El protocolo HTTP

El protocolo HTTP es un idioma básico formado por preguntas y respuestas que entienden todos los clientes web (es decir, los navegadores) y todos los servidores web (es decir, los programas que tienen almacenadas páginas web y que las ofrecen a los clientes que las solicitan). Aunque se utilicen diferentes clientes web (Firefox, Chrome, Safari …) y en "el otro lado" esté funcionando distinto software de servidor web (Apache, Nginx, IIS…), el protocolo HTTP es un estándar que permite la comunicación entre ambos extremos de una forma universal. Concretamente, establece un conjunto muy específico de peticiones que un cliente web puede realizar, y un conjunto de respuestas predefinidas que un servidor web puede ofrecer a estas peticiones. Su versión 1.1 -las anteriores estan ya obsoletas- está desarrollada en el documento RFC 2616 (http://www.ietf.org/rfc/rfc2616.txt), su versión 2.0 -actualmente la más extendida- en el documento RFC 7540 (https://tools.ietf.org/html/rfc7540) y la más reciente, la versión 3.0, en el documento RFC 9114 (https://datatracker.ietf.org/doc/rfc9114)

> NOTA: A més dels documents oficials RFC anteriors, un altre recurs molt complet i útil per aprendre els detalls del protocol HTTP que recomanem (més enllà d'aquest propi document), és la referència mantinguda per Mozilla, disponible a https://developer.mozilla.org/en-US/docs/Web/HTTP , i també https://http.dev

Formato de las peticiones en HTTP 1.1, el método GET y manejo de URLs

Toda petición HTTP (realizada, por ejemplo, al escribir una dirección web en la barra de direcciones de un navegador y pulsar Enter) internamente, en su versión 1.1, no es más que un conjunto de líneas de texto. Esto cambia en las versiones 2.0 y 3.0 del protocolo, ya que, tal como veremos en un apartado posterior, estas versiones hacen que HTTP deje de estar basado en intercambios de texto y pase a tener una estructura de mensajes binaria, pero esto no quita que el estudio de la versión 1.1 nos siga siendo muy útil, ya que se manejan los mismos conceptos básicos y es más sencilla de comprender.

Así pues, nos interesará saber que la primera línea de una petición HTTP 1.1 siempre define el tipo de petición y tiene el formato siguiente: ***<MÉTODO>*** ***<PATH>*** ***<VERSIÓN>***. Las siguientes líneas (que podemos dividir por su significado y contenido entre "cabecera de la petición" y "cuerpo de la petición") aportan información adicional. Todas estas líneas se distinguen entre sí por su final, compuesto de los caracteres "\r\n" (retorno de carro más salto de línea).

El campo *<VERSIÓN>* simplemente indica la versión del protocolo utilizada para realizar la petición (que en este caso siempre será la cadena "**HTTP/1.1**"). El campo *<MÉTODO>* representa la acción que el cliente pretende ejecutar sobre un determinado recurso alojado en el servidor, el cual estará indicado mediante el campo *<PATH>*. Por ejemplo, lo más habitual es que el cliente (normalmente, un navegador) quiera obtener una página web alojada en un servidor; en ese caso, como valor de *<MÉTODO>* se usará el verbo **GET** –en mayúsculas- y como valor de *<PATH>* se indicará la ruta de la página web a conseguir, ruta que forma parte de la dirección escrita (normalmente) por el usuario en la barra de direcciones del navegador (y que ellos automáticamente extraerán de ella).

> NOTA: Els navegadors actuals (i la comanda Curl) per defecte empren el protocol HTTP/1.1 si la connexió s'estableix sense protecció (és a dir, sense TLS) i el protocol HTTP/2 si la connexió està protegida (és a dir, es realitza sobre TLS). En aquest darrer cas, però, podria passar que el servidor no acceptés (encara) el protocol HTTP/2 i calgués "degradar" la comunicació a la versió anterior, la 1.1. ¿Com es fa aquest procés? Doncs en el moment de realitzar el "handshake" TLS (concretament dins del missatge "Client Hello", és a dir, just quan es construeix el canal segur, que és el primer pas abans de començar cap comunicació) gràcies a l'ús d'una extensió del protocol TLS anomenada "**ALPN**" (Application-Layer Protocol Negotiation), la qual serveix precisament per a què el navegador llisti en ordre de preferència els protocols als quals vol "passar" a fer servir un cop s'hagi construït el canal segur (i on primer indicarà HTTP/2 -"h2"- i després, en tot cas, HTTP/1.1) i també per a què el servidor respongui (en el missatge "Server Hello" del handshake TLS) amb el protocol concret finalment escollit de la llista proporcionada pel navegador (si el servidor no retornés cap extensió ALPN, el navegador assumirà que només suporta HTTP/1.1). A partir d'aquí, ja es podrà establir la comunicació entre client i servidor via la versió del protocol HTTP acordada
>
> NOTA: Cal dir que, tot i que en teoria un navegador hauria de ser capaç de passar d'una connexió HTTP/1.1 sense TLS a una connexió HTTP/2.0 (sense TLS també) mitjançant l'ús de la capçalera de petició *Upgrade: h2c* (si el servidor l'accepta) a la pràctica aquest mecanisme és pràcticament inexistent

NOTA: El protocol HTTP/3 és diferent perquè en comptes de funcionar sobre els protocols TCP+TLS (com les versions anteriors) funciona sobre dos protocols diferents: UDP + QUIC (sobre el per què en parlarem més endavant). Com que el navegador no sap si el servidor suportarà el protocol QUIC, el que fa és iniciar la comunicació de la mateixa manera habitual, amb un "handshake" TLS (TCP) incloent l'extensió ALPN indicant que es prefereix la versió "h2" i és llavors el servidor, el que proactivament informarà en la seva resposta que sí que pot gestionar el protocol HTTP/3 mitjançant l'ús d'una capçalera de resposta HTTP anomenada "**Alt-Svc**", la qual indicarà el port UDP a fer servir per iniciar una nova connexió de tipus QUIC (i, opcionalment, el temps màxim que s'esperarà per tancar la connexió TCP actual). Darrerament s'està impulsant un altre sistema més òptim per iniciar una comunicació HTTP/3 que consistiria en rebre la informació de quina versió d'HTTP han de fer servir directament de la resposta DNS que resolgui al servidor HTTP en qüestió, gràcies a l'ús d'un registre DNS especial anomenat genèricament **SVCB** (però que pot ser particularitzat, anomenant-se per exemple **HTTPS en aquest cas),** que inclouria aquesta informació. Podeu aprofundir en aquest mecanisme llegint l'article https://blog.cloudflare.com/speeding-up-https-and-http-3-negotiation-with-dns/

NOTA: Otros métodos definidos en el protocolo HTTP son **POST,PUT,PATCH,DELETE,HEAD,TRACE** o **OPTIONS**; cada uno de ellos realiza una operación diferente sobre el recurso que se le haya indicado y serán estudiados más adelante

NOTA: Las "direcciones web" escritas en la barra de direcciones de un navegador han de escribirse respetando un determinado formato llamado URI (de "Uniform Resource Identificator") definido en el RFC 3986. No obstante, en la mayoría de ocasiones, se utiliza un caso particular de URI llamado URL (de "Uniform Resource Locator") concretado en el RFC 1738

Tal como hemos dicho, el campo <*PATH*> sirve, en el caso de usar el método GET, para indicar al servidor la ruta del recurso solicitado (el cual hay que hacer notar que puede ser no solo una página web, sino también una fotografía, un PDF…y en general, cualquier tipo de fichero). Y que dicha ruta forma parte de la URL indicada en la barra de direcciones del navegador (o en otras palabras, que el valor de <*PATH*> es una subcadena de dicha dirección). A continuación se muestra una URL de ejemplo genérica donde aparece el valor del campo <*PATH*> señalado en negrita y donde...:

*protocolo://maquinaremota:puerto/**ruta/hacia/un/recurso?clave1=valor1&clave2=valor2&clave3=valor3***

*El valor de *protocolo* puede ser la cadena "http" o "https" (este último solo si la conexión se estableciese sobre un canal cifrado; actualmente es el elegido por defecto por los navegadores si el usuario no escribe ninguno explícitamente).

*El valor de *maquinaremota* es el nombre DNS o dirección IP del servidor web al que se conecta

*El valor de *puerto* es el número donde estará escuchando ese servidor (si el protocolo empleado es "http" y dicho servidor usa el puerto nº80, no hará falta indicarlo, así como tampoco si el protocolo empleado es "https" y dicho servidor usa el puerto nº443)

*El valor "/ruta/hacia/un/recurso" es específicamente la ruta donde se aloja el recurso en cuestión dentro de la jerarquía de carpetas del servidor web al que ya se ha conectado

*Si ese recurso fuera de tipo dinámico (es decir, si fuera un programa ejecutable que cada vez pudiera retornar un resultado variable, como es por ejemplo una página PHP) y para solicitarlo/ejecutarlo se usara el método GET, se le podrían pasar parámetros de entrada mediante una cadena final cuyo formato consiste en escribir la primera pareja clave/valor precedida de "?" y después todas las demás separadas entre sí por "&". A esta cadena de parejas clave/valor enviada al servidor para ser procesada se le llama "querystring".

NOTA: Tal como hemos visto, respecto una URL típica, en el valor del campo <PATH> se omite el protocolo, el nombre/dirección del servidor y su puerto de escucha Esta diferencia es debida a que estos valores ya han debido de ser especificados en el momento de establecer la conexión entre cliente y servidor (proceso que es previo al envío de la petición propiamente dicha) y, por tanto, volverlos a indicar sería redundante. Por ejemplo, la primera línea de una petición HTTP 1.1 de tipo GET que solicitara (a un determinado servidor ya contactado) una página llamada "index.html" ubicada directamente bajo la carpeta "raíz", debería tener un aspecto como: `GET /index.html HTTP/1.1`. Así pues, y solo para acabar de aclarar este aspecto: ¿cuál debería ser la petición GET necesaria para obtener, por ejemplo, la página http://arduino.cc/en/Reference/HomePage? Respuesta: `GET /en/Reference/HomePage HTTP/1.1`

Resumiendo, si escribimos en la barra de direcciones de un navegador una URL real tal como http://www.google.com/search?q=guapo y pulsamos Enter, éste generará automáticamente una petición HTTP de tipo GET (este método es el utilizado por defecto por los navegadores si no se indica lo contrario) para acceder a un recurso llamado en este caso "search" ubicado en el servidor "www.google.com" (en otras palabras, al buscador de Google), al cual le estaremos indicando una pareja clave-valor (concretamente, la pareja "q=guapo"); con esta querystring, el buscador de Google estará recibiendo la información necesaria para realizar una búsqueda (clave "q") usando la palabra "guapo". De hecho, podemos comprobar fácilmente cómo, si accedemos a la página del buscador de Google a través de un navegador normal, dependiendo de lo que escribimos en el cuadro de búsqueda, así se modifica la cola "search?q=" de la dirección visible en la barra de direcciones.

Es importante tener en cuenta que un computador que actúe como servidor web normalmente tiene sus recursos compartidos (páginas web, imágenes, documentos, etc) alojados dentro de una carpeta "raíz" determinada (establecida en la configuración del programa Apache/Nginx/IIS/etc.), cuya ruta real dentro de su sistema operativo no es conocida por el cliente porque este solamente "ve" a partir de esa carpeta en adelante. Por ejemplo, si esa carpeta tuviera de ruta real "/var/www/html/misitioweb" y ahí dentro hubiera una subcarpeta "imagenes" con todas las imágenes de nuestro sitio web, para solicitar una imagen llamada "mifoto.png", en el navegador deberíamos escribir http://www.miservidor.com/imagenes/mifoto.png y no http://www.miservidor.com/var/www/html/misitioweb/imagenesmifoto.png.

Otro detalle que debemos saber a la hora de construir URLs es que no todos los caracteres están permitidos en una URL: solamente podemos utilizar las letras del alfabeto inglés minúsculas y mayúsculas, los dígitos del 0 al 9, los caracteres -_.!*'() y ciertos caracteres con significado especial dentro de las URLs, los cuales son: espacio en blanco,",#,$,%,&,+,´,/,:,;,<,=,>,?,@,[,\,],{,|,} y ~.

Por otro lado, si deseáramos escribir alguno de estos últimos sin mantener su significado especial (es decir, si quisiéramos, por ejemplo, que "?" dejara de marcar el inicio de la "querystring" para pasar a ser un simple interrogante) deberíamos "codificarlos" según la tabla mostrada a continuación:

Carácter	Codificación
	%20
"	%22
#	%23
$	%24
%	%25
&	%26
+	%2B
,	%2C
/	%2F
:	%3A
;	%3B
<	%3C
=	%3D
>	%3E
?	%3F
@	%40
[	%5B
\	%5C
]	%5D
{	%7B
|	%7C
}	%7D
~	%7E

Así pues, si una página web tuviera por ejemplo un espacio en blanco en su nombre (pongamos que se llama "datos clientes.html"), una URL válida incluiría dicho nombre transformado así: "datos %20clientes.html".

Las peticiones POST

Además del método GET, otro método ampliamente utilizado es el método POST. Este método está específicamente diseñado para enviar información desde el cliente (contenida en el cuerpo de la petición) hacia el recurso identificado por *<PATH>* (ubicado en el servidor), para que éste la procese según lo tenga programado. Un caso típico es el envío de datos desde un formulario web hacia una página PHP.

Pero hemos visto que el método GET también permite que un cliente pueda enviar datos al servidor mediante la creación de una querystring al final de la URL del recurso solicitado. Entonces, ¿cuál es la diferencia entre ambos métodos a la hora de enviar datos al servidor? ¿Cuándo convendrá utilizar un método y cuándo otro? Pues la diferencia principal está en el lugar dentro de la petición donde se encuentran los datos a enviar al servidor: con el método GET se envían, tal como hemos dicho, dentro de la propia URL solicitada (en forma de querystring, y por tanto, visibles para todo el mundo en la barra de direcciones del navegador) y con el método POST se envían dentro del cuerpo de la petición (y por tanto, permanecen algo más internos).

Los datos enviados mediante peticiones GET están escritos en un formato llamado "**application/x-www-form-urlencoded**" (o, para acortar, "URL-encode"); esto significa que cumplen dos requisitos: siguen la estructura –ya vista- de una querystring y usan las reglas de codificación de caracteres mostradas en la tabla anterior. Los datos enviados mediante POST pueden estar escritos igualmente en el formato "URL-encode" (es decir, seguir la estructura de querystring y estar codificados convenientemente -eso sí, dentro del cuerpo de la petición-) pero también pueden tener otro formato más específico llamado "**multipart/form-data**", diseñado especialmente para la transferencia de datos binarios (útil, pues, para la "subida" de ficheros) u otros más recientes, como "**application/json**", para transmitir datos en formato JSON. El formato utilizado por defecto en todas las peticiones POST, de todas formas, es siempre el "URL-encode" (para cambiarlo, se deberá utilizar una cabecera de cliente específica, "Content-Type", de la cual hablaremos próximamente).

Cabeceras de las peticiones

Las peticiones HTTP 1.1 no se componen solamente de una sola línea de tipo *<MÉTODO> <PATH> <VERSIÓN>* sino que están formadas por más líneas, las cuales pueden formar parte o bien de la llamada "cabecera" de la petición o bien del "cuerpo" de la petición. La separación entre cabecera y cuerpo se realiza gracias a una línea en blanco (es decir, una línea que contiene solamente los caracteres "\r\n" y ninguno más) entre ambas secciones.

Las líneas de cuerpo solamente aparecen en peticiones de tipo POST porque son las que contienen los datos que el cliente envía al servidor (tal como ya se ha comentado).

Las líneas de la cabecera sirven para informar al servidor sobre detalles técnicos de la petición, permitiendo que éste pueda componer una respuesta más adecuada. Todas estas líneas tienen la forma *Nombre:Valor* (no se distinguen mayúsculas de minúsculas) y están definidas en el documento RFC 2616. Todas ellas son opcionales excepto una: la línea "Host:xxxx" (donde "xxxx" representa en este caso el nombre DNS -o dirección IP- del servidor web al que el cliente quiere conectar). A continuación se lista una (muy) breve selección de las líneas de cabeceras más comunes:

> **Host** : Sirve para indicar, como hemos dicho, el nombre DNS (seguido de ":" y un nº de puerto si éste fuera diferente de 80) del servidor HTTP al que se le envía la petición. Por ejemplo, *Host: elpuig.xeill.net:4567* Esta cabecera puede parecer redundante, pero hay que pensar que actualmente es muy habitual que un servidor HTTP con una única dirección IP tenga asociados varios nombres DNS (en un entorno de "hosting", por ejemplo); en estos casos, esta cabecera permite localizar correctamente el recurso indicado en la URL, ya que hay que tener en cuenta que la petición cuando llega al servidor HTTP ya ha sido previamente resuelta a su (única) dirección IP a través del protocolo DNS y, por tanto, si no se indicara esta cabecera, no se podría saber a cuál de los eventualmente múltiples nombres DNS del servidor se debería ir para acceder al recurso deseado. Es la única línea de cabecera obligatoria.

Accept : Sirve para indicar al servidor, separados por comas, qué tipos MIME (de los recursos solicitados) el cliente es capaz de aceptar. Los recursos cuyo tipo MIME no esté incluido en la lista especificada en esta línea de cabecera, serán rechazados por el cliente. Un ejemplo (que solamente acepta páginas HTML) podría ser este: *Accept: text/html* . El orden en el que estén indicados los tipos MIME en esta cabecera es irrelevante, aunque que se puede añadir el modificador "q=nº" para indicar la preferencia de un cierto conjunto de tipos MIME sobre otros (un valor "q" de 1 es el más preferente, de 0.9 es algo menos, de 0.8 algo menos, etc...ver los ejemplos para mayor claridad)

Un tipo **MIME** es una cadena (cuyo formato está definido en el documento RFC 1341 (https://www.ietf.org/rfc/rfc1341.txt) que sirve para clasificar un recurso según su naturaleza: texto, imagen, audio... Cada tipo MIME consta de un tipo principal genérico ("text", "image", "audio"…) y un subtipo más concreto (adecuado al tipo principal) que indica el formato específico de dicho recurso. La lista oficial de tipos MIME está en la web de la IANA (aquí: http://www.iana.org/assignments/media-types/media-types.xhtml) pero algunos de los tipos MIME más habituales aparecen descritos en la web de Mozilla (aquí: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types).

Podemos destacar: *text/plain* (texto plano genérico), *text/html* (código HTML), *text/css* (código CSS), *application/javascript* (código Javascript), *application/json* (datos de tipo JSON), *application/x-www-urlencoded* (datos en formato "URL-encode"), *text/csv* (datos en formato CSV), *image/gif* (imagen GIF), *image/jpeg* (imagen JPG), *image/png* (imagen PNG), *application/pdf* (documento PDF), *application/gzip* (datos comprimidos de tipo Gzip) o *application/octet-stream* (datos sin estructura reconocida), entre otros. Para indicar en conjunto todos los subtipos de un tipo dado, se puede usar el símbolo especial "*", así, por ejemplo: *image/**. Igualmente, para indicar todos los tipos MIME posibles, se puede escribir */*.

Content-Type : Esta línea de cabecera solamente aparece en peticiones de tipo POST y sirve para informar al servidor sobre el tipo MIME de los datos enviados en el cuerpo de la petición (el cual suele ser "application/x-www-urlencoded" o "application/json").Si están en formato "URL-encode" (que es lo predeterminado), el valor de esta línea de cabecera será, por tanto, *Content-Type: application/x-www-form-urlencoded* . Siempre viene acompañada de otra línea de cabecera llamada **Content-Length**, la cual sirve para informar al servidor del tamaño (en bytes) del cuerpo de la petición (es decir, de los datos transferidos); un ejemplo: *Content-Length: 348*

Date : Sirve para informar al servidor de la fecha y hora en la que la petición fue enviada. Su valor se ha de expresar (en las pocas ocasiones que pueda ser necesario indicar esta línea de cabecera) en un formato (definido en el documento RFC 1123 (http://www.ietf.org/rfc/rfc1123.txt) que tiene el siguiente aspecto general: "díadelasemana, díadelmes mes año hora:minuto:segundo GMT". Por ejemplo, Date:Sat, 6 Jun 2016 10:10:10 GMT

User-Agent : Sirve para informar al servidor sobre qué programa concreto (y su versión) es el cliente que realiza la petición. Por ejemplo, *User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:32.0) Firefox/32.0* Esta línea de cabecera nos puede venir bien para simular ser un determinado navegador y comprobar el comportamiento del servidor según este dato.

Referer : Sirve para informar al servidor de la URL del recurso que fue solicitado justo antes del recurso actual. En la práctica, su valor suele ser la dirección de la página web que contenía el enlace que ha llevado a la petición vigente hacia la página actual. Por ejemplo, *Referer: http://www.rebelion.org*

Códigos de las respuestas

La respuesta HTTP 1.1 enviada por un servidor a un cliente (correspondiente a la petición hecha anteriormente por éste) también se compone internamente de varias líneas de texto terminadas con los caracteres "\r\n". La primera de estas líneas siempre define el tipo de respuesta ofrecida y tiene el formato siguiente: ***<VERSIÓN> <CÓDIGO> <FRASE>***. Las siguientes líneas (que podemos dividir en cabecera de la respuesta y cuerpo de la respuesta) aportan, respectivamente, información sobre detalles más técnicos de la respuesta y el contenido propiamente dicho de ésta.

El campo <*VERSIÓN*> simplemente indica la versión del protocolo utilizada para realizar la respuesta, (que en este caso siempre será la cadena "**HTTP/1.1**"). El campo <*CÓDIGO*> es un número de tres dígitos que define qué tipo de respuesta se está proporcionando. El campo <*FRASE*> contiene una breve explicación textual del valor de <*CÓDIGO*>; el documento RFC 2616 da recomendaciones sobre estas frases, pero son opcionales y pueden cambiarse como se estime oportuno.

Los valores posibles del campo <*CÓDIGO*> se pueden clasificar en cinco categorías, dependiendo del resultado obtenido en el servidor tras el procesamiento de la petición. Estas categorías vienen identificadas por el dígito de más a la izquierda, sirviendo los otros dos para especificar más al detalle el tipo de respuesta concreta. A continuación presentamos algunos de los códigos de respuesta comunes (aunque una lista más exhaustiva y completa se puede encontrar, además de en el RFC oficial, en https://httpstatuses.com o https://www.restapitutorial.com/httpstatuscodes.html)

1xx (Información) : El servidor indica que la petición ha sido recibida y que procederá a procesarla. Se usa en casos muy específicos que no veremos.

2xx (Éxito) : El servidor indica que la petición ha sido recibida y procesada con éxito. El código más típico de esta categoría es **200**, obtenido cuando la consulta ha sido satisfactoria y el recurso solicitado es devuelto correctamente (suponiendo que el método de la petición sea GET; si el método es POST el código entonces es **201**).

3xx (Redirección) : El servidor indica que la petición no se ha completado y se deben realizar más acciones para conseguir finalizarla. Generalmente se usa como respuesta a peticiones GET para señalar un cambio de localización del recurso solicitado (es decir, una nueva URL de una página ya existente). Esta nueva localización se indica en una cabecera enviada por el servidor justo para ese propósito: *Location*. Si el cambio es permanente, se emplea el código **301** y para peticiones siguientes el cliente debe usar la ubicación proporcionada en *Location*; si el cambio es temporal existen otros códigos (**302, 303, 307**…) y las siguientes peticiones deben seguir haciéndose a la URL original. Normalmente estos códigos son interpretados automáticamente por cualquier navegador actual, realizando la acción pertinente sin necesidad de notificarlo al usuario.

4xx (Error de cliente) : El servidor indica que la petición proveniente del cliente está mal formada o contiene errores. El código devuelto más común en estos casos es el **404**, que indica que el servidor no ha encontrado el recurso solicitado en la URL de la petición. Pero hay muchos más: el código **400** indica que la sintaxis de la petición HTTP es errónea; el código **401** indica que el usuario no ha presentado las credenciales adecuadas para poder estar autorizado a acceder a un recurso protegido (el proceso de autenticación y autorización de recursos está definido en un documento RFC específico, el 2617 (https://www.ietf.org/rfc/rfc2617.txt); el código **403** indica que el recurso está protegido para la solicitud actual, etc.

5xx (Error de servidor) : El servidor indica que, aunque la petición es correcta, ha fallado al procesarla. Los distintos motivos se señalan con un código concreto. Por ejemplo, **500** avisa de un error indeterminado o **501** indica que el servidor no soporta el método de la petición, entre otros.

Cabeceras de las respuestas

Las respuestas HTTP 1.1 no se componen solamente de una sola línea de tipo <*VERSIÓN*> <*CÓDIGO*> <*FRASE*> sino que, tal como ya se ha dicho, están formadas por más líneas, las cuales pueden formar parte o bien de la llamada "cabecera" de la respuesta o bien del "cuerpo" de la respuesta. La separación entre cabecera y cuerpo se realiza gracias a la existencia de una línea en blanco (es decir, una línea que contiene solamente los caracteres "\r\n" y ninguno más) entre ambas secciones.

Las líneas de cuerpo solamente aparecen en respuestas a peticiones de tipo GET y son el contenido del recurso propiamente solicitado (es decir, lo que habitualmente suele ser el código de una página web). Este contenido es el que los navegadores obtienen, interpretan y, como resultado, muestran en pantalla. Así pues, aunque el cuerpo de las respuestas a peticiones GET no está ceñido a ningun formato en particular; en

cada respuesta el servidor deberá indicar al cliente el tipo MIME concreto del contenido enviado (mediante la directiva *Content-Type*, que veremos enseguida) para que dicho cliente lo pueda interpretar convenientemente al recibirlo.

Las líneas de la cabecera sirven para informar al cliente sobre detalles más técnicos de la respuesta, permitiendo que éste pueda obtener una información más completa sobre el recurso en cuestión. Todas estas líneas tienen la forma *Nombre:Valor* (no se distingue entre mayúsculas y minúsculas) y también están definidas en el documento RFC 2616. A continuación se lista una breve selección de las líneas de cabeceras más habituales en las respuestas de los servidores web:

Connection : Sirve para informar al cliente sobre si la conexión TCP creada por el servidor para enviar la respuesta HTTP seguirá abierta o no. Si sí (valor k*eep-alive)*, esa conexión TCP podrá ser reutilizada para recibir más peticiones HTTP posteriores del mismo cliente; si no (valor *close*), será cerrada y para enviar una nueva respuesta HTTP el servidor tendrá que crear una nueva conexión.

Content-Type : Sirve, en respuestas a peticiones GET, para informar al cliente sobre cuál es el tipo MIME del contenido del cuerpo del recurso devuelto. También informa (pero sólo si el tipo MIME es textual, como *text/plain* o *text/html*) de su sistema de codificación (ASCII, UTF-8,etc), para que el cliente pueda mostrar dicho contenido correctamente. Esto último se indica mediante la palabra especial "charset" tras un punto y coma, así: *Content-Type: text/html; charset=UTF-8* (si el sistema de codificación no se indicara, el cliente empleará uno por defecto que dependerá de su configuración, el cual puede no coincidir con el del recurso). Salvo raras excepciones, UTF-8 es el sistema de codificación recomendado por ser el más versátil, óptimo y compatible entre clientes

Content-Length : Sirve para informar al cliente del tamaño (en bytes) del contenido del cuerpo del recurso devuelto (una vez comprimido, si es el caso). Si el valor de la línea *Connection* es *close*, el valor de esta línea es fácilmente calculable y se puede indicar directamente, así: *Content-Length: 348* . No obstante, si *Connection* es *keep-alive* (por ejemplo cuando se ofrece un fichero en "streaming"), el servidor no sabe de antemano este dato, por lo que esta línea de cabecera suele ser sustituida por la línea *Transfer-Encoding: chunked*, la cual indica al cliente que el recurso es enviado "a pedazos" progresivamente hasta enviar una marca final.

> NOTA: Aquesta capçalera, a HTTP/2.0 deixa de tenir sentit perquè aquesta informació està inclosa dins del propi "frame" binari de la resposta

Date : Sirve para informar al cliente de la fecha y hora en la que la respuesta fue generada. Su valor se ha de expresar en un formato definido en el documento RFC 1123 (igual que ocurre con la línea de cabecera de cliente homónima).

Expires : Sirve para informar al cliente de la fecha y hora en la que la respuesta dada se considerará caducada. Su valor se ha de expresar en un formato definido en el documento RFC 1123.

Location : Sirve para indicar al cliente la nueva URL de un recurso cuando la URL solicitada (correspondiente a ese recurso) ya no es válida. En general, gracias a esta información un navegador actual será capaz, al recibir una respuesta de tipo 3xx, de redireccionar automáticamente su petición a la URL recién indicada. Por ejemplo, *Location: http://www.fbi.gov*

Server : Informa al cliente del nombre y sistema del servidor. Por ejemplo, *Server:Apache/2.4 (Unix)*

Respecte les diferents versions del protocol

Els diferents mètodes de les peticions, els diferents codis de resposta, la codificació utilitzada de les rutes, els tipus MIME reconeguts i totes les capçaleres de client i de servidor que hem anat describint en els paràgrafs anteriors són les mateixes per les diferents versions del protocol HTTP, així que en aquest sentit la "semàntica" del protocol no canvia. Però existeixen algunes diferències importants entre la versió 1.1 (que hem fet servir com a referència en aquest document fins ara) i les versions 2.0 i 3.0 en referència a com s'implementa el protocol a baix nivell, "sobre el cable". Les detallarem a continuació:

Novetats a HTTP/2.0

* Els missatges intercanviats entre client i servidor passen a tenir un format binari en lloc d'estar basat en text. Això significa que per les màquines és més senzill d'interpretar, no té ambigüetats i és més ràpid de processar.

NOTA: Per entendre el nou format binari del protocol HTTP, cal tenir clar uns quants conceptes:

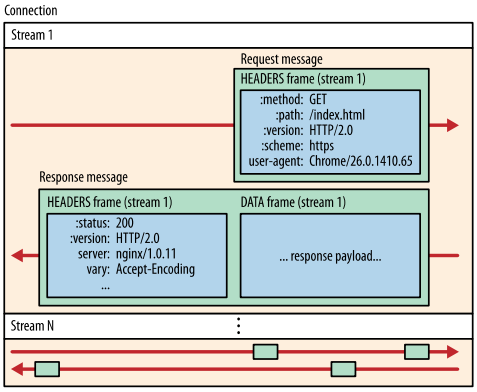
* S'anomena "**stream**" a cada canal bidireccional HTTP (dins d'una determinada connexió TCP) per on el client enviarà com a mínim una petició i en rebrà la resposta corresponent. Opcionalment, cada "stream" pot contenir informació de prioritat
* S'anomena "**frame**" a la unitat més petita de comunicació HTTP; tots els missatges transmesos per un "stream" estan compostos com a mínim per un "frame" (binari) contenint capçaleres (de petició o de resposta) i, eventualment, també un "frame" (binari) contenint el cos (de petició o de resposta). Cada "frame" ha d'incloure, en qualsevol cas, l'identificador de "stream" al qual pertany

Una petició GET típica està formada només per un "frame" de capçalera amb els següents valors:

:scheme: a HTTP/2.0 val sempre "https"
:method: el mètode de la petició (GET, POST, etc)
:path: la ruta del recurs demanat (seguint les mateixes restriccions indicades als paràgrafs anteriors)
:authority: equivalent a la capçalera Host
user-agent: una capçalera de petició...a partir d'aquí s'afegiran totes les altres capçaleres que calgui (noteu com no s'indica cap ":" a l'inici del seu nom)

La resposta corresponent a la petició GET anterior està formada, a més del "frame" de dades on s'inclou el cos de la resposta demanat, també per un "frame" de capçalera de resposta amb alguns dels següents valors:

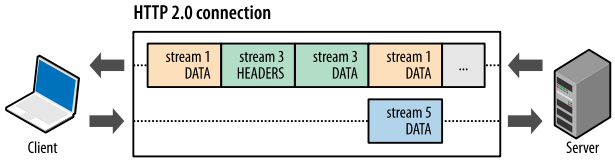
:status: el codi de resposta (200, 404, etc)
content-type: una capçalera de resposta...a partir d'aquí s'afegiran totes les altres capçaleres que calgui (noteu com no s'indica cap ":" a l'inici del seu nom)



* S'habilita la compressió binària de capçaleres de forma nativa, per tal d'intercanviar la menor quantitat d'informació possible entre els extrems i reduir així la latència de la comunicació, amb una tècnica anomenada HPACK

* S'utilitza, per part del client, una única connexió TCP per realitzar diferents peticions HTTP en paral·lel, cadascuna de les quals servirà per rebre un element diferent (pàgina HTML, document CSS o Javascript, imatge, etc) però tots transmesos dins de la mateixa, única connexió TCP. Això s'aconsegueix associant, a cada "frame" binari que es rep/s'envia a través de la mateixa connexió,

l'identificador de l'"stream" al que pertany (implementant així el que tècnicament s'anomena "multiplexació": els "frames" de diferents "streams" poden viatjar "mesclats" en un sola connexió sense problema perquè ja es reorganitzaran a l'altre extrem i sense bloquejar-se entre sí). L'objectiu és agilitzar l'intercanvi d'informació i la reutilització de sessions, a més de també fer consumir menys recursos de CPU, memòria i xarxa als dos extrems



* S'introdueix la funcionalitat "server push", la qual consisteix en què el servidor pot enviar directament a la memòria cau del navegador aquells elements que es preveuen que seran necessaris per visualitzar correctament la pàgina web servida (imatges, fonts, CSS, Javascript, etc) abans que el propi codi HTML d'aquesta pàgina hagi arribat com a resposta al navegador (i hagi sigut, doncs, interpretat per aquest). L'objectiu, un altre cop, és agilitzar l'intercanvi d'informació evitant de forma preventiva noves eventuals peticions explícites. Cal dir que aquesta funcionalitat actualment la comanda Curl no la implementa ja que requeriria mantenir el seu procés en segon pla més enllà del cicle de petició/resposta, cosa que per disseny no es desitja

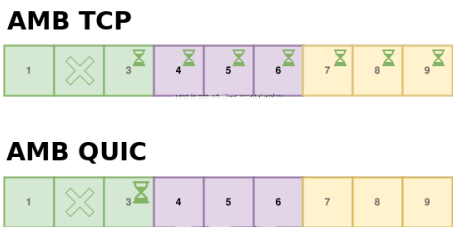
NOTA: Teniu més informació a https://http2.github.io/faq i https://http2-explained.haxx.se Altres articles d'interès són https://cabulous.medium.com/http-2-and-how-it-works-9f645458e4b2 o https://web.dev/articles/performance-http2

Novetats a HTTP/3.0

La gran diferència entre les versions anteriors del protocol i HTTP/3.0 és que aquesta ja no es basa en TCP sinó en UDP. La raó és per seguir optimitzant encara més la velocitat en la comunicació entre els dos extrems. No obstant, UDP, no incorpora moltes de les funcionalitats que, tot i de forma no prou òptima, TCP ofereix, com per exemple la capacitat de detectar la pèrdua/corrupció/desordre de paquets o la de controlar la congestió del tràfic, entre altres. Per tant, es va decidir implementar un nou protocol de transport que funcionés sobre UDP com a base però que incorporés (només) les parts de TCP optimitzades per les necessitats d'una comunicació àgil però fiable entre els extrems. Aquest protocol s'anomena QUIC i està estandaritzat al RFC 9000 (https://datatracker.ietf.org/doc/rfc9000)

A més, ja que s'estava inventant un nou protocol de transport "ad-hoc" per funcionar sota HTTP, moltes de les funcionalitats que a la versió 2.0 havia de realitzar el propi protocol HTTP ara es va aprofitar per "baixar-les" a un nivell inferior, al protocol QUIC, per tal d'optimitzar encara més el seu funcionament, com per exemple la multiplexació d'"streams" lliure de bloqueigs .

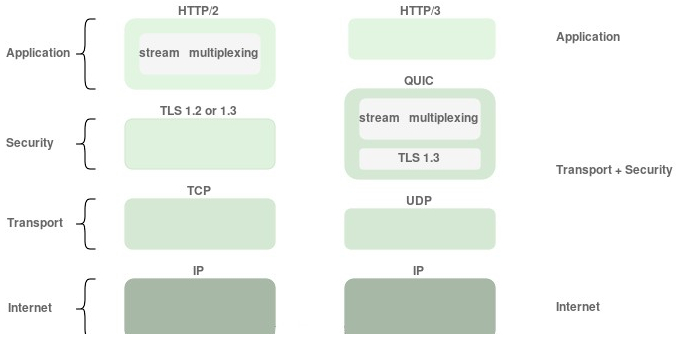
NOTA: Més en concret, un problema gros de TCP és que pot bloquejar tot un "stream" sencer (i els següents) si un sol paquet d'aquest "stream" es perd, fins que aquest paquet és retransmès (aquest problema és conegut com a bloqueig "Head-Of-Line"). QUIC elimina aquest problema evitant aquest bloqueig quan hi ha pèrdues de paquets. És el que mostra la imatge següent:



Una altra funcionalitat que es va aprofitar d'afegir a QUIC és la implementació del protocol TLS al seu interior (i no qualsevol, sinó la més moderna, la TLS v1.3!), de manera que forma part intrínseca d'ell mateix, no com una capa superior com passava fins llavors amb TCP. Això vol dir, doncs, que HTTP/3.0 només és "HTTPS": no existeix una variant no segura. Per tant, un servidor HTTP/3.0 haurà d'incorporar obligatòriament un certificat per poder funcionar, compte.

NOTA: Una avantatge colateral d'incloure el protocol TLS dins del propi QUIC és que s'estalvia haver de fer primer el "handshake" TCP i tot seguit el "handshake" TLS, ja que amb un sol "handshake" (el del QUIC) ja n'hi ha prou per a què la transferència i el xifratge de les dades ocorri de forma simultània, reduint-se així encara més la latència. És el que mostra la imatge següent:

Tot el que es comenta en els paràgrafs anteriors es pot veure de forma gràfica al següent esquema:



Altres millores de HTTP/3.0 respecte HTTP/2.0:

*Migració de connexió: Si un usuari canvia la seva interfície de xarxa (p.ex., de Wi-Fi a dades mòbils), QUIC pot continuar utilitzant la mateixa connexió sense interrupcions. Per contra, TCP vincula connexions a l'adreça IP específica, de manera que canviar de xarxes sovint significa iniciar una nova connexió.

*Reiniciació del 0-RTT: QUIC permet que un client enviï dades abans que s'estableixi formalment una connexió, accelerant així la primera sol·licitud al servidor web

*Format de compressió de capçaleres: Es reemplaça el format HPACK per QPACK per tal de fer la compressió compatible amb QUIC

NOTA: Cal tenir en compte, com ja s'ha comentat, que mentre que HTTP/2.0 pot ser negociat directament en el "handshake TLS" amb l'extensió ALPN, HTTP/3.0 funciona sobre QUIC, així que si el servidor HTTP informa (mitjançant la capçalera *Alt-Svc* que és compatible), el client necessitarà establir una nova connexió alternativa sobre QUIC i desfer-se'n de la connexió TCP original.

Teniu més informació a https://quicwg.org i https://http3-explained.haxx.se