstate.com)
* Create a new bucket by pressing the little “+” in the top left part of the screen



Type the name of the bucket and let the server create a Bucket Key and an Access Key.

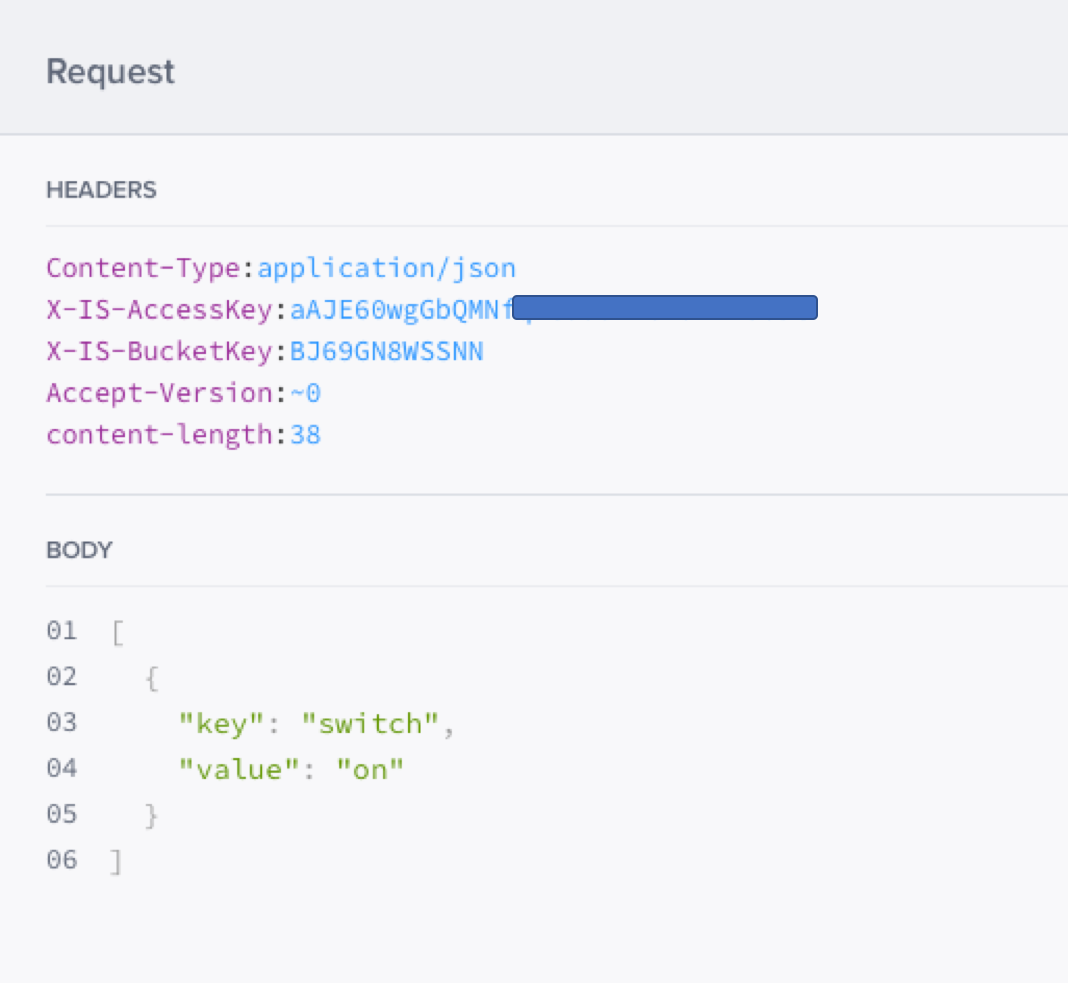


Once you have the Bucket setup, you can send HTTP Post requests to Initial State using one of two methods

A JSON document

* Server = groker.intialstate.com
* HTTP Header for “X-IS-AccessKey:”
* HTTP Header for “X-IS-BucketKet:”
* JSON Document with an Array of Keymaps with keys of “key”, “value”, “epoch”, “iso8601”

For example, an HTTP document that looks like this:



HTTP Options

* accessKey
* bucketKey
* eventKey0
* eventValue0

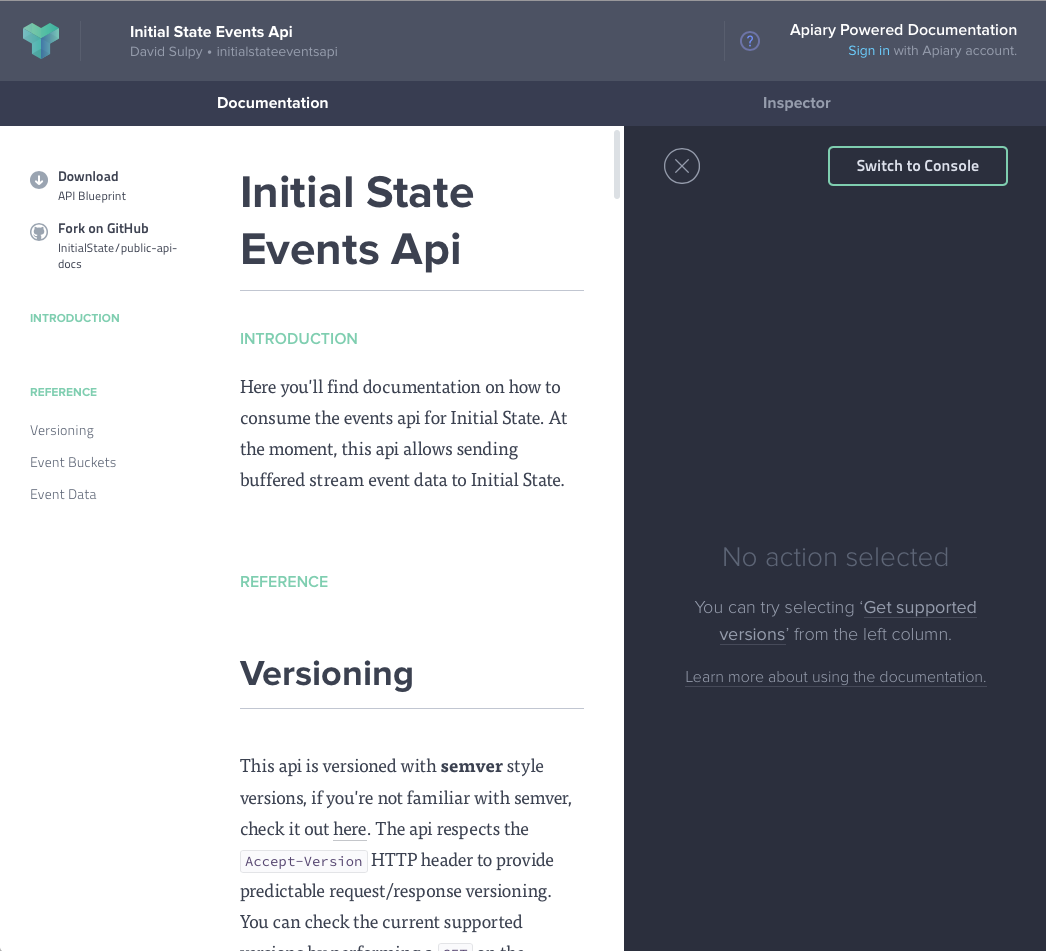
For example, you could send the event “switch on” by sending an HTTP Post request with the following settings:

Server = https://groker.initialstate.com

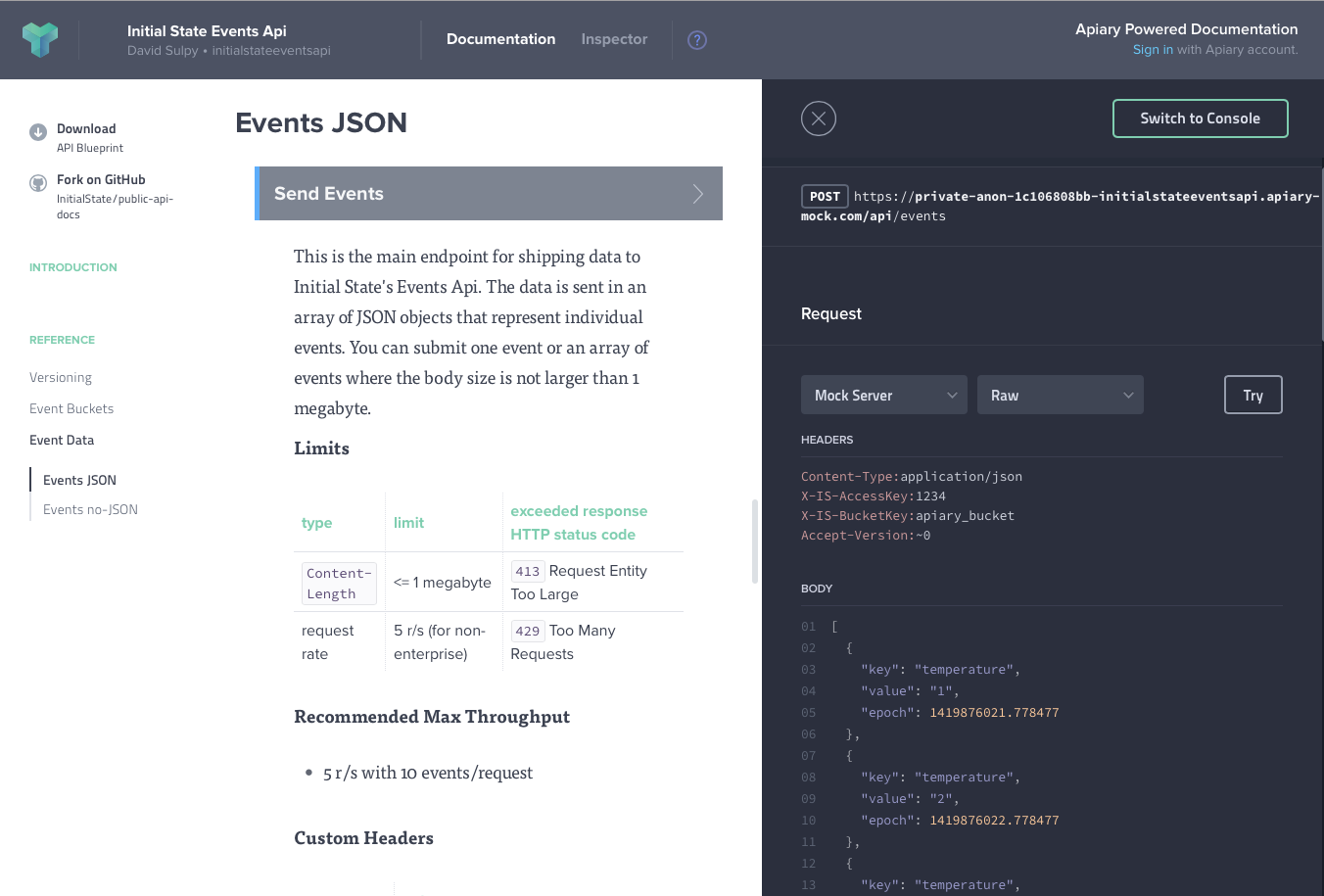
Resource = /api/events

Options=accessKey=12345&bucketKey=987&eventKey0=switch&eventValue0=on

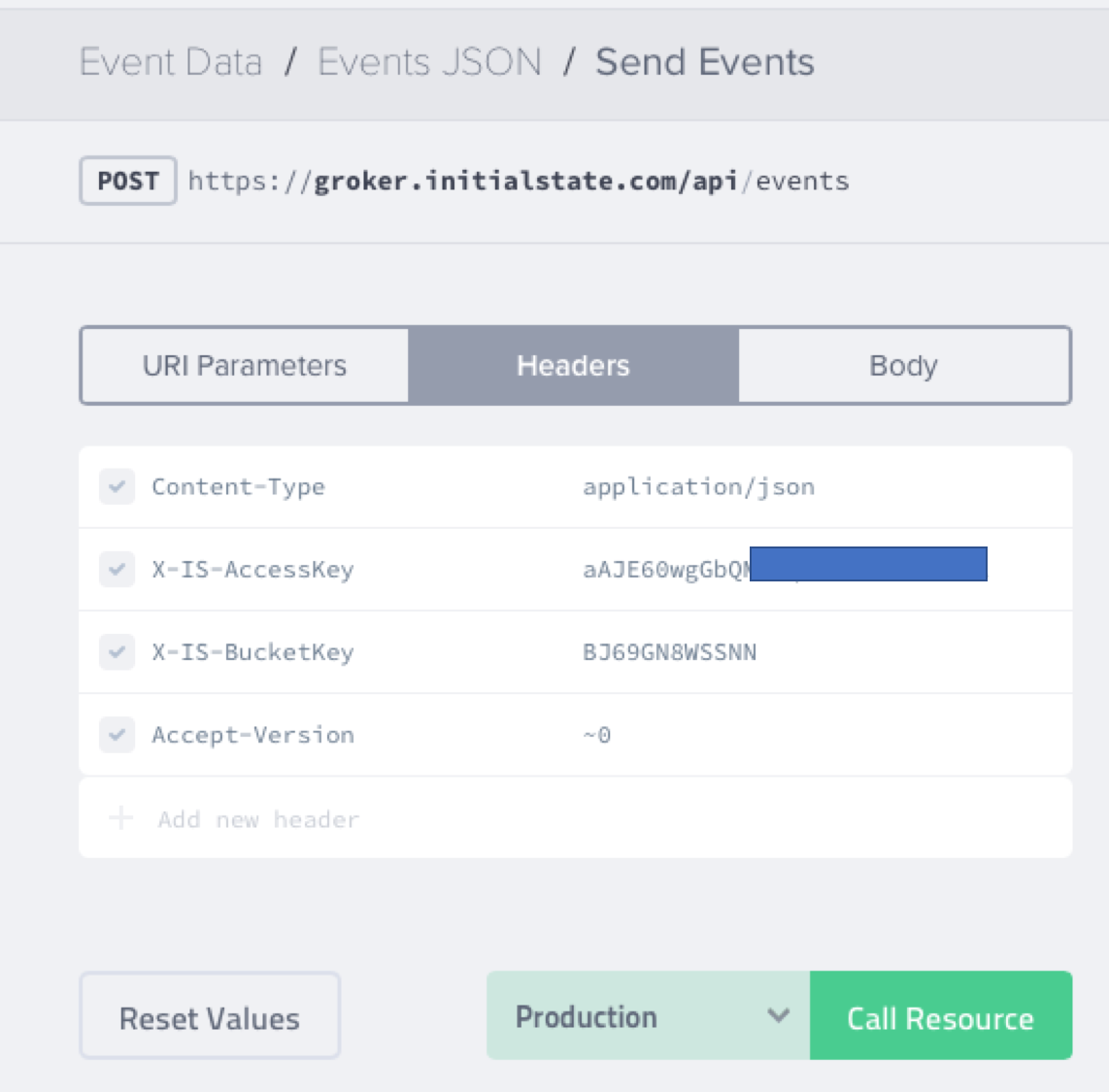
Initial State has documented their Web API with a tool called “[APIARY](http://docs.initialstateeventsapi.apiary.io/)”. This is a web based tool which shows all the APIs and how to use them with examples. It can also switch to “console” mode where you can fill in the boxes in HTTP requests and it will send them to the Initial State Web Server.



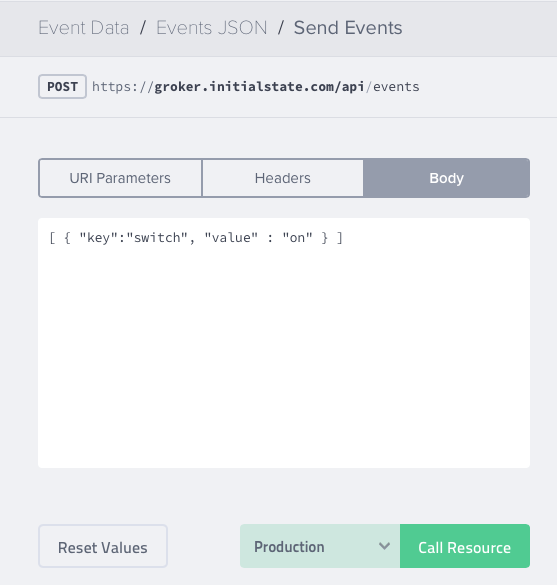
Documentation for sending events:



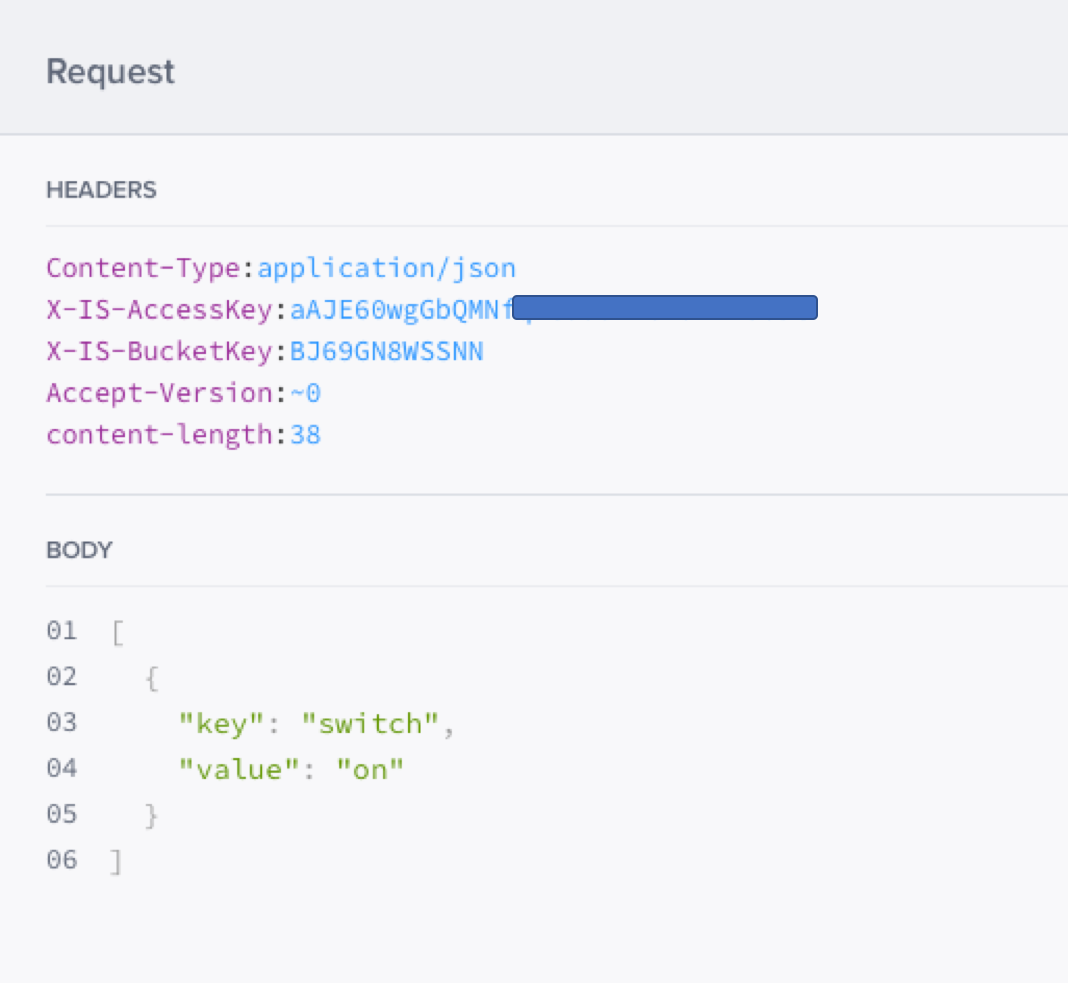
If you switch to the console you can create and send requests, for example sending the event “switch on” could be done by first clicking on the “Switch to Console”. Then clicking “Header” and filling in your API Key and Bucket Key.



Then clicking on “Body” and typing in the JSON document for the message.



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



# Amazon Web Services IoT

Chapter 7B talks about how to use the Amazon Web Services IoT Cloud using MQTT. AWS also supports a [REST API](http://docs.aws.amazon.com/iot/latest/developerguide/thing-shadow-rest-api.html) 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:

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

# Exercise(s)

* 1. (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. You should

* Open a Stream Socket to example.com port 80
* Sprint your HTTP request including headers into a buffer
* Send that buffer via TCP
* Flush the TCP stream buffer
* Read the TCP stream and print it onto the screen
  1. HTTP Bin

The Website httbin.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:

* Run the snip.httpbin\_org
* Modify it to perform a POST of a JSON document of your choice
* Modify it to perform a PUT of a JSON document of your choice
* Modify it to perform a HEAD
* Modify it to perform an OPTIONS
  1. RequestBin
  2. Use Webapis

Make a program 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. The following RESTful Web API will provide this functionality

<http://webservices.daehosting.com/services/TemperatureConversions.wso?op=CelsiusToFahrenheit>

Make a program to send a Tweet with the temperature each time the button on the board is pressed

* 1. Initial State – Virtual LED

Use CURL to turn on and off a virtual LED on Initial State

* 1. Initial State – Real LED

Make a program to turn on and off an “LED” on Initial State when the button on your AFE Shield is pressed.

* 1. Initial State – Temperature & Humidity

Make a program to write the temperature and humidity to Initial State each time you press the button

* 1. Initial State – Graphing Temperature

Make a program to poll the temperature from the AFE Shield and send it to initial state each time it changes more than 1 degree. Display the data on a graph.

* 1. Initial State – WebAPI

Make a program to write the temperature in F to initial state (use the WEB API to convert it)

* 1. AWS IoT

Write an HTTP\_Client program that can

* Read and display the weather data from the AWS IoT “thing” WW101\_01.
* Read the local weather data from the AFE Shield and update your AWS “thing”
* Use the buttons on the shield (either Capsense or mechanical) to switch between the two screen (local data and WW101\_01 data)
* Read the actual data from WUNDERGROUND and display it as a third screen.

# 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 |

# Known Errata + Enhancements + Comments