Contents

[The HTTP Request/Response Life Cycle 2](#_Toc73127526)

[Step 1: Local Processing 2](#_Toc73127527)

[Step 2: Resolve an IP 2](#_Toc73127528)

[Step 3: Establish a TCP Connection 3](#_Toc73127529)

[Step 4: Send an HTTP Request 3](#_Toc73127530)

[Step 5: Tearing Down and Cleaning Up 5](#_Toc73127531)

[References 6](#_Toc73127532)

# The HTTP Request/Response Life Cycle

## Step 1: Local Processing

Depending on how in depth you want to get, much can happen during this step depending on the application making the request. I am going to proceed on the understanding this request is being made by a browser, as opposed to cURL, an API client like Postman, or some other app.

1. Your browser extracts the "scheme"/protocol (we have established that this will be HTTP), host ([www.example.com](http://www.example.com)), and optional port number, resource path, and query strings that are specified in the form:  
   <protocol>://<host><:optional port>/<path/to/resource><?query>.

An example is:

|http|://|www.example.com||:5000||/mainpage||?query=param&query2=param2|

1. Now that the browser has the intended hostname for the request, it needs to resolve an IP address. The browser will then look through its own cache of recently requested URLs, the operating system’s cache of recent queries, your router’s cache, and your DNS cache.

## Step 2: Resolve an IP

Like the processing done locally, resolving an IP from a "DNS server" is a sequence that includes many steps, and includes failovers if the first request fails to return an address.

1. If the cache lookup fails (we will assume it does), your browser fires off a *DNS request* using *UDP*. The DNS request contains the preconfigured IP for your DNS server and your return IP in its header. The hostname for which you are trying to resolve an IP is in the request’s "Question" section. UDP is a lightweight protocol, but the trade-off is that it offers no guarantees in terms of delivery, and there is no acknowledgement other than a response being sent and received.
2. Your request will now have to travel many network devices to reach its target DNS server. Whenever the packet hits a piece of networking equipment, the device uses a routing table to determine which other device it is connected to that is most likely situated along the shortest path to the destination.
3. Once your request arrives at your configured DNS server, the server looks for the address associated with the requested hostname. If it finds one, it sends a response. If, on the other hand, the DNS server you have targeted cannot locate the given hostname, it passes the request along to another DNS server it is configured to defer to. This happens recursively until the address is found, or an *"authoritative" nameserver* is hit. If an address for the given domain cannot be resolved, the server responds with a failure and your browser returns an error.
4. We will assume the request is successful though, given that all of this is still a precursor. If the response makes it back (remember, with UDP there’s no guarantee!), the requesting client now has a target IP. It will also have received a piece of information as part of the answer that will let it know how long the returned answer can be cached for. This means subsequent requests will get to take a shortcut from **Step 1.2** to here.

## Step 3: Establish a TCP Connection

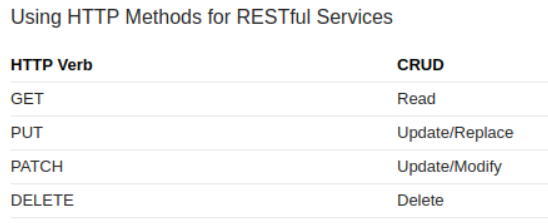
Now that the client has an IP address, it can send an HTTP request, right? Almost, but first, since the request is sent over *TCP (Transmission Control Protocol)*, which is a transport layer protocol like UDP, the client must open a TCP connection.

1. One of the key differences between TCP and UDP is that TCP ensures delivery and ordered data transmission. Much of this is done very simply, using what’s known as a sequence number for every byte sent. This allows the receiver to re-order received packets back into their original order, and allows the sender to retransmit any packet that does not get acknowledged by the receiver.
2. These guarantees and more can be found on Wikipedia, and are worth reading about, but what’s most relevant is that TCP connections are opened using what’s known as a three-way handshake. The server must already be "listening" on a port, performing a passive open, after which the client can initiate an active open, and the handshake works as follows:
   1. The client initiates the active open by sending a SYN (*synchronize*) "control" packet to the server. The client sets the sequence number for the first packet to a random value purposely, in service of security. We’ll refer to this number as x for now.
   2. The server responds with a SYN-ACK message, which contains an acknowledgement number for the original message that is always x+1, and a new sequence number for the response itself, which is another random number y.
   3. In the third step, the client sends an ACK message back to the server with a sequence number equal to x+1, which should match the SYN-ACK message’s acknowledgment number and ensure that our data is being delivered reliably. The ACK message’s acknowledgment number (since it is acknowledging the SYN-ACK) is set to one more than the received sequence number, or y+1.
3. We now have a completed three-way handshake and an established connection where both the client and server have received acknowledgment of the connection from the other party. The connection has also established a random, sequential sequence for each direction of communication (client->server, server->client), allowing for bidirectional, concurrent communication along the connection, which is also known as **full duplex communication.**

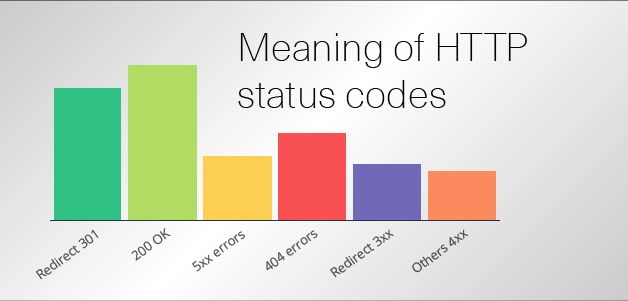
## Step 4: Send an HTTP Request

Wow, that was a bunch of steps! But now that the client has an IP address *and* a TCP connection, it can finally send an *HTTP* request! Except…..no I’m kidding, we can send a request for real this time!

1. The request is made up of a "request line", request header, and a body. The "request line" is simply a line that indicates the HTTP **method**, the resource being requested, and the protocol version. The header of the request is made up of pairs in the form name:value <CR><LF>. Two consecutive <CR><LF> pairs indicate the end of the header section. The only mandatory field in an HTTP request is HOST, which contains the domain and port that the request is being sent to (domain.com:8080), although in some cases the port can be omitted. Outside of the host field, common standard HTTP header fields include Origin, Accept, Accept-Encoding, and many more. The request can also include any non-standard header fields, and historically non-standard fields are indicated by prefixing X- to the field’s name. The body content of an HTTP request is completely optional, but often contains something like form data or JSON.



1. Once the HTTP request is sent, it follows a similar routing procedure as the one discussed earlier, with the difference being that using TCPs magic sequence number powers, the server can ensure it receives the whole request, in the correct order.
2. Once the server receives the request, processes it, and finds the resource being requested, it generates an HTTP response. An HTTP response has a similar structure to an HTTP request, containing a "status line", response header fields, and an optional body. The status line contains an *HTTP status code* indicating the success, failure, or error-state of the request along with a "reason message" that provides detail.



Everything from status code 200 to 299 indicates a successful request. 200 is used most commonly, and it simply means OK. Code 300 to 399 tell the browser to redirect to another page. For instance, 301 means Moved Permanently, and you often use it to tell the crawl of a search engine to replace an indexed page by a new one.

Anything from 400 to 499 indicates a problem on the client. Like some kind of bad request indicated with 400. Insufficient privileges like with 403: Forbidden or a non-existent URL indicated by 404: File Not Found. And then there are the errors from 500 to 599. They indicate a server problem. Most commonly you will see 500, which means Internal Server Error.

1. Once the response is generated, the server responds to the request. At the TCP layer, the client receives the first data packet, the first byte of which should contain the HTTP response header. More packets start coming in, and at the TCP layer they are re-ordered as needed. For every two packets that the client receives at the TCP layer, it sends an ACK message to the server. This goes on until the response is (hopefully) fully loaded.

## Step 5: Tearing Down and Cleaning Up

We’re almost there!

1. Once the response has been fully delivered, the client sends a FIN packet at the TCP level, to which the server responds with an ACK, and then generally sends a FIN of its own, which the client responds to with its own ACK signal. The client then waits for a brief timeout, during which it cannot accept new connections, to prevent delayed packets from previous connections arriving during subsequent activities on the port. This **four way handshake** signals the end of the TCP connection.
2. At this point, your browser begins processing what it has received. If it is an image, data, or other media file that is being consumed by some application inside the browser, a variety of things can happen. If the data received is HTML, the browser will start parsing the HTML, and rendering the page you requested. As it parses, the browser may come across links to images or other media that are external to the HTML it has received, and will spin up new requests for that content, restarting this whole process (although usually skipping steps 1 & 2 thanks to caching). But, given that we were only interested in the lifecycle of an individual request: our (application’s) work is done, congratulations!

# References

<https://dev.to/dangolant/things-i-brushed-up-on-this-week-the-http-request-lifecycle->

<https://medium.com/@officialrahulmandal/the-lifecycle-of-an-http-request-f6b1c1635312>

<https://www.exceptionnotfound.net/the-asp-net-web-api-2-http-message-lifecycle-in-43-easy-steps-2/>