El Protocolo HTTP

El protocolo **HTTP** (HyperText Transfer Protocol) es un protocolo de transferencia de hipertexto que sigue el modelo **cliente-servidor** y establece las normas para el intercambio de información

contenida en las páginas web.

Es un protocolo de la **capa de aplicación** del que se utiliza prácticamente para **transmitir la mayoría de los archivos y datos de Internet**, ya sean archivos HTML, imágenes, resultados de consultas u otras cosas.

Es un protocolo **sin estado**. El servidor no guarda un historial con las peticiones que ha realizado anteriormente un cliente.

> Si HTTP es un protocolo sin estado, ¿cómo sabe un servidor web de una tienda online que he añadido un producto al carrito? Pues con otros mecanismos como las **cookies**.
> Las **cookies** son pequeños fragmentos de texto que los sitios web que visitas envían al navegador. Permiten que los sitios web recuerden información sobre tu visita, lo que puede hacer que sea más fácil volver a visitar los sitios y hacer que estos te resulten más útiles.
> Gracias a las cookies podemos personalizar sitios web, aunque también tienen implicaciones en cuanto a la privacidad.

El cliente envía **mensaje de petición** y el servidor responde con **mensajes de respuesta**, codificados en **ASCII**, es decir, en texto plano.

El servidor HTTP utiliza por defecto el **puerto 80**. Si estamos hablando de HTTPS (HTTP cifrado), por defecto el puerto es el **443**.

En principio, las conexiones entre cliente y servidor son conexiones **TCP**, puesto que **se transmiten ficheros**. Ya veremos que en versiones posteriores de HTTP esto cambia cambia con el protocolo QUIC que se sustenta sobre **UDP**. QUIC es un protocolo desarrollado por Google que se ha adoptado como estándar cuyo objetivo es agilizar la web.

Funcionamiento básico del protocolo

¿Qué ocurre cuando abrimos el navegador y tecleamos la URL de una página web?

1. El navegador solicita al **resolver o servidor DNS** que resuelva el nombre del servidor especificado en la URL.
2. Obtenida la **IP** del servidor HTTP, se establece una **conexión TCP entre cliente y servidor HTTP**.
3. Se envía un **mensaje de petición** del documento web al servidor (u otra acción) y éste devuelve su contenido en un **mensaje de respuesta**.

4. Si el documento web incluye elementos adicionales como imágenes, hay un proceso de petición/respuesta por cada recurso.

Mensajes HTTP

Los **mensajes HTTP** son textos codificados en **ASCII** y constan de 4 campos:

1. **Línea de petición o de respuesta**: Contiene la información principal sobre la petición o la respuesta.
2. **Cabeceras** (o encabezados o headers): Contienen **información adicional** sobre las opciones relativas al mensaje. Una línea de encabezado por cada opción que se especifique.
3. **Una línea vacía**, generada con un ENTER. Actúa como separador entre las líneas de encabezados y el cuerpo del mensaje.
4. **Cuerpo del mensaje**: Este campo es **opcional**. En los mensajes de respuesta se utiliza para enviar el contenido del recurso solicitado. Puede incluír contenidos de un archivo, de una consulta, datos binarios, etc.

Mensajes de petición

Un **mensaje de petición** es un mensaje que envía el **cliente** al **servidor**.

En un mensaje de petición, la **línea de petición** contiene tres datos separados por un espacio:

1. **Método de la petición**: El **método** de la petición indica la **acción que se pretende realizar sobre el recurso**. Los métodos más utilizados son **GET** y **POST** y los veremos más adelante.
2. **Dirección URL del recurso**: La dirección URL del recurso especifica **la ruta del recurso** dentro del directorio raíz del sitio web (aunque no tiene porqué ser siempre así). Por ejemplo, */imagenes/foto-playa.png* indica que el recurso sobre el que se va a realizar la acción es el archivo *foto-playa.png* y que se encuentra dentro de la carpeta *imagenes/*.
3. **Versión del protocolo HTTP**. El protocolo HTTP ha ido evolucionando. Veremos cómo lo ha hecho y qué mejoras se han producido.

A continuación se envía el **cuerpo del mensaje** (body). Si es una petición **GET**, el cuerpo estará vacío. Si es una petición **POST**, en el cuerpo viajarán datos como por ejemplo, los campos de un formulario.

Ejemplo 1: Petición GET simple

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

Esta es una petición **GET** que manda un navegador para obtener la página de inicio "**/**" del sitio **https://developer.mozilla.org**. También se especifica la versión del protocolo que el cliente pretende utilizar en la comunicación, en este caso la **HTTP/1.1**.

Además, junto a la petición, se envían **dos cabeceras**:

- La cabecera **Host:developer.mozilla.org** que indica el sitio que queremos visitar. Esto es útil por si el servidor almacena varias páginas web de varios sitios (ya veremos su importancia al hablar de **Virtual Hosts**).
- La cabecera **Accept-Language:fr** en la que el navegador indica que, si es posible, se le devuelva la página en francés. Si no es posible, el servidor le enviará la página en el idioma por defecto.

Hay una línea vacía al final de la petición, que separa las cabeceras del bloque de datos. Pero en este caso, el **bloque de datos** está vacío al ser una petición **GET**.

Ejemplo 2: Petición POST con envío de formulario

```
POST /contact_form.php HTTP/1.1
Host: developer.mozilla.org
Content-Length: 64
Content-Type: application/x-www-form-urlencoded

name=Juan%20Garcia&request=Envieme%20uno%20de%20sus%20catalogos
```

En esta petición el navegador utiliza el **método POST** para enviar un formulario que será procesado por el script **/contact_form.php**. También especificamos la versión de HTTP que usamos, la **HTTP/1.1**.

El navegador envia tres **cabeceras**:

- La cabecera **Host:developer.mozilla.org** que indica el sitio que queremos visitar.
- La cabecera **Content-Length:64** que indica el tamaño del contenido del cuerpo de la petición en bytes.

- La cabecera **Content-Type:application/x-www-form-urlencoded**, que indica el tipo de contenido que se manda en el cuerpo de la petición. En este caso, datos de formulario codificados en URL Encoded.

El **cuerpo de la petición** es:

```
name=Juan%20Garcia&request=Envieme%20uno%20de%20sus%20catalogos
```

Está codificado en URL Encoded, tal como especificamos en el tipo y contiene dos pares de variables y valores:

- Variable **name** con valor de "Juan García".
- Variable **request** con valor "Envieme uno de sus catalogos".

Mensajes de respuesta

Un **mensaje de respuesta** es un mensaje que envía el **servidor** al **cliente** en respuesta a un **mensaje de petición**.

En un mensaje de respuesta, la **línea de estado** contiene tres campos separados por espacios:

1. **Versión de HTTP**.
2. **Código de estado**. Se explicarán a continuación.
3. **Razón o frase**: Explica el significado del código de estado.

Tras la **línea de estado**, vendrán **una o más cabeceras**, una **línea en blanco** y, opcionalmente, un **cuerpo de mensaje**.

Códigos de estado

Los **códigos de estado** están formados por tres dígitos donde el primero de ellos define la **clase de la respuesta**.

- **1XX Código informativo:** Significa que se recibió la solicitud y el proceso continúa.
- **2XX Éxito:** La acción fue recibida, entendida y aceptada con éxito.
- **3XX Redirección:** Significa que hay que tomar medidas adicionales para completar la solicitud.
- **4XX Error del cliente:** Significa que la solicitud contiene una sintaxis incorrecta o no se puede cumplir.
- **5XX: Error del servidor:** Significa que el servidor no cumplió con una solicitud aparentemente válida.

Ejemplos de respuestas:

Respuesta a una solicitud de una página llamada **hello.htm**:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Respuesta en la que se muestra un **Error 404** ya que el cliente solicitó una página que no existe:

```
HTTP/1.1 404 Not Found
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 230
Connection: Closed
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>404 Not Found</title>
</head>
<body>
   <h1>Not Found</h1>
   <p>The requested URL /t.html was not found on this server.</p>
</body>
</html>
```

Respuesta en la que el servidor encontró una versión de HTTP incorrecta en la solicitud HTTP:

```
HTTP/1.1 400 Bad Request
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 230
Content-Type: text/html; charset=iso-8859-1
Connection: Closed

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>400 Bad Request</title>
</head>
<body>
   <h1>Bad Request</h1>
   <p>Your browser sent a request that this server could not understand.</p>
   <p>The request line contained invalid characters following the protocol string.</p>
</body>
</html>
```

Cabeceras HTTP

Las **cabeceras** (headers) HTTP permiten al cliente y al servidor enviar **información adicional** junto a una petición o respuesta.

Una cabecera está compuesta por su nombre (no sensible a las mayúsculas) seguido de ':' y a continuación de su valor, sin saltos de línea. Los espacios en blanco a su izquierda son ignorados.

```
nombre:valor
```

- Lista de cabeceras HTTP

Existen varios tipos de encabezados (headers):

1. Encabezados de solicitud

Los encabezados de solicitud contienen más información sobre el recurso que se va a recuperar o sobre el cliente que solicita el recurso.

2. Encabezados de respuesta

Los encabezados de respuesta contienen información adicional sobre la respuesta, como su ubicación o sobre el servidor que la proporciona.

3. Encabezados de representación

Los encabezados de representación contienen información sobre el cuerpo del recurso, como su tipo MIME o la codificación/compresión aplicada.

4. Encabezados de carga útil

Los encabezados de carga útil contienen información independiente de la representación sobre los datos de carga útil, incluida la longitud del contenido y la codificación utilizada para el transporte.

```
GET /home.html HTTP/1.1
Host: developer.mozilla.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://developer.mozilla.org/testpage.html
Connection: keep-alive
Upgrade-Insecure-Requests: 1
If-Modified-Since: Mon, 18 Jul 2016 02:36:04 GMT
If-None-Match: "c561c68d0ba92bbeb8b0fff2a9199f722e3a621a"
Cache-Control: max-age=0
```

En esta petición GET, hay encabezados de solicitud **Accept-** en los que el cliente indica al servidor cómo debería de adaptarse su respuesta.

También hay encabezados de representación, como por ejemplo **Content-Type**.

Métodos HTTP

	La petición tiene cuerpo	La respuesta satisfactoria tiene cuerpo	Seguro (no altera datos en el servidor)	Idempotente	Cacheable	Permitido en formularios
GET	NO	SÍ	SÍ	SÍ	SÍ	SÍ
HEAD	NO	NO	SÍ	SÍ	SÍ	NO
POST	SÍ	SÍ	NO	NO	NO (salvo excepciones)	SÍ
DELETE	QUIZÁ	QUIZÁ	NO	SÍ	NO	NO
CONNECT	NO	SÍ	NO	NO	NO	NO
TRACE	NO	NO	SÍ	SÍ	NO	NO
PATCH	SÍ	SÍ	NO	NO	NO	NO

El protocolo **HTTP** define un conjunto de métodos de petición para **realizar la acción que se desea** sobre un recurso. Son también conocidos como "verbos" (**HTTP verbs**).

- **GET**: Lee una web.
- **HEAD**: Lee las cabeceras de una web.
- **POST**: Añade una web.
- **PUT**: Almacena una web.
- **DELETE**: Borra una web.
- **TRACE**: Hace "eco" de la petición.
- **CONNECT**: Conecta a través de un proxy.
- **OPTIONS**: Consulta los métodos de una web.
- **PATCH**: Realiza modificaciones parciales a una web.

Método GET

El **método GET** solicita una representación de un recurso especificado. En otras palabras, **solicita que se le envíe un recurso** como puede ser una página web, una imagen, etc. Las peticiones que utilizan el método GET **sólo deben recuperar datos**.

El **método GET** es el más utilizado.

```
GET /images/logo.png HTTP/1.1
```

En esta **petición GET** se solicita el recurso **/images/logo.png** y se indica que se utiliza la versión **HTTP/1.1** para la comunicación.

Método HEAD

El **método HEAD** solicita una respuesta idéntica a **GET** pero **sin el cuerpo de la respuesta**, es decir, **sin solicitar que se le envíe el el recurso**. Por lo tanto, **HEAD** Es útil para recuperar **meta-informacón** escrita en los encabezados (headers) de la respuesta sin tener que transportar todo el contenido.

```
HEAD /index.html
```

Con esta descripción de HEAD, su cometido puede resultar confuso. Imaginemos que visitamos una página web www.ejemplo.com y nuestro navegador web se descarga la página principal (index.html) y nos la muestra, y salva una copia en su caché (un directorio donde va guardando las webs que vamos visitando). Cinco minutos más tarde, volvemos a visitar la página web, pero nuestro navegador sabe que ya la hemos visitado. **¿Para qué va a descargarla de nuevo si quizá no haya cambiado desde la última vez que la visitamos?** Pues a través del método HEAD el navegador puede recuperar metadatos (de las cabeceras) tales como **la última vez que se modificó la web**. Si la fecha coincide con la que el navegador tiene almacenada en su caché, nos muestra la web almacenada en caché más rápido y evita volver a descargar archivos. Y nosotros visualizaremos la web más rápido y el servidor no nos tendrá que mandar una web que ya tenemos. Todos salimos ganando, a cambio, claro, de que los clientes (navegadores) utilicen algo de espacio en disco para estos archivos temporales.

Existen otros métodos para cachear webs.

Método POST

El **método POST** se utiliza para enviar datos al servidor, como por ejemplo, un **formulario web** que hayamos cubierto.

El método post causa a menudo un **cambio en el estado o efectos secundarios** en el servidor.

La URL que se solicita normalmente no es un recurso, por lo general es un **script** o programa al que se envían **los datos para que sean procesados**.

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

variable1=valor1&variable2=valor2
```

Método GET vs Método POST

Los navegadores pueden enviar datos a un servidor (como los campos de un formulario) utilizando los métodos **GET** y **POST**.

Los métodos **GET**, **POST** y **HEAD** son los **más comunes**.

- **GET**
  - En una **petición GET** los datos se añaden a la propia **URL** en pares de **nombre y valor**.
  - La longitud de la **URL** está limitada a **3000** caracteres.
  - **Nunca** se debe utilizar **GET** para enviar datos sensibles (como contraseñas), puesto que serían visibles en la URL.
  - **GET** está bien para enviar datos no seguros, como **términos de búsqueda** ya que podrían **añadir a los marcadores** o **compartir el enlace**.
- **POST**
  - En una **petición POST** los datos se envían dentro del **cuerpo de la petición HTTP**.
  - Los **datos** no se ven en la **URL**, por lo que es el método adecuado para enviar datos sensibles como **usuario y clave** de un formulario de **login**.
  - No existe un límite de tamaño.
  - Las peticiones POST **no se pueden añadir a marcadores** ya que los datos no van en la propia URL.

Veámoslo con un ejemplo. Creamos una página web que tiene dos tipos de formularios:

- Un **formulario de búsqueda**.
- Un **formulario de login**.

En el caso de un **formulario de búsqueda**, enviaremos el formulario por GET, con los valores en la propia URL.

Los datos de búsqueda no son sensibles y además así permitimos que el usuario pueda añadir su búsqueda a marcadores, o compartirla.

Un ejemplo de este tipo de formulario en HTML sería:

```
<form action="/action_page.php" method="get">
  <label for="buscar">Término de búsqueda:</label>
  <input type="text" id="buscar" name="buscar"><br><br>
  <input type="submit" value="¡Buscar!">
</form>
```

En el caso de un **formulario de LOGIN**, donde el usuario va a mandar su nombre y contraseña, no tiene sentido utilizar GET. Se utiliza **POST** y así **el usuario y la contraseña irán en el cuerpo de la petición y no serán visibles en la URL**.

Obviamente, si estamos utilizando HTTP y no HTTPS (con cifrado), si alguien captura el tráfico se haría con las credenciales ya que viajarían en texto plano. Por eso es importante utilizar **HTTPS**.

```
<form  action="https://www.codelr.com/login/" method="post">

    <label  for="username">Username:</label>
    <input class="userbox" type="text" name="username" required="required" /><br />
    <label for="password">Password:</label>
    <input type="text" name="password" required="required" />
    <input  class="button" type="submit" value="submit" />
</form>
```

En la etiqueta de HTML **form** especificamos el método: **GET** o **POST** dentro del atributo **method**. Pero esto ya deberías saberlo de Lenguaje de Marcas 😃

Método PUT

El **método PUT** reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición, es decir, **sube o actualiza un recurso especificado con lo que haya en el cuerpo de la petición** (body).

PUT es parecido a POST, excepto por su **diferencia semántica**. Mientras en POST la URL identifica el recurso que manejará los datos enviados (como un script que procese el formulario), con PUT la URL identifica al **recurso que se creará o reemplazará con el contenido del cuerpo de la petición**.

Ya se ha comentado que el **método POST** causa un **efecto secundario** o una **modificación** en el servidor. Por ejemplo, una petición POST puede añadir un mensaje a un foro. La diferencia con PUT es que **PUT es un método idempotente** y llamarlo una o más veces de forma sucesiva tiene **el mismo efecto**.

- **Si repetimos una petición POST**, es posible que dupliquemos el mismo mensaje en un foro. Una sucesión de peticiones POST idénticas pueden tener efectos adicionales.
- **Si repetimos una petición PUT**, un recurso se modifica por el contenido del cuerpo de la petición. Como el cuerpo de la petición no varía, no se producen efectos adicionales.

Un ejemplo de petición PUT sería:

```
PUT /new.html HTTP/1.1
Host: example.com
Content-type: text/html
Content-length: 16

<p>New File</p>
```

Y las posibles respuestas del podrían ser:

- Si el recurso de destino no tiene una representación actual y la solicitud PUT crea una correctamente, el servidor debe informar al cliente enviando una respuesta **201** (creado):

```
HTTP/1.1 201 Created
Content-Location: /new.html
```

- Si el recurso de destino tiene una representación actual y esa representación se modifica correctamente de acuerdo con el estado de la representación adjunta, entonces el servidor de origen debe enviar una respuesta **200** (OK) o **204** (Sin contenido) para indicar la finalización exitosa de la solicitud:

```
HTTP/1.1 204 No Content
Content-Location: /existing.html
```

Método DELETE

El **método DELETE** borra el recurso especificado.

Obviamente muchos servidores no aceptan peticiones **PUT** o **DELETE**, o requieren algún tipo de identificación, si no cualquiera podría borrar o modificar las páginas webs de otras personas.

```
DELETE /file.html HTTP/1.1
Host: example.com
```

Si el **método DELETE** tiene éxito, el servidor puede responder:

- Con un **código de estado 202 (Aceptado)** si es probable que la acción tenga éxito pero aún no se haya promulgado.
- Con un **código de estado 204 (Sin contenido)** si la acción se ha llevado a cabo y no se debe proporcionar más información.
- Con un **código de estado 200 (OK)** si la acción se ha realizado y el mensaje de respuesta incluye una representación que describe el estado.

Por ejemplo:

```
HTTP/1.1 200 OK
Date: Wed, 21 Oct 2015 07:28:00 GMT

<html>
  <body>
    <h1>File deleted.</h1>
  </body>
</html>
```

Método CONNECT

El **método CONNECT** inicia la comunicación en dos caminos con la fuente del recurso solicitado. Puede ser utilizado para **abrir una comunicación túnel**.

El método CONNECT puede ser usado para acceder a sitios web que usan SSL (HTTPS). El cliente realiza la petición al Servidor Proxy HTTP para establecer una conexión tunel hacia un destino deseado. Entonces el servidor Proxy procede a realizar la conexión en nombre del cliente y una vez establecida la conexión el servidor Proxy envía los datos desde y hacia el cliente.

Es un método de salto entre servidores.

Algunos servidores proxy pueden necesitar autorización para crear túneles. Para eso se envía el encabezado **Proxy-Authorization**.

```
CONNECT server.example.com:80 HTTP/1.1
Host: server.example.com:80
Proxy-Authorization: basic aGVsbG86d29ybGQ=
```

Método OPTIONS

El método **HTTP OPTIONS** solicita los comandos que se pueden utilizar para una URL o para un servidor.

Petición:

```
OPTIONS /index.html HTTP/1.1
OPTIONS * HTTP/1.1
```

Posible respuesta del servidor:

```
HTTP/1.1 204 No Content
Allow: OPTIONS, GET, HEAD, POST
Cache-Control: max-age=604800
Date: Thu, 13 Oct 2016 11:45:00 GMT
Server: EOS (lax004/2813)
```

La respuesta contiene una cabecera **Allow** con los **métodos permitidos**.

Podemos utilizar la herramienta de consola **curl** para ver qué métodos soporta un servidor:

```
curl –X OPTIONS https://example.org –i
```

Método TRACE

```
TRACE /index.html
```

El método **HTTP TRACE** solicita al servidor que envíe en el cuerpo de su mensaje de respuesta los datos que reciba del mensaje de solicitud: **prueba de bucle**.

Se utiliza con **fines de comprobación y diagnóstico**.

Método PATCH

El método HTTP PATCH aplica modificaciones parciales a un recurso.

```
PATCH /file.txt HTTP/1.1
Host: www.example.com
Content-Type: application/example
If-Match: "e0023aa4e"
Content-Length: 100

[description of changes]
```

A diferencia de PUT, **PATCH no es idempotente**. Peticiones idénticas sucesivas pueden tener efectos diferentes. Sin embargo es posible emitir peticiones PATCH de tal forma que sí sean idempotentes.

PATCH al igual que POST **puede provocar efectos secundarios** en otros recursos.

Para averiguar si un servidor soporta PATCH, puede notificar su compatibilidad al añadirlo a la lista en el header: Allow o Access-Control-Allow-Methods.

Otra indicación implícita es la presencia de la cabecera Accept-Patch.

Una posible respuesta a un mensaje **PATCH** sería:

```
HTTP/1.1 204 No Content
Content-Location: /file.txt
ETag: "e0023aa4f"
```

- Más información sobre Métodos HTTP

Evolución del protocolo HTTP

HTTP/0.9 - El protocolo de una línea

HTTP/0.9 era extremadamente simple: las solicitudes constaban de una sola línea y comenzaban con el único método posible **GET** seguido de la ruta al recurso. No se incluyó la URL completa, ya que el protocolo, el servidor y el puerto no eran necesarios una vez conectado al servidor.

Ejemplo de petición:

```
GET /mypage.html
```

Ejemplo de respuesta:

```
<html>
  A very simple HTML page
</html>
```

La respuesta consistía en simplemente el archivo solicitado.

No había **cabeceras** y por ello **solo se podían transmitir archivos HTML**. No había **códigos de estado o de error**, por lo que si había un problema, un archivo **HTML** específico era generado con la descripción del problema.

HTTP/1.0 - Creando extensibilidad

HTTP/0.9 era muy limitado, pero los navegadores y servidores rápidamente lo hicieron más versátil:

- La **información de versión** se envió dentro de cada solicitud (HTTP/1.0 se agregó a la línea GET).
- También se envió una línea de **código de estado** al comienzo de una respuesta. Esto permitió que **el propio navegador reconociera el éxito o el fracaso de una solicitud y adaptara su comportamiento** en consecuencia. Por ejemplo, actualizar o usar su caché local de una manera específica.
- El concepto de **encabezados HTTP se introdujo tanto para solicitudes como para respuestas**. Se podían transmitir metadatos y el protocolo se volvió extremadamente **flexible y extensible**.
    - Los documentos que no sean archivos HTML simples podrían transmitirse gracias al encabezado **Content-Type** que especifica el tipo **MIME** del recurso solicitado.

Ejemplo de petición:

```
GET /mypage.html HTTP/1.0
User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)
```

Ejemplo de respuesta:

```
200 OK
Date: Tue, 15 Nov 1994 08:12:31 GMT
Server: CERN/3.0 libwww/2.17
Content-Type: text/html
<HTML>
A page with an image
  <IMG SRC="/myimage.gif">
</HTML>
```

HTTP/1.1 – El protocolo estandarizado

En paralelo a las diversas implementaciones de HTTP/1.0 se estaba creando una versión estandarizada, la **HTTP/1.1** que se publicó a principios de **1997**, solo unos meses después de **HTTP/1.0**.

HTTP/1.1 aclaró ambigüedades e introdujo numerosas mejoras:

- Se podía **reutilizar una conexión**, lo que ahorraba tiempo. Ya no era necesario abrirla varias veces para mostrar los recursos incrustados en el único documento original.
- Se agregó canalización (**pipelining**). Esto permitió enviar una segunda solicitud antes de que se transmitiera completamente la respuesta a la primera. Esto **redujo la latencia de la comunicación**.
- También se admitieron **respuestas fragmentadas**.
- Se introdujeron **mecanismos de control de caché adicionales**.
- Se introdujo la **negociación de contenido**, incluido el idioma, la codificación y el tipo. Un cliente y un servidor ahora podrían acordar qué contenido intercambiar.
- Gracias a la cabecera **Host**, se pueden alojar diferentes dominios en la **misma dirección IP**.

Petición:

```
GET /en-US/docs/Glossary/Simple_header HTTP/1.1
Host: developer.mozilla.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://developer.mozilla.org/en-US/docs/Glossary/Simple_header
```

Respuesta:

```
200 OK
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Wed, 20 Jul 2016 10:55:30 GMT
Etag: "547fa7e369ef56031dd3bff2ace9fc0832eb251a"
Keep-Alive: timeout=5, max=1000
Last-Modified: Tue, 19 Jul 2016 00:59:33 GMT
Server: Apache
Transfer-Encoding: chunked
Vary: Cookie, Accept-Encoding

(content)
```

Petición:

```
GET /static/img/header-background.png HTTP/1.1
Host: developer.mozilla.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://developer.mozilla.org/en-US/docs/Glossary/Simple_header
```

Respuesta:

```
200 OK
Age: 9578461
Cache-Control: public, max-age=315360000
Connection: keep-alive
Content-Length: 3077
Content-Type: image/png
Date: Thu, 31 Mar 2016 13:34:46 GMT
Last-Modified: Wed, 21 Oct 2015 18:27:50 GMT
Server: Apache

(image content of 3077 bytes)
```

HTTP/1.1 fue publicado en la RFC 2068 en enero de 1997.

La extensibilidad de **HTTP/1.1** hacía posible crear nuevas cabeceras y métodos. El protocolo fue refinado en la RFC 2616 publicada en junio de 1999 y en la RFC 7230 - RFC 7235.

¡Esta versión fue estable más de **15 años**! Un montón de tiempo.

HTTP/2 - Un protocolo para un mayor rendimiento

A medida que pasaron los años las páginas web se fueron volviendo mucho más complejas. Hoy en día, muchas webs son **aplicaciones por derecho propio**. Su tamaño aumentó a medida que se les añadía mayor contenido multimedia, y también creció el tamaño de los **scripts** o programas que la dotan de interactividad.

Este crecimiento de las páginas webs supuso un reto, puesto que se aumentaron significativamente las solicitudes HTTP y esto creó más complejidad y sobrecarga para las conexiones HTTP/1.1.

Para solucionar este problema, Google implementó un protocolo experimental llamado **SPDY** a principios de la década de 2010. Este protocolo atrajo el interés de desarrolladores de navegadores y servidores, y resolvió el problema de datos diplucados, sirviendo como base para el **protocolo HTTP/2**.

> **Google** desarrolla protocolos de mejora porque su negocio está en Internet y quiere que los usuarios puedan utilizar mejor sus servicios.

El protocolo HTTP/2 se diferencia de HTTP/1.1 en algunos aspectos:

- Es un **protocolo binario** en lugar de un **protocolo de texto**. Las peticiones y respuestas ya no se envían en texto plano, sino que son flujos de bytes (streams). De esta forma se pueden implementar **técnicas de optimización** mejoradas.
- Es un **protocolo multiplexado**, es decir, se pueden realizar **solicitudes paralelas** a través de la misma conexión, eliminando las restricciones del protocolo HTTP/1.x.
- **Comprime encabezados**, ya que a menudo son similares entre un conjunto de solicitudes, esto elimina la duplicación y la sobrecarga de los datos transmitidos.
- Permite que **un servidor llene datos en un caché de cliente** a través de un mecanismo llamado inserción del servidor.
- Fue estandarizado oficialmente en mayo de 2015. En este enlace pueden consultarse estadísticas sobre su uso.

La rápida adopción de **HTTP/2** se debió a que no requería cambios en los sitios web y aplicaciones. Para usarlo, sólo es necesario que un servidor actualizado se comunicase con un navegador reciente.

HTTP/3 - HTTP sobre QUIC

HTTP/3 tiene las mismas semánticas que las anteriores versiones de HTTP pero utiliza **QUIC** en lugar de **TCP** en la capa de transporte. En octubre de 2022, el 26% de los sitios web utilizaban HTTP/3.

QUIC está diseñado para proporcionar una **latencia mucho más baja para las conexiones HTTP**. Al igual que HTTP/2, es un protocolo multiplexado, pero HTTP/2 se ejecuta en una sola conexión TCP, por lo que la detección de pérdida de paquetes y la retransmisión manejada en la capa TCP pueden bloquear todos los flujos. **QUIC ejecuta varios flujos sobre UDP e implementa la detección de pérdida de paquetes y la retransmisión de forma independiente para cada flujo**, de modo que si ocurre un error, solo se bloquea el flujo con datos en ese paquete.

QUIC está definido en la RFC 9114 y HTTP/3 está soportado por la mayoría de los navegadores modernos.

- Fuente: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP