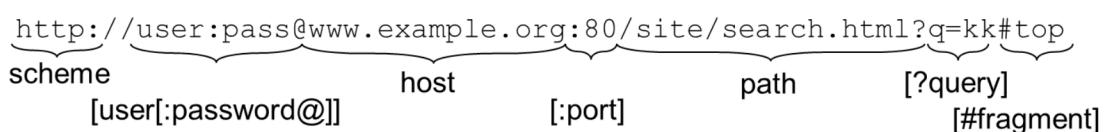


## 2.10. HyperText Transfer Protocol (HTTP)

HTTP es el protocolo empleado para navegar por la *World Wide Web* (WWW), puesto que es el protocolo que permite a los navegadores web descargar el código de las páginas web (codificadas en lenguaje HTML), de sus recursos asociados (imágenes, código *javascript*, hojas de estilo CSS, ficheros, etc.), así como interactuar con ellas (e.g. enviar los datos de un formulario al servidor web). Los recursos web se identifican mediante URLs (*Uniform Resource Locator*) [[RFC1738](#)] (**Figura 2.20**), lo que permite identificarlos y acceder a ellos, incluso si están en otro servidor.



**Figura 2.20** – Formato de una URL

HTTP es un protocolo-cliente servidor que funciona sobre TCP (puerto 80). El protocolo HTTPS no es más que el protocolo HTTP sobre TLS/TCP (puerto 443). El protocolo HTTP es un protocolo orientado a texto en el que el cliente envía peticiones, típicamente solicitando recursos, y el servidor responde con el recurso, más ciertas cabeceras de metainformación.

Las peticiones HTTP (**Figura 2.21**) están formadas por una línea de petición, seguida por varias cabeceras. Todas ellas son líneas de texto que terminan con los símbolos

ASCII de salto de línea y retorno de carro (\r\n). Algunos tipos de peticiones HTTP (POST, PUT) pueden incluir también un cuerpo de mensaje con datos, que se separa de las cabeceras mediante una línea en blanco.

```
GET /HTTP/1.1\r\n
Host: www.example.com\r\n
User-Agent: Wget/1.20.3 (linux-gnu)\r\n
Accept: */*\r\n
Accept-Encoding: identity\r\n
Connection: Keep-Alive\r\n
\r\n
```

**Figura 2.21 – Ejemplo de petición HTTP**

La línea de petición (en negrita) define el método HTTP utilizado, el *path*, *query* y *fragment* del recurso solicitado (o la URL completa si se utiliza un *proxy web*), y la versión del protocolo más alta que soporta el cliente (separados con espacios). La primera versión de HTTP/1.0 solo permitía descargar un único recurso por conexión TCP (puesto que utilizaba el fin de la conexión para señalizar el final de la descarga del recurso), por lo que fue rápidamente reemplazado por la versión HTTP/1.1 [RFC2616], que permite utilizar la misma conexión TCP para enviar múltiples peticiones y respuestas, una tras otra. HTTP/2.0 [RFC7540], aunque es compatible con los mensajes y cabeceras de sus antecesores, es un protocolo binario bastante más complejo, que define múltiples canales para poder enviar y recibir múltiples peticiones y respuestas simultáneamente sobre la misma conexión TCP.

Los principales métodos de HTTP son:

- **GET:** Sigue la descarga de un recurso.
- **POST:** Sigue añadir información (incluida en el cuerpo de la petición) a un recurso ya existente. Se utiliza por ejemplo para enviar los datos de un formulario al servidor.
- **PUT:** Sigue crear un recurso (incluido en el cuerpo de la petición).
- **DELETE:** Sigue borrar el recurso indicado.
- **HEAD:** Sigue descargar únicamente la metainformación de un recurso (i.e. sus cabeceras HTTP), no el recurso en sí.
- **OPTIONS:** Sigue la lista de métodos que se pueden ejecutar sobre el recurso.
- **CONNECT:** Este método se emplea para comunicarse con un servidor HTTPS a través de un *proxy* HTTP. En lugar de un recurso, se indica al *proxy* el nombre DNS o la dirección IP y el puerto del servidor al que debe conectarse.

Aunque para la navegación web normalmente solo se utilizan los métodos *GET*, *HEAD* y *POST*, las APIs REST [[REST](#)] están basadas en el protocolo HTTP y utilizan todos los métodos para realizar operaciones sobre los recursos (que también se representan con URLs).

Además de la línea de petición, las cabeceras más habituales que pueden aparecer en una petición HTTP son:

- ***Accept***: Lista de tipos de contenidos soportados por el cliente (e.g. "text/html, text/plain").
- ***Accept-Encoding***: Lista de formatos de compresión soportados por el cliente (e.g. "identity" o "gzip").
- ***Accept-Language***: Lista de lenguajes del usuario (e.g. "es-ES, en;q=0.8").
- ***Authorization / Proxy-Authorization***: Tipo de autenticación (Basic | Digest | Bearer) y autenticador para el servidor/proxy en Base64 (e.g. "Basic dXNlcjpwYXNzd29yZA==").
- ***Cache-control***: Directivas para la cache superiores (e.g. "no-cache").
- ***Connection***: Permite controlar la conexión TCP subyacente (e.g. "keep-alive").
- ***Cookie***: Devuelve una cookie al servidor (e.g. "SID=31d4d96e407aad42; lang=es-es")
- ***Content-Length***: Longitud del cuerpo de la petición en bytes.
- ***Content-Type***: Tipo MIME del cuerpo de la petición (e.g. "application/x-www-form-urlencoded")
- ***Host***: Nombre DNS del servidor (virtual).
- ***If-None-Match***: Petición condicional basada en la versión del contenido (i.e. cabecera de respuesta ETag).
- ***If-Modified-Since***: Petición condicional basada en la fecha de modificación del recurso (i.e. cabecera de respuesta Last-Modified).
- ***Max-Forwards***: Número máximo de saltos a través de proxies intermedios.
- ***Range***: Permite solicitar la descarga parcial del recurso (e.g. "bytes=1024-2048")
- ***Referer***: URL de la página web anterior.
- ***User-Agent***: Identificador del navegador web (e.g. "Mozilla/5.0 (Windows NT 10.0; WOW64; rv:42.0) Gecko/20100101 Firefox/42.0")
- ***Via***: Lista de proxies (<versión HTTP> <proxy> <agente>) por la que ha pasado la petición (e.g. "1.1 proxy.example.org:8080 (squid)").

Las respuestas HTTP ([Figura 2.22](#)) también están formadas por una línea de respuesta, seguida de múltiples cabeceras, y separadas del cuerpo de la respuesta (que puede ser un contenido en formato texto o binario) mediante una línea en blanco (\r\n).

```
HTTP/1.1 200 OK\r\n
Server: ECS (mic/9AF8)\r\n
Date: Wed, 06 Jan 2021 16:22:43 GMT\r\n
Etag: "3147526947+ident"\r\n
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT\r\n
Age: 495344\r\n
Cache-Control: max-age=604800\r\n
Expires: Wed, 13 Jan 2021 16:22:43 GMT\r\n
X-Cache: HIT\r\n
Content-Type: text/html; charset=UTF-8\r\n
Content-Length: 1256\r\n
\r\n
<!doctype html>
<html>
<head>
<title>Example Domain</title>

<meta charset="utf-8" />
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
...
</head>
<body>
...
</body>
</html>
```

**Figura 2.22 – Ejemplo de respuesta HTTP**

La línea de respuesta (en negrita) está formada por la versión HTTP soportada por el servidor, un código numérico y un mensaje de error (todo ello separado mediante espacios).

Los códigos y mensajes de error HTTP más comunes son:

- **200 OK:** La petición ha tenido éxito, el recurso solicitado está incluido en el cuerpo del mensaje.
- **302 Moved Permanently:** El recurso solicitado ya no se encuentra en esta localización. La cabecera Location: indica la nueva URL donde se encuentra el recurso.
- **304 Not Modified:** El recurso de la petición condicional no ha cambiado respecto al versión (If-None-Match) o fecha (If-Modified-Since) indicadas.
- **400 Bad Request:** La petición no se puede procesar porque tiene un error de sintaxis.
- **401 Unauthorized:** El recurso solicitado requiere autenticación. El mecanismo de autenticación requerido por el servidor se indica en la cabecera WWW-Authenticate.
- **404 Not Found:** El recurso solicitado no existe.

- **500 Internal Server Error:** La petición no se ha podido procesar por un error interno del servidor.

Las cabeceras más habituales que pueden aparecer en una respuesta HTTP son:

- **Age:** Número de segundos que lleva el objeto en la cache.
- **Allow:** Métodos soportados por el recurso, solicitados con el método OPTIONS (e.g. "GET, HEAD").
- **Cache-control:** Directivas para el control de las caches inferiores (e.g. "private, max-age=3600").
- **Connection:** Permite controlar la conexión TCP subyacente (e.g. "close")
- **Content-Encoding:** Mecanismo de compresión del cuerpo del mensaje (e.g. "gzip")
- **Content-Language:** Lenguaje empleado en el recurso (e.g. "es-es").
- **Content-Length:** Longitud en bytes del cuerpo del mensaje.
- **Content-Range:** Rango de bytes descargado del recurso y tamaño total del mismo (e.g. "bytes 1024-2048/4096").
- **Content-Type:** Tipo MIME y codificación del cuerpo del mensaje (e.g. "text/html; charset=UTF-8").
- **Date:** Fecha cuando se envió la respuesta (e.g. "Tue, 24 Nov 2015 13:07:31 GMT").
- **ETag:** Identifica la versión del recurso, por ejemplo, realizando un hash de su contenido (e.g. "737060cd8c284d8af7ad3082f209582d").
- **Expires:** Fecha cuando el recurso dejará de ser válido (e.g. "Tue, 24 Nov 2015 14:00:00 GMT").
- **Last-Modified:** Fecha de la última modificación realizada (e.g. "Tue, 24 Nov 2015 13:00:00 GMT")
- **Location:** URL con la nueva localización del objeto (para respuestas con código 302 *Moved Permanently*).
- **Refresh:** Redirige el cliente a una nueva URL tras un cierto número de segundos (e.g. "5; url=https://www.example.com")
- **Server / X-Powered-By:** Identifican el servidor HTTP y las librerías que usa (e.g. "Server: Apache/2.4.1 (Unix)", "X-Powered-By: PHP/5.4.0").
- **Set-Cookie:** Establece una cookie en el cliente, que éste debería enviar de vuelta cada vez que se conecte al servidor (e.g. "SID=31d4d96e407aad42; max-age=86400; domain=.example.com; path=/; secure; httpOnly")
- **WWW-Authenticate:** Mecanismo de autenticación requerido para acceder al recurso (e.g. "Basic").

Las cabeceras que comienzan por **X-** (e.g. X-Powered-By) no están estandarizadas, lo que permite a las aplicaciones definir cabeceras nuevas.

HTTP es un protocolo sin estado. Eso quiere decir que el servidor puede procesar las peticiones de los clientes de manera independiente, sin que exista alguna noción

a nivel HTTP de que un conjunto de peticiones pertenezca a una misma “sesión de navegación”. De hecho, cuando los recursos de un servidor requieren autenticación HTTP, todas las peticiones del cliente tienen que incluir la cabecera `Authorization` con sus credenciales. Sin embargo, muchas aplicaciones web por encima de HTTP sí necesitan un concepto de sesión, por ejemplo, para comprobar si un cliente ya se ha autenticado, o para recordar los productos que ha introducido en el carro de la compra. Para ello, HTTP proporciona un mecanismo de *Cookies* a la aplicación, que permite al Servidor enviar al Cliente (en la cabecera `Set-Cookie`) una o más Cookies con información opaca, y que el cliente debe devolver la próxima vez que envíe una petición al mismo servidor (en la cabecera `Cookie`). De este modo, a un cliente autenticado se le puede enviar una *Cookie* de sesión (con un valor aleatorio y/o firmado por el servidor) que sirve de índice en la base de datos donde se almacena la información de ese cliente, o simplemente incluir toda esa información en la propia *Cookie* (que puede ocupar hasta 4 KiB), idealmente de una manera cifrada y firmada.

### 2.10.1. Proxies HTTP

Un aspecto muy interesante de HTTP es que, a pesar de ser un protocolo cliente-servidor, permite que las comunicaciones atravesen cero o más *proxies*. Hace unos años, los *proxies* web se desplegaban típicamente en el lado de los clientes, para que estos almacenaran en una cache los contenidos más populares, y ahorrar así capacidad en el acceso a Internet de las organizaciones. Sin embargo, la gran cantidad contenido dinámico y personalizado de la web 2.0 hace que hoy en día la mayoría de los ***proxies* del lado cliente** se desplieguen por motivos de seguridad: para evitar que los equipos de los empleados salgan directamente a Internet, y para monitorizar los contenidos que intercambian (e.g. con un antivirus o para limitar el acceso a sitios web maliciosos o con contenidos cuestionables).

Normalmente los usuarios son conscientes de la existencia de los *proxies* del lado cliente, y configuran sus navegadores para que los utilicen para salir a Internet. En ese caso, el navegador web establece una conexión TCP con el *proxy* configurado (por defecto usando el puerto TCP/8080), y le envía todas las peticiones HTTP que genera, tal cual haría con el servidor web, con la salvedad que en lugar de enviar URLs relativas (e.g. “/index.html”) en la línea de petición, se envían URLs absolutas (e.g. <http://www.example.com:80/index.html>) para indicarle al *proxy* el servidor web al que debe reenviar la petición. El *proxy* establece entonces una conexión TCP con el servidor web y le reenvía la petición del cliente (añadiendo una cabecera `via` para indicar que ha pasado por él). Cuando el servidor envíe la respuesta al *proxy*, éste se la reenviará al navegador. Por lo tanto, cuando se utiliza un *proxy* siempre hay dos conexiones TCP: una entre el navegador y el *proxy*, y otra entre el *proxy* y el servidor web, por lo que el *proxy* se encarga de reenviar las peticiones y respuestas entre ellos (tras analizar los contenidos, y cacheándolos si es posible).

El comportamiento de los *proxies* HTTPS es bastante diferente. Aunque las peticiones HTTPS también pueden reenviarse a través de un *proxy*, la sesión TLS se sigue estableciendo entre el navegador y el servidor web, por lo que un *proxy* HTTPS estándar solo puede ver los datos cifrados de la sesión TLS, por lo que no puede analizar o cachear los contenidos intercambiados. Para utilizar un **proxy HTTPS**, el navegador establece una conexión TCP con él y le envía una petición especial con el método CONNECT, indicando el nombre de dominio y el puerto TCP del servidor. El *proxy* HTTPS establece una conexión TCP con el destino, pero en este caso no le reenvía la petición CONNECT del cliente, sino que simplemente responde al cliente con una respuesta 200 OK, y a partir de ese momento simplemente reenvía los bytes que reciba del cliente y el servidor entre ellos. De esta forma el cliente puede iniciar la negociación de una sesión TLS con el servidor, incluso si no están conectados directamente sino a través del *proxy*. Una vez establecida la sesión TLS, el cliente y el servidor pueden intercambiarse mensajes HTTP de manera segura. En el **Capítulo 7** estudiaremos técnicas para intentar descifrar y capturar el tráfico de sesiones TLS, aunque esto solo es posible si contamos con la colaboración de uno de los extremos de la comunicación.

A veces, en lugar de emplear un *proxy* cliente (que requiere configurar todos los navegadores y resto de aplicaciones para que lo usen), las organizaciones despliegan en su salida a Internet los conocidos como **proxies transparentes**, que actúan como *proxies* cliente, aunque los navegadores no son conscientes de que los están usando. Funcionan de la siguiente forma: los navegadores no tienen ningún *proxy* configurado, así que asumen que pueden comunicarse directamente con el servidor web final, por lo que resuelven su nombre DNS e intentan establecer una conexión TCP contra su IP. Sin embargo, antes de salir a Internet, el *proxy* transparente intercepta esos paquetes (para ello el *proxy* debe estar en la ruta de salida a Internet de la organización) y responde a los mismos como si fuera el servidor destino (i.e. falsificando su dirección IP). Así que la conexión TCP del cliente realmente se establece con el *proxy* transparente, y éste establece otra conexión TCP con el servidor web final (utilizando la IP y el puerto destino que ha enviado el cliente). El navegador envía entonces sus peticiones HTTP al *proxy* (con URLs relativas puesto que piensa que está hablando con el servidor), el *proxy* la reenvía al servidor, y luego hace lo mismo con la respuesta (tras analizar los contenidos y cachearlos como un *proxy* tradicional). Por lo tanto, el comportamiento final es exactamente el mismo que con un *proxy* cliente explícito, pero sin que los navegadores (y a veces los usuarios) sean conscientes que sus comunicaciones están siendo monitorizadas. Las sesiones HTTPS también funcionan a través de un *proxy* transparente, porque este actúa como un *proxy* a nivel TCP, aunque, como en el caso del proxy HTTPS explícito, normalmente solo ve el tráfico cifrado y no puede monitorizar la sesión.

En los últimos años también se ha popularizado el uso de los *proxies* en el lado del servidor, denominados **proxies inversos**, que se utilizan principalmente como *firewalls* a nivel de aplicación (i.e. WAF – *Web Application Firewall*), filtrando peticiones maliciosas de los clientes (e.g. evitando ataques de *path transversal*, *SQL injection* o *Cross Site Scripting*), así como para reducir la carga de los servidores web finales, por ejemplo, funcionando como caches de contenidos dinámicos, como terminadores de sesiones TLS, o como balanceadores de carga delante de una granja de servidores web. Normalmente los clientes tampoco son conscientes de la existencia de los *proxies* inversos (a no ser por la cabecera *Via*), porque típicamente se comportan como si fueran el servidor real (i.e. su dirección IP se publica en el DNS asociada al nombre del servidor). La única diferencia es que normalmente solo son capaces de responder a las peticiones HTTP que les envían los clientes si ya las tienen en la cache. De lo contrario, se las reenvían al servidor web real y esperan su respuesta para reenviársela al cliente. Los *proxies* inversos normalmente sí son capaces de inspeccionar el tráfico HTTPS, porque tienen el certificado digital y la clave privada del servidor por el que se hacen pasar, de forma que son los que terminan la sesión TLS con el cliente, y por tanto pueden analizar todo el tráfico intercambiado.