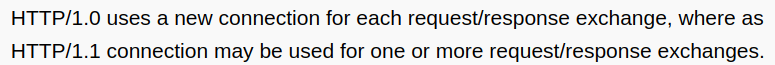
**HTTP**

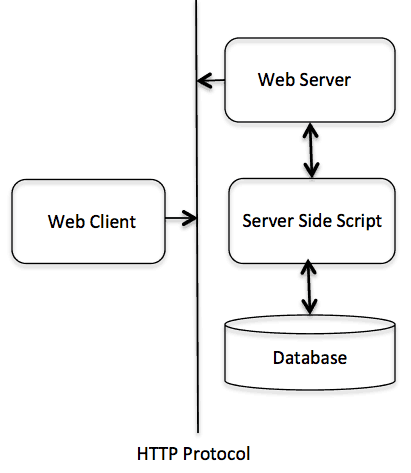
The **Hypertext Transfer Protocol** (HTTP) is an **application-level protocol** for distributed, collaborative, hypermedia information systems. This is the foundation for data communication for the World Wide Web (i.e. internet) since 1990. HTTP is a generic and stateless protocol which can be used for other purposes as well using extensions of its request methods, error codes, and headers.

**HTTP** is a **TCP/IP based** communication protocol, that is used to deliver data (HTML files, image files, query results, etc.) on the World Wide Web. **The default port is TCP 80**, but other ports can be used as well. It provides a standardized way for computers to communicate with each other. HTTP specification specifies how clients' request data will be constructed and sent to the server, and how the servers respond to these requests.

There are three basic features that make HTTP a simple but powerful protocol:

* **HTTP is connectionless:** The HTTP client, i.e., a browser initiates an HTTP request and after a request is made, the client disconnects from the server and waits for a response. The server processes the request and re-establishes the connection with the client to send a response back. Keep-alive ??? Burda sanırım demek istenen: keep-alive olmadığı zaman requesti gönderdikten sonra bağlantı kesiliyor, response geleceğinde yeniden bir bağlantı/connection açılıyor.
* **HTTP is media independent:** It means, **any type of data can be sent by HTTP** as long as both the client and the server know how to handle the data content. It is required for the client as well as the server to specify the content type using appropriate MIME-type.
* **HTTP is stateless:** As mentioned above, HTTP is connectionless and it is a direct result of HTTP being a stateless protocol. The server and client are aware of each other only during a current request. Afterwards, both of them forget about each other. Due to this nature of the protocol, neither the client nor the browser can retain information between different requests across the web pages.



**Basic Architecture**

The HTTP protocol is a **request/response protocol** based on the client/server based architecture where web browsers, robots and search engines, etc. act like HTTP clients, and the Web server acts as a server.

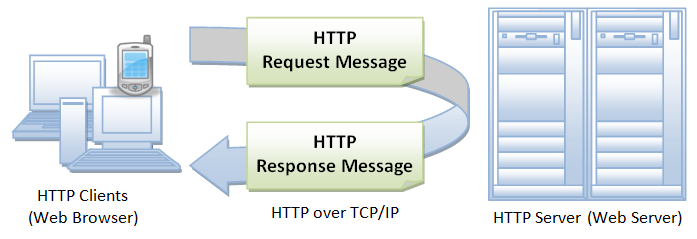
**Client**

The HTTP client sends a request to the server in the form of a **request method**, **URI**, and **protocol version**, followed by a MIME-like message containing request modifiers, client information, and possible body content **over a TCP/IP connection**.

**Server**

The HTTP server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta information, and possible entity-body content.

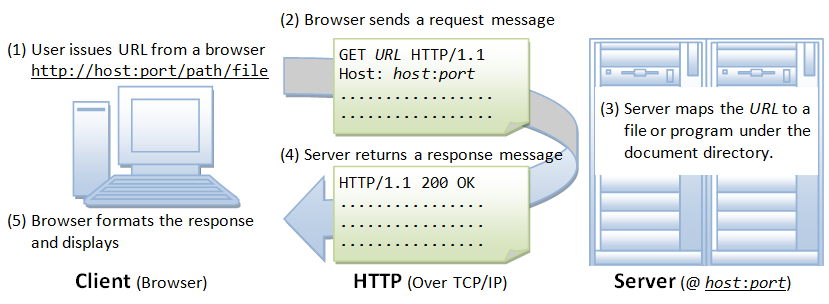
HTTP is an **asymmetric request-response client-server protocol** as illustrated. An HTTP client sends a request message to an HTTP server. The server, in turn, returns a response message. In other words, HTTP is a pull protocol, the client pulls information from the server (instead of server pushes information down to the client).



* HTTP is a stateless protocol. In other words, the current request does not know what has been done in the previous requests.
* HTTP permits negotiating of data type and representation, so as to allow systems to be built independently of the data being transferred.

**Browser**

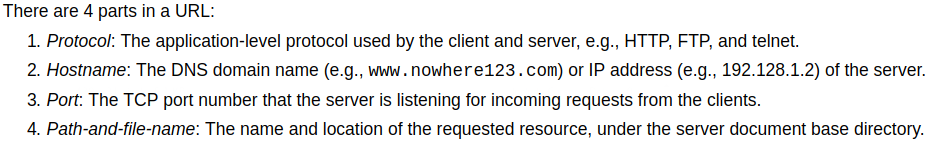
Whenever you issue a URL from your browser to get a web resource using HTTP, e.g. http://www.nowhere123.com/index.html, the browser turns the URL into a request message and sends it to the HTTP server. The HTTP server interprets the request message, and returns you an appropriate response message, which is either the resource you requested or an error message. This process is illustrated below:



**Uniform Resource Locator (URL)**

A URL (Uniform Resource Locator) is used to uniquely identify a resource over the web. URL has the following syntax:

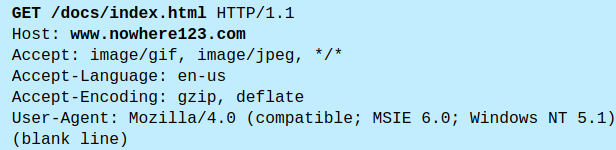




For example, in the URL http://www.nowhere123.com/docs/index.html, the communication protocol is HTTP; the hostname is www.nowhere123.com. The port number was not specified in the URL, and takes on the default number, which is TCP port 80 for HTTP. The path and file name for the resource to be located is "/docs/index.html".

**HTTP Protocol**

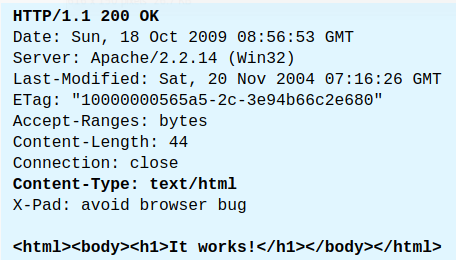
As mentioned, whenever you enter a URL in the address box of the browser, the browser translates the URL into a request message according to the specified protocol; and sends the request message to the server. For example, the browser translated the URL http://www.nowhere123.com/doc/index.html into the following **request message**:



When this request message reaches the server, the server can take either one of these actions:

1. The server interprets the request received, maps the request into a file under the server's document directory, and returns the file requested to the client.
2. The server interprets the request received, maps the request into a program kept in the server, executes the program, and returns the output of the program to the client.
3. The request cannot be satisfied, the server returns an error message. 404

An example of the HTTP **response** message is as shown:

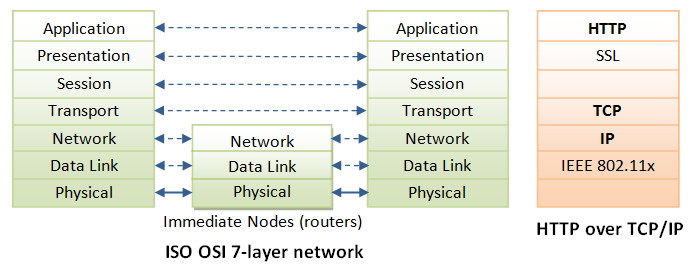


The browser receives the response message, interprets the message and displays the contents of the message on the browser's window according to the media type of the response (as in the Content-Type response header). Common media type include "text/plain", "text/html", "image/gif", "image/jpeg", "audio/mpeg", "video/mpeg", "application/msword", and "application/pdf".

In its idling state, an HTTP server does nothing but listening to the IP address(es) and port(s) specified in the configuration for incoming request. When a request arrives, the server analyzes the message header, applies rules specified in the configuration, and takes the appropriate action. The **webmaster's main control over the action of web server is via the configuration**, which will be dealt with in greater details in the later sections.

**HTTP over TCP/IP**

HTTP is a client-server application-level protocol. It typically runs over a TCP/IP connection, as illustrated. (HTTP needs not run on TCP/IP. It only presumes a reliable transport. Any transport protocols that provide such guarantees can be used.)



**TCP/IP** (Transmission Control Protocol/Internet Protocol) is a set of transport and network-layer protocols for machines to communicate with each other over the network.

**IP** (Internet Protocol) is a network-layer protocol, deals with network addressing and routing. In an IP network, each machine is assigned a unique IP address (e.g., 165.1.2.3), and the IP software is responsible for routing a message from the source IP to the destination IP. In IPv4 (IP version 4), the IP address consists of 4 bytes, each ranges from 0 to 255, separated by dots, which is called a quad-dotted form. This numbering scheme supports up to 4G addresses on the network. The latest IPv6 (IP version 6) supports more addresses. Since memorizing number is difficult for most of the people, an english-like domain name, such as www.nowhere123.com is used instead. The **DNS (Domain Name Service)** translates the domain name into the IP address (via distributed lookup tables). A special IP address 127.0.0.1 always refers to your own machine. It's domian name is "localhost" and can be used for local loopback testing.

**TCP** (Transmission Control Protocol) is a transport-layer protocol, responsible for establish a connection between two machines. TCP consists of 2 protocols: **TCP** and **UDP** (User Datagram Package). TCP is reliable, each packet has a sequence number, and an acknowledgement (ACK) is expected. A packet will be re-transmitted if it is not received by the receiver. Packet delivery is guaranteed in **TCP**. **UDP** does not guarantee packet delivery, and is therefore not reliable. However, **UDP** has less network overhead and can be used for applications such as video and audio streaming, where reliability is not critical.

TCP multiplexes applications within an IP machine. For each IP machine, TCP supports (multiplexes) up to 65536 ports (or sockets), from port number 0 to 65535. An application, such as HTTP or FTP, runs (or listens) at a particular port number for incoming requests. Port **0 to 1023** are **pre-assigned** to popular protocols, e.g., **HTTP at 80**, **FTP at 21**, Telnet at 23, SMTP at 25, NNTP at 119, and DNS at 53. Port 1024 and above are available to the users.

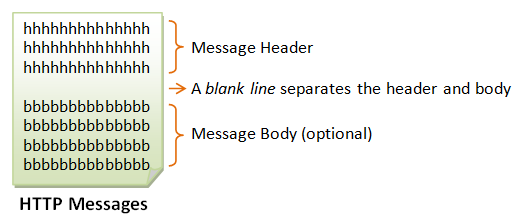
Although TCP port 80 is pre-assigned to HTTP, as the default HTTP port number, this does not prohibit you from running an HTTP server at other user-assigned port number (1024-65535) such as 8000, 8080, especially for test server. You could also run multiple HTTP servers in the same machine on different port numbers. When a client issues a URL without explicitly stating the port number, e.g., http://www.nowhere123.com/docs/index.html, the browser will connect to the default port number 80 of the host www.nowhere123.com. You need to explicitly specify the port number in the URL, e.g. http://www.nowhere123.com:8000/docs/index.html if the server is listening at port 8000 and not the default port 80. In brief, to communicate over TCP/IP, you need to know (a) IP address or hostname, (b) Port number.

**HTTP Specifications**

The HTTP specification is maintained by **W3C (World-wide Web Consortium)** and available at http://www.w3.org/standards/techs/http.

There are currently **two versions of HTTP**, namely, **HTTP/1.0** and **HTTP/1.1**. The original version, **HTTP/0.9 (1991)**, written by Tim Berners-Lee, is a simple protocol for transferring raw data across the Internet. HTTP/1.0 (1996) (defined in RFC 1945), improved the protocol by allowing MIME-like messages. HTTP/1.0 does not address the issues of proxies, caching, persistent connection, virtual hosts, and range download. These features were provided in HTTP/1.1 (1999) (defined in RFC 2616).

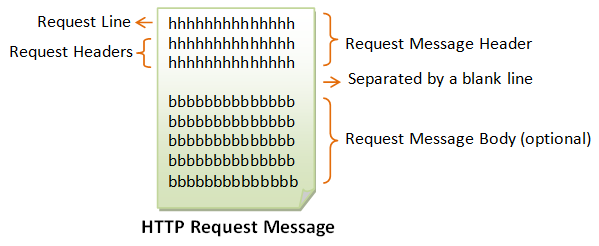
**HTTP Request and Response Messages**



HTTP client and server communicate by sending text messages. The client sends a request message to the server. The server, in turn, returns a response message.

An HTTP message consists of a **message header** and an optional **message body**, separated by a blank line, as illustrated below:

**HTTP Request Message**



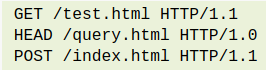
The format of an HTTP request message is as follow:

**Request Line**

The first line of the header is called the request line, followed by optional request headers. The request line has the following syntax:



* **request-method-name**: HTTP protocol defines a set of request methods, e.g., GET, POST, HEAD, and OPTIONS. The client can use one of these methods to send a request to the server.
* **Request-URI [*unified resource identifier*]**: specifies the resource requested.
* **HTTP-version**: Two versions are currently in use: HTTP/1.0 and HTTP/1.1.

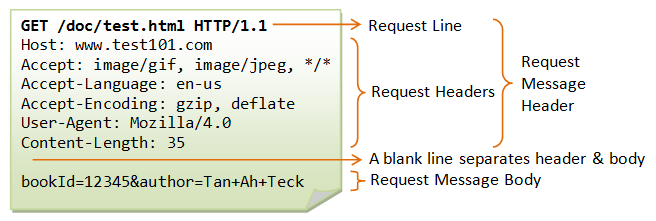
Examples of request line are:

**Request Headers**

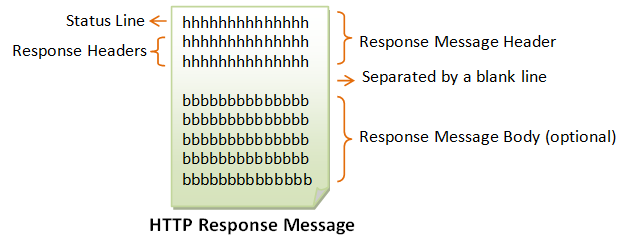
The request headers are in the form of name:value pairs. Multiple values, separated by commas, can be specified.



The following shows a sample HTTP request message:



**HTTP Response Message**

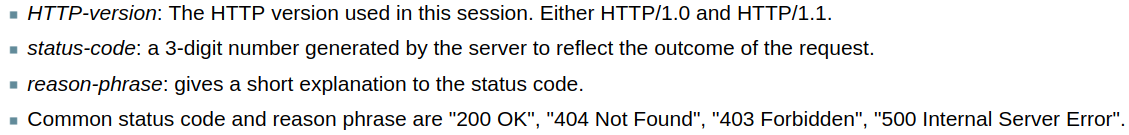


The format of the HTTP response message is as follows:

**Status Line**

The first line is called the status line, followed by optional response header(s). The status line has the following syntax:

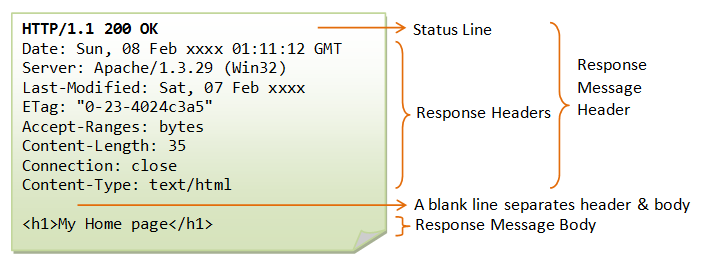




**Response Headers**

The response headers are in the form name:value pairs:



The following shows a sample response message:

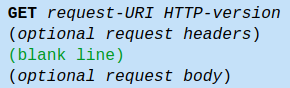
**HTTP Request Methods**

HTTP protocol defines a set of request methods. A client can use one of these request methods to send a request message to an HTTP server.

The methods are:

* **GET**: A client can use the GET request to get a web resource from the server.
* **HEAD**: A client can use the HEAD request to get the header that a GET request would have obtained. Since the header contains the last-modified date of the data, this can be used to check against the local cache copy.
* **POST**: Used to post data up to the web server. The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server
* **PUT**: Ask the server to store the data. The PUT method replaces all current representations of the target resource with the request payload.
* **DELETE**: Ask the server to delete the data.
* **TRACE**: Ask the server to return a diagnostic trace of the actions it takes.
* **OPTIONS**: Ask the server to return the list of request methods it supports.
* **CONNECT**: Used to tell a proxy to make a connection to another host and simply reply the content, without attempting to parse or cache it. This is often used to make SSL connection through the proxy.
* Other extension methods.

**"GET" Request Method**

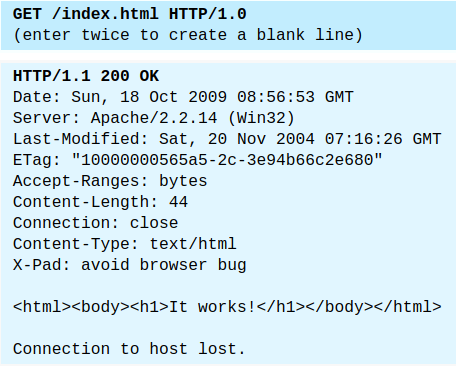
GET is the most common HTTP request method. A client can use the GET request method to request (or "get") for a piece of resource from an HTTP server. A GET request message takes the following syntax:

* The keyword GET is case sensitive and must be in uppercase.
* **request-URI**: specifies the path of resource requested, which must begin from the root "/" of the document base directory.
* **HTTP-version**: Either HTTP/1.0 or HTTP/1.1. This client negotiates the protocol to be used for the current session. For example, the client may request to use HTTP/1.1. If the server does not support HTTP/1.1, it may inform the client in the response to use HTTP/1.0.
* The client uses the optional request headers (such as Accept, Accept-Language, and etc) to negotiate with the server and ask the server to deliver the preferred contents (e.g., in the language that the client preferred).
* GET request message has an optional request body which contains the query string (to be explained later).

**HTTP** (HyperText Transfer Protocol) is a protocol that utilizes **TCP** to transfer its information between computers (usually Web servers and clients). The client makes an **HTTP** request to the Web server using a Web browser, and the Web server sends the requested information (website) to the client.

Remember, **IP** is required to connect all networks; **TCP** is a mechanism that allows us to transfer data safely; and **HTTP**, which utilizes **TCP** to transfer its data, is a specific protocol used by Web servers and clients.

**HTTP/1.0 GET Request**



The following shows the response of an HTTP/1.0 GET request (issue via telnet or your own network program - assuming that you have started your HTTP server):

In this example, the client issues a GET request to ask for a document named "/index.html"; and negotiates to use HTTP/1.0 protocol. A blank line is needed after the request header. This request message does not contain a body.

The server receives the request message, interprets and maps the request-URI to a document under its document directory. If the requested document is available, the server returns the document with a response status code "200 OK". The response headers provide the necessary description of the document returned, such as the last-modified date (Last-Modified), the MIME type (Content-Type), and the length of the document (Content-Length). The response body contains the requested document. The browser will format and display the document according to its media type (e.g., Plain-text, HTML, JPEG, GIF, and etc.) and other information obtained from the response headers.

**Notes**:

* The request method name "GET" is case sensitive, and must be in uppercase.
* If the request method name was incorrectly spelt, the server would return an error message "501 Method Not Implemented".
* If the request method name is not allowed, the server will return an error message "405 Method Not Allowed". E.g., DELETE is a valid method name, but may not be allowed (or implemented) by the server.
* If the request-URI does not exist, the server will return an error message "404 Not Found". You have to issue a proper request-URI, beginning from the document root "/". Otherwise, the server would return an error message "400 Bad Request".
* If the HTTP-version is missing or incorrect, the server will return an error message "400 Bad Request".
* **In HTTP/1.0, by default, the server closes the TCP connection after the response is delivered**. If you use telnet to connect to the server, the message "Connection to host lost" appears immediately after the response body is received. **You could use an optional request header "Connection: Keep-Alive" to request for a persistent (or keep-alive) connection, so that another request can be sent through the same TCP connection to achieve better network efficiency. On the other hand, HTTP/1.1 uses keep-alive connection as** **default**.

**Response Status Code**

The first line of the response message (i.e., the status line) contains the response **status code**, which is generated by the server to indicate the outcome of the request.

The status code is a 3-digit number:

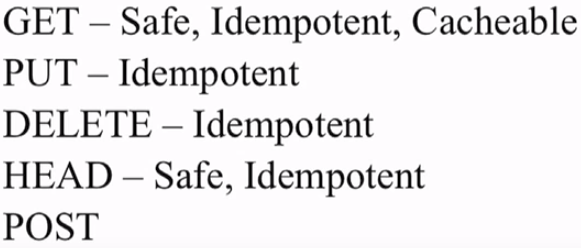
* **1xx** (Informational): Request received, server is continuing the process.
* **2xx** (Success): The request was successfully received, understood, accepted and serviced.
* **3xx** (Redirection): Further action must be taken in order to complete the request.
* **4xx** (Client Error): The request contains bad syntax or cannot be understood.
* **5xx** (Server Error): The server failed to fulfill an apparently valid request.

**Example: Keep-Alive Connection**

By default, for HTTP/1.0 GET request, the server closes the TCP connection once the response is delivered. You could request for the TCP connection to be maintained, (so as to send another request using the same TCP connection, to improve on the network efficiency), via an optional request header "**Connection: Keep-Alive**". **The server includes a "Connection: Keep-Alive" response header to inform the client that he can send another request using this connection, before the keep-alive timeout**. Another response header "Keep-Alive: timeout=x, max=x" tells the client the timeout (in seconds) and the maximum number of requests that can be sent via this persistent connection.

**Encoded URL**

URL cannot contain special characters, such as blank or '~'. Special characters are encoded, in the form of %xx, where xx is the ASCII (**UTF-8 mi demek istedi acaba?**) hex code. For example, '~' is encoded as %7e; '+' is encoded as %2b. A blank can be encoded as **%20** or '**+**'. The URL after encoding is called encoded URL.

 Idempotent means the request can be done multiple times

Safe there are no side effects performing that action

Post has non of these characteristics



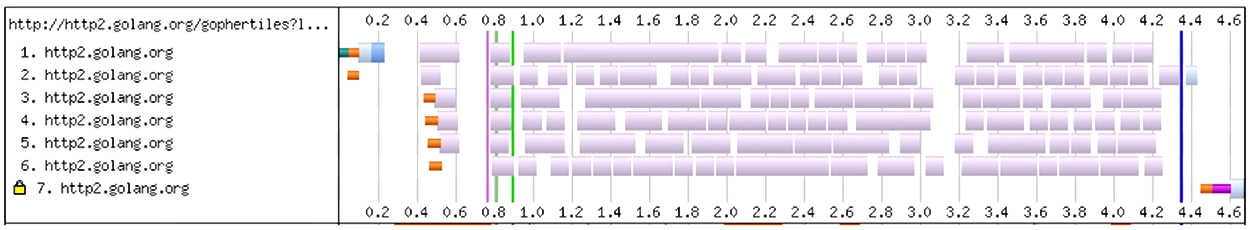
**Flaw 1: Head of Line Blocking (one resource at a time)**

When you send a request on a connection to a server, that connection becomes from now on useless until that point that the server has responded completely with a response.

Originally (also written in the http specs) a browser was only allow to have **2 concurrent connections to the same server**. So you would put to each one some requests and you would wait till you get a response back to send a new request. This was a bottleneck. Then people raised the limit to **6 concurrent connections**.

This means that you cannot download more than 6 large files from e.g. dropbox. Different browsers have different limits. The word here is **concurrent** (also **same server**). Meaning that you need to wait till one of the 6 connection finishes its fetching and then you can send a request from that connection again to the server.

<https://http2.golang.org/gophertiles> this is a nice demo that shows the difference in h1 and h2 speeds when it comes to HOL Blocking.



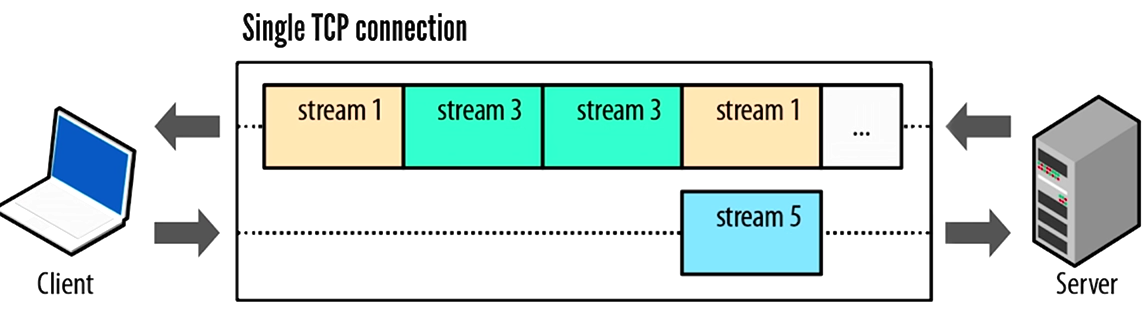
**Flaw 2: Meta Data**

Headers repeat a lot across requests. The most notorious examples for this are cookies and user agents. Both of them are quite long and static, being added to each request again and again. Since http is stateless, cookies are now also added in the header over and over again.

**HTTP 2.0**

H2 is a protocol upgrade. Every connection starts out as a h1 and then if the client supports h2 then it will switch, if not it will stay h1.

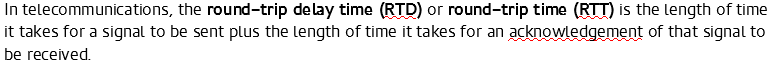
H2 is a **single TLS encrypted connection**. Because head of line blocking, the issue of having ‘only’ 6 connections is being addressed on a protocol level. So we can actually work with a single connection.

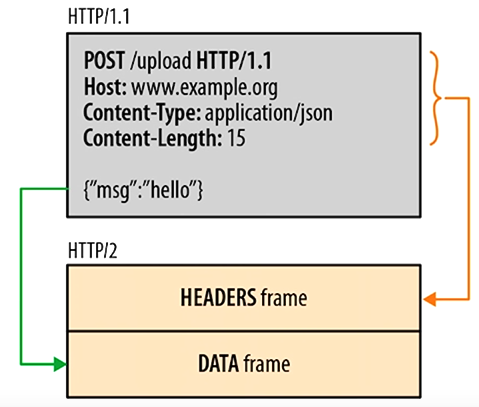
The connection looks like this:

What used to be a request-response pair in h1 is now a logical stream. All streams are actually sharing this connection. They are **multiplexed** on this single connection and share the bandwidth.

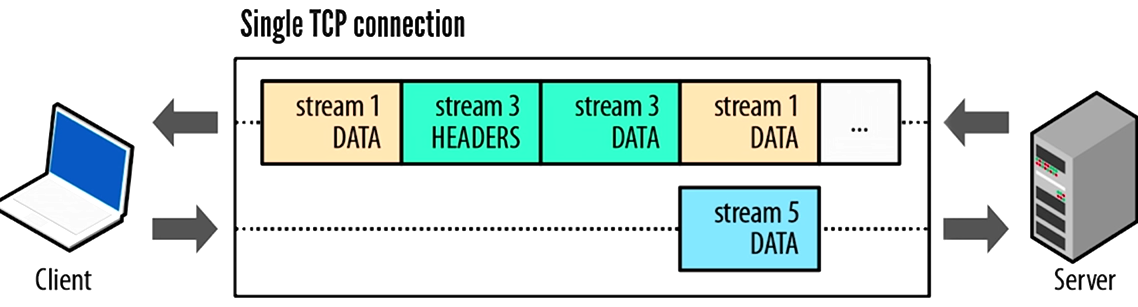
HoL blocking is not an issue anymore because these streams are being chopped into frames and put on the connection. This means if one stream goes into the blocking state where its waiting for a response or for the client to push more data, another stream can come in and utilize the connection fully.

An additional feature is that the client can specify dependenc ies and weights for each of these streams so things like “this stream is only useful when this other stream has finished”. The weights are there to distribute the bandwidth more appropriately.



So the data is chopped into frames. There is multiple frame types but the most important ones are

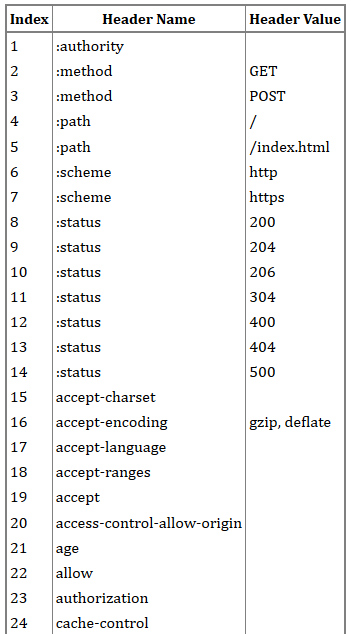
* Header Frames
* Data Frames



* **Don’t forget that all of this is end to end encrypted because TLS is enabled**.



This feature alone, having no HoL blocking and the single connection that does multiplexing actually invalidates (removes the need of) a lot of the best practices that we already know and have to do. All these best practices were there to reduce the number of requests and to get more out of a single request.

Since the header data is separate from the actual data, we can actually compress headers now. Not only that, **HPACK** was developed specifically for http headers.

The compression is not done for each stream but rather for the whole connection. There is a lookup table (static talbe), which was created from the most frequent header fields used by popular web sites. There are **62 predefined headers (1-61)**, some of them also with values.

Both sides have a dynamic table, an it is initially empty. Entries are added as each header block is decompressed.

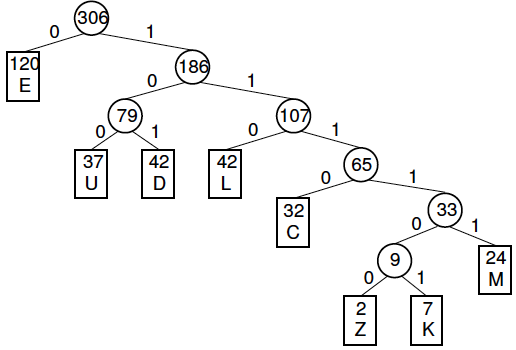
So you can just pass index values to the other side. Then the header will be reconstructed at the other side with these index values.

On the first request, the unknown headers will be saved first on the sender side in the dynamic table, also when the other side receives the request it will write it to its own dynamic table. If you want to send the same request later, you just send 2,7,62.





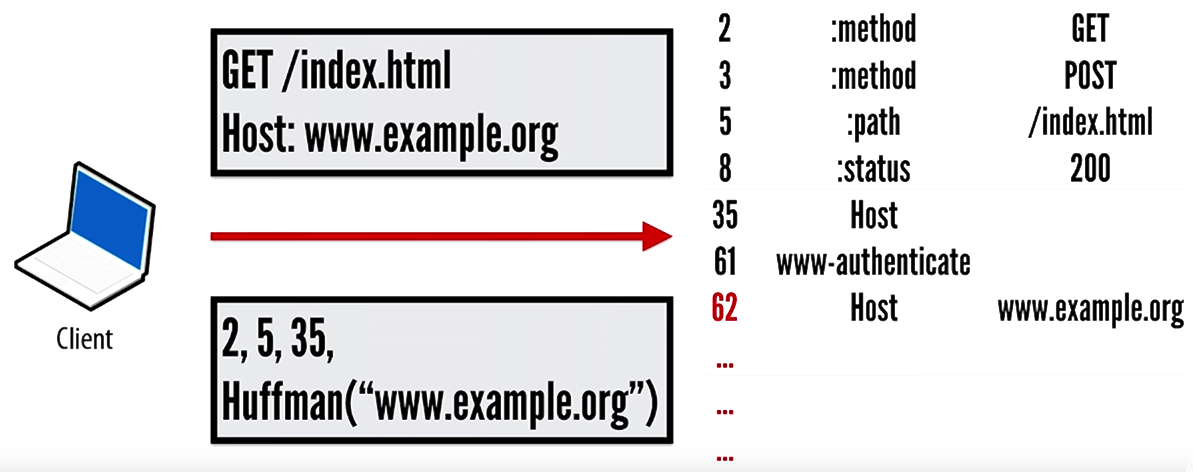
On top of this comes the **Huffman coding**. Its invented in **1952**. There are much better and modern versions of this (using blocks instead single characters) but still this is the basic foundation of modern text compression. The re-occuring letters will be used again in a smart way to save space. Huffman **proved** that this is the **most efficient way to assign 0’s and 1’s to single characters**. But if you start working with bigger blocks and not just one characters at a time then you will end up with things like **zip** **files**.

You count how many times each character is used and you put that in a list in order. En az kullanılan 2 sayıyı yan yana yazıp, indexlerini toplayıp yeni bir node yapıyorsun. Bu yeni sayıyı tekrar listenin içine sırasına göre koyuyorsun. Bütün sayılar için bunu yaptıktan sonra eline bir **Huffman Tree** geçiyor.

You can encode the word using this tree. Just go to each letter in your word and write down the bits on the path. If you go to the left, write a 0. To the right, write a 1, till you reached the desired letter. At the end you will have a code that is usually shorter than 8 bits.

There is already a pre-definied huffman table available (?) It will be on both sides so some amounts of bits can be saved by just sending the shorter version to the other side (?)

* So the workd Hullemulle is originally 80 bytes but after the compression it becomes 22 bytes with the following bits: 110 100 0 0 101 111 100 0 0 101. This can be also calculated by using the tree.

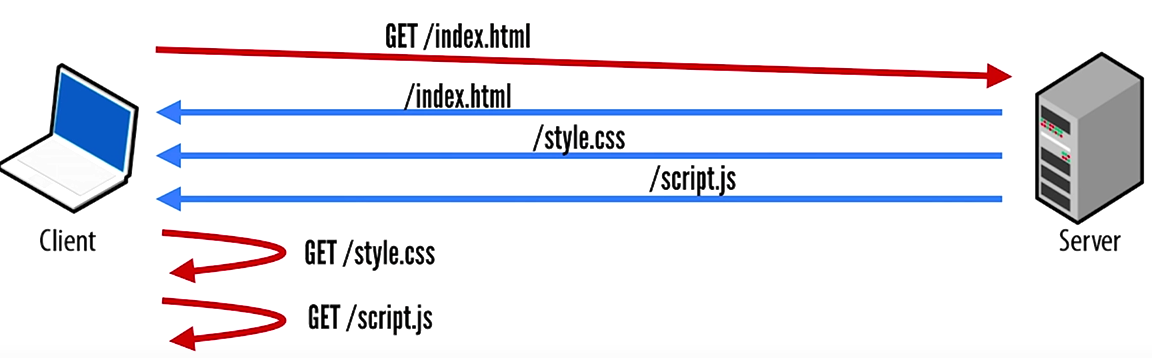


Once we send a request to example.org, it will be saved to the dynamic table. The next request to the same host with a different path, we can just write

**2, HuffmanOf(“/something”), 62**



And having this feature again saves us from doing a lot of the best practices. Even really long session id’s become highly compressible.

**Push**

Probably the best known feature of h2 is push. It allows you to respond to a request that hasnt even been send yet. So when you request /index.html, the server will give you the file but also it will predict that you will need some other files as well with index.html.

Pushing mindlesly can have a negative impact. So need to follow the best practices for what should be and what shouldn’t be pushed.

There are still things that you as a developer need to take care of:

* Setting sensible cache control headers
* Reduce the number of origins and therefore DNS lookups.
* Taking care of a good first render
* Most importantly compression. It is performance for free. Just have gzip enabled on your server. You don’t need to upload gziped files. That is something that your web server does for you, compression on the fly. In 2015, 47% HTML, 36% CSS, 26% JS is sent uncompressed.

All the major browsers support h2.Also a lot of the servers have merged the h2 implementation. Most languages also have implementations for http2 but some features might not be completele. If you are wondering about this, check [www.bit.ly/http2implementations](http://www.bit.ly/http2implementations)

Tear 1 commitment to h2: So if you want to switch to h2, put all your static assets to an h2 enabled CDN. The biggest number of request to a website is for assets.

Tear 2 commitment to h2: having a h2 enabled reverse proxy infront of your current server. It basically removes HoL blocking and does the header compression for you. It works because from backend to backend you don’t have a limit of just 6 connections, speeding your app.

Future: Web sockets might not be a thing anymore. They still work just as before but they are a separate protocol and therefore a separate connection. They are not beeing multiplexed over the h2 connection because its a separate protocol. Time will tell.

? HTTP 2 is a **binary** protocol ???

h1 is text?? telnet wont work with h2

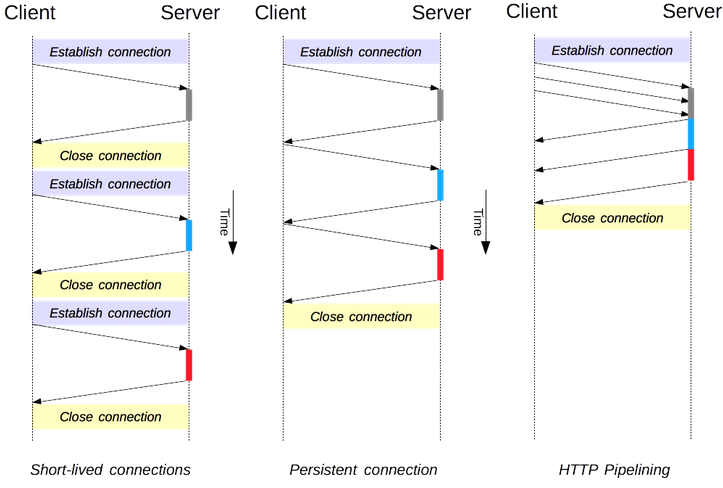
**HTTP persistent connection**, also called **HTTP keep-alive**, or **HTTP connection reuse**, is the idea of using a single TCP connection to send and receive multiple HTTP requests/responses, as opposed to opening a new connection for every single request/response pair. The newer HTTP/2 protocol uses the same idea and takes it further to allow multiple concurrent requests/responses to be multiplexed over a single connection.

**Connection management in HTTP/1.x**

Connection management is a key topic in HTTP: opening and maintaining connections largely impacts the performance of Web sites and Web applications. In HTTP/1.x, there are several models: short-lived connections, persistent connections, and HTTP pipelining.

HTTP mostly relies on TCP for its transport protocol, providing a connection between the client and the server. In its infancy, HTTP used a single model to handle such connections. These connections were short-lived: a new one created each time a request needed sending, and closed once the answer had been received.

This simple model held an innate limitation on performance: opening each TCP connection is a resource-consuming operation. Several messages must be exchanged between the client and the server. Network latency and bandwidth affect performance when a request needs sending. Modern Web pages require many requests (a dozen or more) to serve the amount of information needed, proving this earlier model inefficient.

Two newer models were created in HTTP/1.1. The persistent-connection model keeps connections opened between successive requests, reducing the time needed to open new connections. The HTTP pipelining model goes one step further, by sending several successive requests without even waiting for an answer, reducing much of the latency in the network.

**HTTP/2 adds additional models for connection management.**

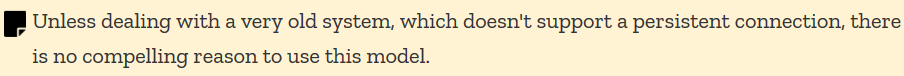
It's important point to note that connection management in HTTP applies to the connection between two consecutive nodes, which is hop-by-hop and not end-to-end. The model used in connections between a client and its first proxy may differ from the model between a proxy and the destination server (or any intermediate proxies). The HTTP headers involved in defining the connection model, like Connection and Keep-Alive, are hop-by-hop headers with their values able to be changed by intermediary nodes.

A related topic is the concept of HTTP connection upgrades, wherein an HTTP/1.1 connection is upgraded to a different protocol, such as TLS/1.0, WebSocket, or even HTTP/2 in cleartext. This protocol upgrade mechanism is documented in more detail elsewhere.

**Short-lived connections**

The original model of HTTP, and the default one in HTTP/1.0, is short-lived connections. Each HTTP request is completed on its own connection; this means a TCP handshake happens before each HTTP request, and these are serialized.

The TCP handshake itself is time-consuming, but a TCP connection adapts to its load, becoming more efficient with more sustained (or **warm**) **connections**. Short-lived connections do not make use of this efficiency feature of TCP, and performance degrades from optimum by persisting to transmit over a new, cold connection.

This model is the default model used in HTTP/1.0 (if there is no Connection header, or if its value is set to close). In HTTP/1.1, this model is only used when the Connection header is sent with a value of close.

**Persistent connections**

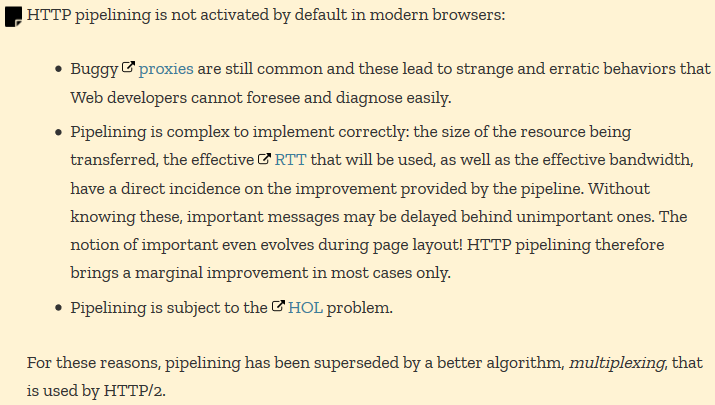
SectionShort-lived connections have two major hitches: the time taken to establish a new connection is significant, and performance of the underlying TCP connection gets better only when this connection has been in use for some time (warm connection). To ease these problems, the concept of a **persistent** **connection** has been designed, even prior to HTTP/1.1. Alternatively this may be called a **keep-alive connection**.

A persistent connection is one which remains open for a period of time, and can be reused for several requests, saving the need for a new TCP handshake, and utilizing TCP's performance enhancing capabilities. This connection will not stay open forever: idle connections are closed after some time (a server may use the Keep-Alive header to specify a minimum time the connection should be kept open).

Persistent connections also have drawbacks; even when idling they consume server resources, and under heavy load, **DoS attacks** can be conducted. In such cases, using non-persistent connections, which are closed as soon as they are idle, can provide better performance.

HTTP/1.0 connections are not persistent by default. Setting Connection to anything other than close, usually retry-after, will make them persistent.

In HTTP/1.1, persistence is the default, and the header is no longer needed (but it is often added as a defensive measure against cases requiring a fallback to HTTP/1.0).



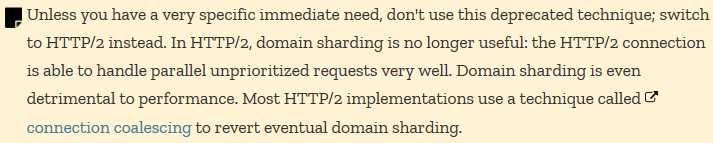
**HTTP pipelining**

By default, HTTP requests are issued sequentially. The next request is only issued once the response to the current request has been received. As they are affected by network latencies and bandwidth limitations, this can result in significant delay before the next request is seen by the server.

Pipelining is the process to send successive requests, over the same persistent connection, without waiting for the answer. This avoids latency of the connection. Theoretically, performance could also be improved if two HTTP requests were to be packed into the same TCP message. The typical MSS (Maximum Segment Size), is big enough to contain several simple requests, although the demand in size of HTTP requests continues to grow.

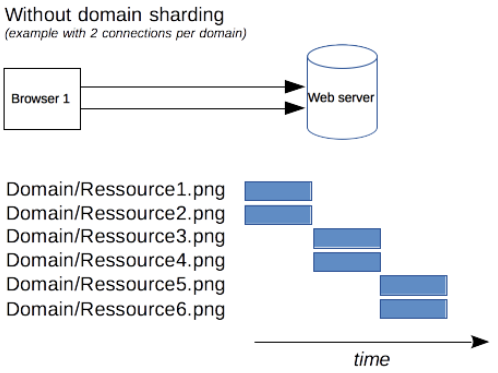
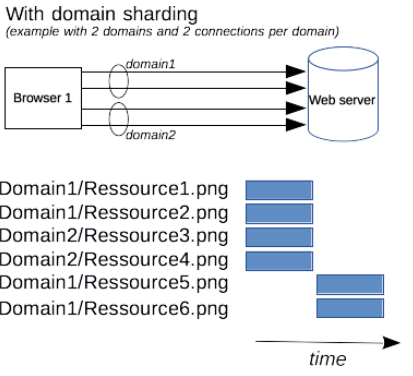
Not all types of HTTP requests can be pipelined: only idempotent methods, that is GET, HEAD, PUT and DELETE, can be replayed safely: should a failure happen, the pipeline content can simply be repeated.

Today, every HTTP/1.1-compliant proxy and server should support pipelining, though many have limitations in practice: a significant reason no modern browser activates this feature by default.

**Domain sharding**

As an HTTP/1.x connection is serializing requests, even without any ordering, it can't be optimal without large enough available bandwidth. As a solution, browsers open several connections to each domain, sending parallel requests. Default was once 2 to 3 connections, but this has now increased to a more common use of 6 parallel connections. There is a risk of triggering DoS protection on the server side if attempting more than this number.

If the server wishes a faster Web site or application response, it is possible for the server to force the opening of more connections. For example, Instead of having all resources on the same domain, say **www.example.com**, it could split over several domains, **www1.example.com**, **www2.example.com**, **www3.example.com**. Each of these domains resolve to the same server, and the Web browser will open 6 connections to each (in our example, boosting the connections to 18). **This technique is called domain sharding**.



**Cookie vs Cookie Jar in HttpClients**

As you described in your question, cookies are managed by browsers (HTTP clients) and they allow to store information on the clients' computers which are sent automatically by the browser on subsequent requests.

If your application acts as a client (you connect to remote HTTP servers using the net/http package), then there is no browser which would handle / manage the cookies. By this I mean storing/remembering cookies that arrive as Set-Cookie: response headers, and attaching them to subsequent outgoing requests being made to the same host/domain. Also cookies have expiration date which you would also have to check before deciding to include them in outgoing requests.

The http.Client type however allows you to set a value of type http.CookieJar, and if you do so, you will have automatic cookie management which otherwise would not exist or you would have to do it yourself. This enables you to do multiple requests with the net/http package that the server will see as part of the same session just as if they were made by a real browser, as often HTTP sessions (the session ids) are maintained using cookies.

The package net/http/cookiejar is a CookieJar implementation which you can use out of the box. Note that this implementation is in-memory only which means if you restart your application, the cookies will be lost.

**Cookie and Set-Cookie are different…**

Asd

**CORS**

Get can be still send to server

But for POST PUT etc. An OPTION request is send first and if that doesnt return cors allowed, then the rest of the request won’t go out.

So does that mean before GET there is no OPTION?

Table

Description automatically generated

<https://www.mocklab.io/blog/which-java-http-client-should-i-use-in-2020/>

**RestTemplate**

<https://stackoverflow.com/questions/52364187/resttemplate-exchange-vs-postforentity-vs-execute>

**SSL / TLS Certificate and Stuff**

<https://ubuntu.com/server/docs/security-certificates>