

# Practica 2

## 1. ¿Cuál es la función de la capa de aplicación?

Sus funciones son:

- Proveer servicios de comunicación a los usuarios y a las aplicaciones, incluye las aplicaciones mismas.
- Existe modelo de comunicación machine to machine (M2M), no hay usuarios (personas).
- Interfaz con el usuario -User Interface (UI)- u otras aplicaciones/servicios.
- Las aplicaciones que usan la red pertenecen a esta capa.
- Los protocolos que implementan las aplicaciones también.
- Existen aplicaciones que NO son de red que deben trabajar con aplicaciones/servicios para lograr acceso a la red.

## 2. Si dos procesos deben comunicarse:

a. ¿Cómo podrían hacerlo si están en diferentes máquinas?

b. Y si están en la misma máquina, ¿qué alternativas existen?

a) Si los procesos están en **diferentes máquinas**

Necesitan usar la **red** para comunicarse. Algunas formas:

- **Sockets de red:**
  - El más usado. Se identifican con **IP + Puerto**.
  - Pueden usar:
    - **TCP** → comunicación confiable (ej: HTTP, FTP).
    - **UDP** → más rápido pero sin control de errores (ej: DNS, streaming).
- **Protocolos de aplicación:**
  - HTTP, FTP, SMTP, etc. sobre los sockets.

👉 Ejemplo: tu navegador (proceso cliente) en tu PC se comunica con un servidor web (proceso servidor) en otra máquina usando **TCP + HTTP**.

---

b) Si los procesos están en la **misma máquina**

No hace falta la red, se puede usar **IPC local**. Existen varias alternativas:

- **Memoria compartida** 🧠: ambos procesos leen/escriben en la misma zona de memoria.
- **Pipes** (tuberías) ➡️: un proceso escribe y otro lee (ejemplo en Linux: `ls | grep txt`).
- **Named Pipes (FIFOs)**: igual que pipes, pero con nombre, permiten comunicación entre procesos no relacionados.
- **Sockets locales (UNIX sockets / loopback)**: funcionan igual que los sockets de red, pero sin salir de la máquina.
- **Señales (signals)**: un proceso notifica a otro con un evento (ej: matar un proceso).
- **Colas de mensajes**: el SO ofrece un buzón en el que los procesos depositan mensajes.
- **Semáforos**: más usados para sincronización que para enviar datos, pero también sirven en IPC.

👉 Ejemplo: en Linux, dos procesos pueden hablar usando **UNIX domain sockets** o una **cola de mensajes de System V**.

---

#### ✅ Resumen rápido:

- En máquinas diferentes → comunicación por **red** usando sockets (TCP/UDP).
- En la misma máquina → **mecanismos de IPC**: memoria compartida, pipes, sockets locales, colas de mensajes, señales.

3. Explique brevemente cómo es el modelo Cliente/Servidor. Dé un ejemplo de un sistema Cliente/Servidor en la "vida cotidiana" y un ejemplo de un sistema

informático que siga el modelo Cliente/Servidor. ¿Conoce algún otro modelo de comunicación?

### Modelo Cliente/Servidor

- Es un modelo de comunicación **asimétrico**:
  - El **cliente** inicia la comunicación, pide un recurso o servicio.
  - El **servidor** escucha de manera pasiva y responde con la información solicitada.
- La carga está **compartida**:
  - El cliente suele encargarse de la **interfaz**.
  - El servidor hace el **procesamiento principal** (datos, lógica, servicios).

👉 Ejemplo de interacción:

El cliente manda una **request** (pedido).

El servidor recibe, procesa y envía una **response** (respuesta).

---

### Ejemplo en la vida cotidiana

- **Restaurante**:
    - Cliente: pide un plato.
    - Camarero/cocina (servidor): prepara el pedido y lo entrega.
    - El cliente solo pide → el servidor provee.
- 

### Ejemplo informático

- **Navegador Web y Servidor Web**:
  - Cliente: Chrome, Firefox → pide una página web (HTTP request).
  - Servidor: Apache, Nginx → devuelve el contenido (HTTP response).

Otros ejemplos:

- Cliente de correo (Thunderbird, Gmail) ↔ servidor de correo (IMAP/SMTP).
  - Aplicación bancaria ↔ servidor central del banco.
- 

### Otros modelos de comunicación

Además de Cliente/Servidor, existen:

- **Mainframe (cliente tonto / dumb client):**
  - Todo el procesamiento en un servidor central, cliente casi no hace nada.
  - Ejemplo: terminales verdes antiguas conectadas a un mainframe IBM.
- **Peer-to-Peer (P2P):**
  - No hay roles fijos, cada nodo puede ser cliente y servidor a la vez.
  - Ejemplo: BitTorrent, eMule, Skype.
- **Modelo híbrido:**
  - Mezcla de C/S y P2P.
  - Ejemplo: Napster (servidor central indexaba, pero los archivos se compartían entre peers).

#### 4. Describa la funcionalidad de la entidad genérica "Agente de usuario" o "User agent".

Agente de Usuario (User Agent)

En redes y comunicaciones, un **User Agent (UA)** es la **entidad de la capa de aplicación** que actúa en nombre del usuario para interactuar con los servicios de red.

En palabras simples: es el **programa o aplicación que usa el usuario para acceder a la red.**

---

Funcionalidad

- **Interfaz con el usuario:**
  - Permite que el usuario acceda a servicios de red (ej: escribir un mail, navegar una web).
- **Generar y procesar mensajes:**
  - Envía requests y recibe respuestas.
- **Implementar protocolos de aplicación:**

- HTTP, SMTP, POP3, IMAP, FTP, etc.
  - **Presentación de datos:**
    - Traduce la información recibida en algo que el usuario pueda entender (texto, imágenes, audio, etc.).
- 

### Ejemplos de User Agents

- En **Web**: navegadores como Chrome, Firefox, Edge.
- En **Correo electrónico**: Outlook, Thunderbird, Gmail (web/app).
- En **Mensajería instantánea**: WhatsApp, Telegram, Slack.

👉 Nota: cuando un navegador se conecta a un servidor web, suele identificarse en el **header HTTP** `User-Agent:`, indicando nombre, versión, sistema operativo, etc.

---

### ✅ En resumen:

Un **User Agent** es la aplicación que sirve de puente entre el usuario y la red. Se encarga de generar las solicitudes, interpretar las respuestas y presentar la información de forma usable.

## 5. ¿Qué son y en qué se diferencian HTML y HTTP?

### HTML (HyperText Markup Language)

- Es un **lenguaje de marcado**.
- Sirve para **estructurar y presentar contenido** en la web (texto, imágenes, enlaces, formularios, etc.).
- Define **qué ve el usuario** en su navegador.
- No es un protocolo, es un **formato de documento**.

👉 Ejemplo: un archivo `.html` que tiene etiquetas como `<h1>`, `<p>`, `<img>`.

---

### HTTP (HyperText Transfer Protocol)

- Es un **protocolo de comunicación** de la capa de aplicación.
- Define **cómo se transmiten los mensajes** entre cliente y servidor en la web.
- Establece reglas:
  - Formato de las solicitudes (**request**) del cliente.

- Formato de las respuestas (**response**) del servidor.
- Métodos como `GET` , `POST` , `PUT` , etc.

👉 Ejemplo: cuando escribís `http://www.unlp.edu.ar` , tu navegador usa **HTTP** para pedir la página al servidor.

## 🔑 Diferencias

Característica	HTML	HTTP
Tipo	Lenguaje de marcado	Protocolo de comunicación
Función	Estructura y presenta información	Transfiere información entre cliente y servidor
Nivel	Contenido (documento)	Comunicación (capa de aplicación)
Ejemplo	<code>&lt;h1&gt;Hola Mundo&lt;/h1&gt;</code>	<code>GET /index.html HTTP/1.1</code>

### ✅ En resumen:

- **HTML** es el "idioma" en que están escritas las páginas web.
- **HTTP** es el "medio" o protocolo que permite que esas páginas viajen desde el servidor hasta tu navegador.

6. HTTP tiene definido un formato de mensaje para los requerimientos y las respuestas.

(Ayuda: apartado "Formato de mensaje HTTP", Kurose).

a. ¿Qué información de la capa de aplicación nos indica si un mensaje es de

requerimiento o de respuesta para HTTP? ¿Cómo está compuesta dicha información? ¿Para qué sirven las cabeceras?

b. ¿Cuál es su formato? (Ayuda:

<https://developer.mozilla.org/es/docs/Web/HTTP/Headers>)

c. Suponga que desea enviar un requerimiento con la versión de HTTP 1.1 desde

curl/7.74.0 a un sitio de ejemplo como [www.misitio.com](http://www.misitio.com) para obtener el recurso

/index.html. En base a lo indicado, ¿qué información debería enviarse mediante

encabezados? Indique cómo quedaría el requerimiento.

## Formato de mensaje en HTTP

HTTP define **dos tipos de mensajes**:

- **Request (petición)** → enviado por el cliente (navegador).
  - **Response (respuesta)** → enviado por el servidor.
- 

a. ¿Cómo diferenciar un *request* de una *response*?

### ◆ Request (mensaje del cliente):

- Comienza con una **línea de solicitud (request line)**:

```
<Método> <URI> <Versión HTTP>
```

Ejemplo:

```
GET /index.html HTTP/1.1
```

- **Método**: acción a realizar ( `GET` , `POST` , `HEAD` , etc.).
  - **URI**: el recurso que pide (ej: `/index.html` ).
  - **Versión HTTP**: ej. `HTTP/1.1` .
- 

### ◆ Response (mensaje del servidor):

- Comienza con una **línea de estado (status line)**:

```
<Versión HTTP> <Código de estado> <Frase de razón>
```

Ejemplo:

```
HTTP/1.1 200 O
```

- **Versión HTTP**: ej. `HTTP/1.1` .
  - **Código de estado**: número que indica el resultado ( `200` , `404` , `500` ...).
  - **Frase de razón**: explicación textual ( `OK` , `Not Found` , `Internal Server Error` ).
- 

¿Para qué sirven las **cabeceras** (headers)?

Son **pares clave:valor** que van después de la línea inicial.

- Aportan **información adicional** sobre el mensaje, el cliente o el servidor.
- Ejemplos de headers en request:
  - `Host: www.unlp.edu.ar` (servidor al que se quiere conectar).
  - `User-Agent: Chrome/115` (navegador que envía la petición).
  - `Accept: text/html` (tipos de contenido aceptados).
- Ejemplos en response:
  - `Content-Type: text/html` (tipo de contenido devuelto).
  - `Content-Length: 1280` (tamaño en bytes).
  - `Set-Cookie: sessionid=12345` (indica al cliente que guarde una cookie).

👉 Sirven para:

- Negociar formatos y lenguajes.
- Manejar sesiones (cookies).
- Optimizar transferencias (caching con `Last-Modified`, `ETag`).
- Controlar seguridad (`Authorization`, `WWW-Authenticate`).

b.

### Formato de las cabeceras HTTP (según MDN)

- **Estructura básica:** cada cabecera en HTTP consta de un **nombre (no sensible a mayúsculas)**, seguido de **dos puntos ( : )**, y luego su **valor**, todo en una sola línea continua. El espacio antes del valor se ignora.

developer.mozilla.org

- **Ejemplo** en una petición HTTP:

```
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X...)
Accept: text/html,application/xhtml+xml,...
Host: developer.mozilla.org
```

Acá, `User-Agent`, `Accept` y `Host` son los nombres, separados del valor por `:`  
developer.mozilla.org



- **Clasificación de cabeceras** (MDN distingue varios tipos):
  - **Generales:** aplican tanto a peticiones como respuestas (ej: `Via` ) `developer.mozilla.org`
  - **De petición:** dan más contexto sobre la solicitud o el cliente (ej: `Accept` , `User-Agent` ) `developer.mozilla.org+1`
  - **De respuesta:** describen información suplementaria del servidor o recurso (ej: `Server` , `Date` ) `developer.mozilla.org+1`
  - **De entidad:** relacionadas con el cuerpo del mensaje, como tipo y tamaño ( `Content-Type` , `Content-Length` ) `developer.mozilla.org+1`

c.

Línea de petición:

GET /index.html HTTP/1.1

Encabezados:

Host: www.misitio.com (servidor al que se quiere conectar)

User-Agent: curl/7.74.0 (navegador que envía la petición)

Accept: text/html (tipos de contenido aceptados)

7. Utilizando la VM, abra una terminal e investigue sobre el comando curl. Analice para qué sirven los siguientes parámetros (-I, -H, -X, -s).

Parámetros de `curl`

- **I (mayúscula i)**

👉 Hace un **HEAD request** en lugar de GET.

- Solo trae los **headers de la respuesta**, sin el cuerpo.
- Ejemplo:

```
curl -I https://www.unlp.edu.ar
```

Te devuelve solo el `HTTP/1.1 200 OK` , `Content-Type` , `Date` , etc.

- **H**

👉 Permite agregar un **header personalizado** a la petición.

- Ejemplo:

```
curl -H "User-Agent: mi-navegador" https://www.misitio.com
```

Envía el header `User-Agent: mi-navegador`.

- `X`

👉 Especifica el **método HTTP** que querés usar (por defecto es `GET`).

- Ejemplo:

```
curl -X POST https://www.misitio.com
```

Envía un `POST` en lugar de un `GET`.

- `s` (**silent mode**)

👉 Activa el **modo silencioso**:

- No muestra la barra de progreso ni mensajes de error.
- Útil en scripts o cuando no querés "ruido extra".
- Ejemplo:

```
curl -s https://www.misitio.com > salida.html
```

8. Ejecute el comando curl sin ningún parámetro adicional y acceda a [www.redes.unlp.edu.ar](http://www.redes.unlp.edu.ar). Luego responda:
- ¿Cuántos requerimientos realizó y qué recibió? Pruebe redirigiendo la salida (>) del comando curl a un archivo con extensión html y abrirlo con un navegador.
  - ¿Cómo funcionan los atributos href de los tags link e img en html?
  - Para visualizar la página completa con imágenes como en un navegador, ¿alcanza con realizar un único requerimiento?
  - ¿Cuántos requerimientos serían necesarios para obtener una página que tiene dos CSS, dos Javascript y tres imágenes? Diferencie cómo funcionaría un navegador respecto al comando curl ejecutado previamente.

Pregunta (a)

¿Cuántos requerimientos realizó y qué recibió?

- `curl` hizo **1 solo requerimiento HTTP**.
  - Recibió como respuesta el **documento HTML** inicial.
- ⚠ No descargó ni CSS, ni JS, ni imágenes.
- 

Pregunta (b)

¿Cómo funcionan los atributos `href` (en `<link>`) e `img` (en `<img>`) en HTML?

- En `<link>` → el atributo `href` apunta a otro recurso (ej: CSS).

```
<link rel="stylesheet" href="estilos.css">
```

Le dice al navegador: "descargá este archivo y úsalo como hoja de estilo".

- En `<img>` → el atributo `src` (no `href`) apunta a la URL de la imagen.

```

```

Le dice al navegador: "traé esta imagen y mostrála acá".

👉 En ambos casos el navegador debe hacer **nuevos requerimientos HTTP** para obtener esos recursos.

---

Pregunta (c)

¿Alcanza con un único requerimiento para visualizar la página completa con imágenes como en un navegador?

- ❌ No.
  - Con un solo `curl` obtenés solo el HTML.
  - El navegador, en cambio, **lee las referencias en el HTML** y hace automáticamente más requests para traer CSS, imágenes, JS, etc.
- 

Pregunta (d)

¿Cuántos requerimientos serían necesarios si la página tiene 2 CSS, 2 JS y 3 imágenes?

- HTML inicial: **1 requerimiento**.
- 2 CSS: **2 requerimientos**.
- 2 JS: **2 requerimientos**.
- 3 imágenes: **3 requerimientos**.

👉 Total = **1 + 2 + 2 + 3 = 8 requerimientos**.

- **Navegador**: hace todos los requests automáticamente.
- **curl simple**: solo trae el HTML (1 request). Si quisieras los demás recursos, deberías pedirlos manualmente con nuevos `curl`.

9. Ejecute a continuación los siguientes comandos:

`curl -v -s www.redes.unlp.edu.ar > /dev/null`

`curl -I -v -s www.redes.unlp.edu.ar`

a. ¿Qué diferencias nota entre cada uno?

b. ¿Qué ocurre si en el primer comando se quita la redirección a `/dev/null`?

¿Por

qué no es necesaria en el segundo comando?

c. ¿Cuántas cabeceras viajaron en el requerimiento? ¿Y en la respuesta?

Comando 1

```
curl -v -s www.redes.unlp.edu.ar > /dev/null
```

- `v` → modo verboso → muestra todo el detalle de la transacción (request y response headers).
- `s` → modo silencioso → suprime la barra de progreso.
- `> /dev/null` → redirige el **cuerpo** de la respuesta (el HTML) a la "nada" (se descarta).

👉 En este caso vas a ver:

- Los **headers del request** que envía `curl`.
- Los **headers del response** que devuelve el servidor.
- **No ves el HTML** porque lo tiraste a `/dev/null`.

## Comando 2

```
curl -I -v -s www.redes.unlp.edu.ar
```

- **I** → hace un **HEAD request** → el servidor solo devuelve **cabeceras de respuesta**, sin cuerpo.
- **v** → verboso → igual que antes, muestra request + response headers.
- **s** → silencioso → quita la barra de progreso.

👉 En este caso ves:

- Los **headers del request**.
- Los **headers de la respuesta**.
- **No hay cuerpo**, porque **HEAD** nunca devuelve HTML.

### a. Diferencias entre ambos

- **Primer comando:** hace un **GET**, baja el HTML pero lo redirige a **/dev/null**, así que solo ves los headers por **v**.
- **Segundo comando:** hace un **HEAD**, directamente el servidor no manda HTML, solo headers.

### b. ¿Qué pasa si quitás **> /dev/null** en el primero?

- Vas a ver en la terminal **los headers (por v) + el HTML completo** de la página.
- No es necesario en el segundo porque **HEAD** nunca trae cuerpo, solo headers.

### c. ¿Cuántas cabeceras viajan?

- **En el request** (desde cliente a servidor): típicamente 2–4 cabeceras, por ejemplo:

```
GET / HTTP/1.1
Host: www.redes.unlp.edu.ar
User-Agent: curl/7.74.0
Accept: */*
```

👉 O sea, unas **3–4 cabeceras**.

- **En la response** (desde servidor): depende del servidor, pero suelen ser varias, por ejemplo:

```
HTTP/1.1 200 OK
Date: Mon, 02 Sep 2024 18:00:00 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 3520
Connection: keep-alive
```

👉 Generalmente unas **5–7 cabeceras** o más.

El número exacto puede variar según la configuración del servidor.

#### 10. ¿Qué indica la cabecera Date?

La cabecera **Date** en HTTP indica la **fecha y hora en que el servidor generó la respuesta**.

- Sigue el formato **RFC 7231** (antes RFC 1123):

```
Date: Tue, 03 Sep 2024 18:42:00 GMT
```

- Siempre se expresa en **GMT (Greenwich Mean Time)**, no en hora local.
- Sirve para:
  - Sincronizar cachés (ej: junto con **Expires** o **Last-Modified** ).
  - Saber cuándo se originó la respuesta.
  - Depurar y verificar consistencia temporal entre cliente y servidor.

#### 11. En HTTP/1.0, ¿cómo sabe el cliente que ya recibió todo el objeto solicitado de manera completa? ¿Y en HTTP/1.1?

En HTTP/1.0

- Cada **objeto** se transfería en una **conexión TCP nueva**.
- Flujo típico:
  1. Cliente abre conexión.

2. Envía un `GET`.
3. Servidor responde con el objeto.
4. **Cuando el servidor cierra la conexión TCP**, el cliente sabe que recibió todo.

👉 Señal de fin = **cierre de conexión por parte del servidor**.

---

En HTTP/1.1

- Se introdujeron las **conexiones persistentes** (Keep-Alive por defecto).
- El servidor ya no cierra siempre la conexión después de cada objeto. Entonces se necesitó otra forma de marcar el final.

Dos mecanismos principales:

#### **Cabecera** `Content-Length`

- El servidor indica el tamaño en bytes del cuerpo de la respuesta.
- El cliente lee exactamente esa cantidad de bytes y sabe que terminó.

```
Content-Length: 3520
```

#### **Transfer-Encoding: chunked**

- Si el servidor no sabe el tamaño de antemano (ej: contenido dinámico), envía la respuesta en "chunks" (bloques).
- Cada bloque va precedido por su tamaño en hexadecimal.
- Un bloque de tamaño 0 marca el final.

```
Transfer-Encoding: chunked
1a\r\n
<datos...>\r\n
0\r\n
\r\n
```

👉 Señal de fin en HTTP/1.1 = **Content-Length** o **fin del último chunk** (no cierre de conexión).

---

#### **Resumen**

- **HTTP/1.0:** el cliente sabe que terminó cuando el servidor **cierra la conexión**.
- **HTTP/1.1:** el cliente sabe que terminó gracias a la cabecera **Content-Length** o al mecanismo de **chunked transfer encoding**, porque la conexión puede seguir abierta para más objetos.

## 12. Investigue los distintos tipos de códigos de retorno de un servidor web y su significado.

### 1xx – Informativos

👉 Indican que la solicitud fue recibida y se está procesando, pero aún no hay respuesta final.

- **100 Continue** → el cliente puede continuar enviando el cuerpo de la petición.
- **101 Switching Protocols** → el servidor acepta cambiar de protocolo (ej: a WebSockets).
- **102 Processing** → usado en WebDAV, indica que el servidor sigue procesando.

### 2xx – Éxito

👉 Indican que todo salió bien.

- **200 OK** → solicitud exitosa, se devuelve el recurso.
- **201 Created** → recurso creado con éxito (ej: tras un POST).
- **202 Accepted** → solicitud aceptada, pero aún no procesada.
- **204 No Content** → éxito, pero no hay cuerpo de respuesta (ej: al borrar algo).

### 3xx – Redirección

👉 El recurso no está donde se pidió, el cliente debe ir a otra URL.

- **301 Moved Permanently** → recurso movido de forma permanente.
- **302 Found** → redirección temporal.
- **303 See Other** → usar otra URL con **GET**.
- **304 Not Modified** → recurso no cambió, usar caché local.



- **307 Temporary Redirect** → como 302, pero sin cambiar el método HTTP.
- 

#### 4xx – Error del cliente

👉 El problema está en la petición del cliente.

- **400 Bad Request** → solicitud mal formada.
  - **401 Unauthorized** → falta autenticación o credenciales inválidas.
  - **403 Forbidden** → acceso prohibido, aunque esté autenticado.
  - **404 Not Found** → recurso no encontrado (el más famoso).
  - **405 Method Not Allowed** → método HTTP no permitido en ese recurso.
  - **408 Request Timeout** → el cliente tardó demasiado en enviar la petición.
  - **429 Too Many Requests** → el cliente superó el límite de peticiones (rate limiting).
- 

#### 5xx – Error del servidor

👉 El servidor falló al procesar la solicitud.

- **500 Internal Server Error** → error genérico del servidor.
  - **501 Not Implemented** → el servidor no soporta la funcionalidad pedida.
  - **502 Bad Gateway** → el servidor actuó como proxy y recibió una respuesta inválida.
  - **503 Service Unavailable** → servidor no disponible (sobrecarga, mantenimiento).
  - **504 Gateway Timeout** → un proxy/gateway no recibió respuesta a tiempo.
- 

#### ✅ Resumen rápido:

- **1xx** → Informativos (procesando).
- **2xx** → Éxito.
- **3xx** → Redirecciones.
- **4xx** → Errores del cliente.
- **5xx** → Errores del servidor.

13. Utilizando curl, realice un requerimiento con el método HEAD al sitio [www.redes.unlp.edu.ar](http://www.redes.unlp.edu.ar) e indique:
- a. ¿Qué información brinda la primera línea de la respuesta?
  - b. ¿Cuántos encabezados muestra la respuesta?
  - c. ¿Qué servidor web está sirviendo la página?
  - d. ¿El acceso a la página solicitada fue exitoso o no?
  - e. ¿Cuándo fue la última vez que se modificó la página?
  - f. Solicite la página nuevamente con curl usando GET, pero esta vez indique que quiere obtenerla sólo si la misma fue modificada en una fecha posterior a la que efectivamente fue modificada. ¿Cómo lo hace? ¿Qué resultado obtuvo? ¿Puede explicar para qué sirve?

a\_

```
redes@debian:~$ curl -I http://www.redes.unlp.edu.ar
HTTP/1.1 200 OK
```

La primera línea (llamada *línea de estado* o *status line*) contiene tres cosas:

Versión del protocolo HTTP. Ej.: `HTTP/1.1` o `HTTP/2`.

Código de estado numérico. Ej.: `200`, `301`, `404`, `500`. Indica la categoría y resultado.

Frase de razón (reason phrase) — texto explicativo (ej.: `OK`, `Not Found`)

b\_

```
redes@debian:~$ curl -I http://www.redes.unlp.edu.ar
HTTP/1.1 200 OK
Date: Wed, 10 Sep 2025 18:09:34 GMT
Server: Apache/2.4.56 (Unix)
Last-Modified: Sun, 19 Mar 2023 19:04:46 GMT
ETag: "1322-5f7457bd64f80"
Accept-Ranges: bytes
Content-Length: 4898
Content-Type: text/html
```

Los encabezados son 7: date, server, last-modified, etag, accept\_ranges, content-length, content-type.

c\_ El servidor es Apache/2.4.56 corriendo en Unix.

d\_ Si. Como la respuesta fue con código 200, fue exitosa.

e\_ La última vez que se modificó fue el domingo 19 de marzo de 2023.

f\_

El header para eso es `If-Modified-Since`. Ejemplo:

```
curl -v -H "If-Modified-Since: Wed, 01 Jan 2025 00:00:00 GMT" http://  
www.redes.unlp.edu.ar/index.html
```

- Estamos diciendo: "traeme la página **sólo si fue modificada después del 1 de enero de 2025**".
- Pero la página en realidad se modificó el **19 de marzo de 2023**, que es anterior a esa fecha → entonces el servidor responde:

```
HTTP/1.1 304 Not Modified
```

👉 Resultado: **no descarga el cuerpo de la página**, porque ya tenemos la versión más nueva en caché.

---

¿Para qué sirve esto?

Sirve para:

- **Optimizar tráfico:** el cliente no baja de nuevo el recurso si no cambió.
- **Mejorar tiempos de carga:** evita descargar archivos grandes sin necesidad.
- **Implementar caching HTTP:** los navegadores y proxies lo usan todo el tiempo.

14\_

```

redes@debian:~$ curl -v -u redes:RYC http://www.redes.unlp.edu.ar/restringido/the-end.php
* Trying 172.28.0.50:80...
* Connected to www.redes.unlp.edu.ar (172.28.0.50) port 80 (#0)
* Server auth using Basic with user 'redes'
> GET /restringido/the-end.php HTTP/1.1
> Host: www.redes.unlp.edu.ar
> Authorization: Basic cmVkZXM6U1lD
> User-Agent: curl/7.74.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Wed, 10 Sep 2025 18:38:13 GMT
< Server: Apache/2.4.56 (Unix)
< X-Powered-By: PHP/7.4.33
< Content-Length: 159
< Content-Type: text/html; charset=UTF-8
<
¡Felicitaciones, llegaste al final del ejercicio!

Fecha: 2025-09-10 18:38:13
* Connection #0 to host www.redes.unlp.edu.ar left intact
Verificación: 033a684d093947ce5adc4914295d8af7fb617d4405ff47fd9c9e8aa9f2f850e1redes@debian:~$ S

```

15\_

a\_

a.

Ejecutar en la VM:

```
curl www.redes.unlp.edu.ar/extras/prueba-http-1-0.txt
```


- `curl` va a hacer un **GET** a ese recurso usando **HTTP/1.0**.
- La salida que muestra es el **contenido exacto del archivo de prueba**:

```

redes@debian:~$ curl www.redes.unlp.edu.ar/extras/prueba-http-1-0.txt
GET /http/HTTP-1.1/ HTTP/1.0
User-Agent: curl/7.38.0
Host: www.redes.unlp.edu.ar
Accept: */*

```

(notar los **dos saltos de línea al final** → importantísimos).

 Esos saltos de línea los tenés que copiar completos porque forman parte del request crudo que después vas a pegar en Telnet.

b. Probar con `telnet`

```
telnet www.redes.unlp.edu.ar 80
```

Cuando se abra la conexión, pegar el contenido que copiaste del paso (a), es decir:

```
GET /extras/prueba-http-1-0.txt HTTP/1.0
```

¿Qué ocurre?

- El servidor procesa la request HTTP/1.0.
- Devuelve la respuesta (headers + cuerpo del archivo).
- **Luego cierra la conexión TCP automáticamente**, porque en HTTP/1.0 la forma de indicar el fin de la respuesta es **cerrando la conexión**. Por eso, si querés repetir el request, tenés que volver a abrir Telnet.


c. Repetir con `HTTP/1.1`

Ejecutando ahora:

```
curl www.redes.unlp.edu.ar/extras/prueba-http-1-1.txt
```

La salida del archivo será algo como:

```
redes@debian:~$ telnet www.redes.unlp.edu.ar 80
Trying 172.28.0.50...
Connected to www.redes.unlp.edu.ar.
Escape character is '^]'.
GET /http/HTTP-1.1/ HTTP/1.1
User-Agent: curl/7.38.0
Host: www.redes.unlp.edu.ar
Accept: */*
```

 Importante: en HTTP/1.1 el header `Host:` es **obligatorio**.

De nuevo, copiás toda la salida (con los saltos de línea incluidos), y la pegamos en Telnet:

```
telnet www.redes.unlp.edu.ar 80
```

¿Qué ocurre ahora?

- El servidor responde con el recurso pedido.
- **NO cierra la conexión TCP inmediatamente** → HTTP/1.1 soporta **conexiones persistentes** por defecto.
- Esto significa que podés **pegar otra vez el mismo request** en la misma sesión de Telnet y el servidor seguirá respondiendo.  
👉 A diferencia de HTTP/1.0, no necesitás reconectar para cada request.

#### 📌 Resumen conceptual

- **HTTP/1.0:** un request por conexión. Fin = servidor cierra la conexión.
- **HTTP/1.1:** conexiones persistentes. Fin = se usa `Content-Length` o `Transfer-Encoding: chunked` para saber dónde termina cada respuesta, sin cerrar la conexión.

16\_a. ¿Qué está haciendo al ejecutar el comando `telnet` ?

- Con `telnet www.redes.unlp.edu.ar 80` te conectás directamente por TCP al puerto 80 del servidor (puerto estándar de HTTP).
- Luego, al pegar el request ( `GET ...` ), estás escribiendo el mensaje HTTP de manera manual y viéndolo "crudo", sin intermediarios como el navegador o `curl` .

👉 En resumen: abrís un socket TCP y enviás tú mismo la petición HTTP.

b. ¿Qué método HTTP utilizó? ¿Qué recurso solicitó?

- Método: GET (el más común en HTTP, para pedir un recurso).
- Recurso: `/extras/prueba-http-1-0.txt` o `/extras/prueba-http-1-1.txt` , según el caso.

👉 O sea, pediste archivos de texto al servidor web.

c. ¿Qué diferencias notó entre los dos casos? ¿Por qué?

- HTTP/1.0:

- El servidor responde con el recurso.
- Luego cierra la conexión TCP.
- Para pedir otro recurso, hay que volver a abrir Telnet/conexión.
- HTTP/1.1:
  - El servidor responde con el recurso.
  - Mantiene la conexión abierta (persistente).
  - Podés enviar más de un request en la misma conexión.
  - Se usa `Content-Length` o `Transfer-Encoding` para saber dónde termina la respuesta.

👉 Diferencia clave: HTTP/1.0 = una conexión por recurso. HTTP/1.1 = una conexión para varios recursos.

---

d. ¿Cuál de los dos casos le parece más eficiente?

- Más eficiente: HTTP/1.1 ✅
  - Porque evita abrir/cerrar una conexión TCP por cada objeto (recordá el ejemplo de una página con HTML + 2 CSS + 2 JS + 3 imágenes → 8 recursos en total).
  - Con HTTP/1.0 serían 8 conexiones TCP.
  - Con HTTP/1.1 podría hacerse con 1 sola conexión persistente.

---

👉 ¿Puede traer algún problema?

Sí, algunos posibles:

- Si una conexión persistente se queda abierta demasiado tiempo → puede consumir recursos del servidor (memoria/sockets).
- En redes con proxies o firewalls antiguos, las conexiones persistentes a veces daban problemas.
- En algunos casos, si una conexión se interrumpe a mitad de la transferencia, hay que volver a pedir todo (aunque esto después se mejoró con HTTP/2 y multiplexación).

17\_

## 18\_HTTP y su problema de **stateless**

- El protocolo **HTTP es *sin estado* (stateless)**: cada request es independiente, el servidor no "recuerda" lo que pasó antes.
- Ejemplo: si entrás a un sitio web, hacés login y luego pedís otra página → el servidor, sin algo extra, **no sabría que sos la misma persona**.

---

### ¿Cómo se soluciona esto? → **Cookies**

Las cookies permiten **guardar estado entre distintas peticiones HTTP**.

Se implementan con dos cabeceras principales:

---

#### 1. **Set-Cookie**

- Es un **header de respuesta** que envía el servidor al cliente (normalmente un navegador).
- Sirve para **indicar al cliente que guarde una cookie** con ciertos datos.
- Ejemplo:

```
Set-Cookie: sessionId=abc123; Path=/; HttpOnly; Secure
```

El servidor le dice al cliente: "guardá esta cookie llamada `sessionId` con valor `abc123`".

---

#### 2. **Cookie**

- Es un **header de request**.
- En las siguientes peticiones al mismo dominio, el cliente devuelve automáticamente la cookie.
- Ejemplo:

```
Cookie: sessionId=abc123
```

El cliente le dice al servidor: "soy el mismo de antes, con esta sesión activa".

---

## Relación con HTTP

- **HTTP sin cookies = stateless** (no recuerda nada).



- **HTTP con cookies = stateful** (permite mantener sesiones, preferencias, carritos de compras, etc.).
- Gracias a `Set-Cookie` + `Cookie` se pueden implementar:
  - **Sesiones de usuario** (login).
  - **Persistencia de datos** (ej: carrito en un e-commerce).
  - **Preferencias de usuario** (idioma, configuración).
  - **Seguimiento** (tracking en publicidad, analytics).

## 19\_ Diferencia entre protocolos basados en texto y binarios

### Protocolos basados en **texto**

- Los mensajes se codifican como **texto legible por humanos** (generalmente ASCII o UTF-8).
- Ventajas:
  - Fácil de leer, depurar y probar con herramientas como `telnet`, `nc` o `curl`.
  - Se pueden entender directamente sin software especial.
- Desventajas:
  - Menos eficientes: más bytes transmitidos (cabeceras largas, repetidas).
  - El parsing (lectura y análisis) es más costoso para la máquina.

👉 Ejemplo: **HTTP/1.0, HTTP/1.1, SMTP, FTP, POP3.**

---

### Protocolos **binarios**

- Los mensajes se codifican en **formatos compactos de bytes** (no legibles directamente por humanos).
- Ventajas:
  - Más eficientes: menos tamaño, más velocidad.
  - Parsing más rápido y menos ambiguo para las máquinas.
- Desventajas:
  - No se pueden leer "a ojo" fácilmente.
  - Requieren herramientas especiales para depuración.

👉 Ejemplo: **HTTP/2, DNS, SSH, TLS, SMB.**

---

### Clasificación de HTTP según la versión

- **HTTP/1.0** → protocolo **basado en texto**.
- **HTTP/1.1** → protocolo **basado en texto** (igual que 1.0, pero con mejoras: cabeceras obligatorias como **Host**, conexiones persistentes, etc.).
- **HTTP/2** → protocolo **binario** (usa *frames* binarios, multiplexación, compresión de cabeceras con HPACK).

20\_a\_

### HTTP/1.0

- En **HTTP/1.0**, cada servidor web estaba pensado para atender **un solo sitio por dirección IP**.
- El request incluía solo la línea de petición:

```
GET /index.html HTTP/1.0
```

- ❌ No existía la cabecera **Host**.
- Problema: si un mismo servidor quería alojar **varios sitios en la misma IP** (ej. virtual hosting compartido), el servidor no tenía forma de saber **a qué dominio** correspondía el request.

---

### HTTP/1.1

- Se introduce la cabecera **Host** como **obligatoria**.
- Ejemplo:

```
GET /index.html HTTP/1.1
Host: www.unlp.edu.ar
```

- Gracias a **Host**, el servidor puede diferenciar qué sitio atender aunque usen la misma IP.
- 👉 Esto hizo posible el **hosting compartido** (muchos dominios en un mismo servidor/IP).
-

## HTTP/2

- HTTP/2 es un protocolo **binario** y no repite cabeceras texto plano como en 1.1.
- Pero sigue necesitando la **información del host** para identificar el sitio.
- En HTTP/2, la información de `Host` se transmite mediante un **pseudo-header** obligatorio:

```
:authority
```

que cumple la misma función que `Host` en HTTP/1.1.

👉 Entonces:

- HTTP/1.1 → `Host` obligatorio.
- HTTP/2 → `:authority` reemplaza a `Host`, pero los navegadores/envíos todavía incluyen `Host` por compatibilidad.

---

## Resumen

- **HTTP/1.0:** no existía `Host`. Un IP = un sitio.
- **HTTP/1.1:** `Host` obligatorio. Permite *virtual hosting* (varios dominios en una IP).
- **HTTP/2:** reemplaza `Host` por el pseudo-header `:authority`, aunque `Host` suele enviarse igual por compatibilidad.

b\_ El request que planteás

```
GET /index.php HTTP/1.1
User-Agent: curl/7.54.0
```

---

¿Es correcto en **HTTP/1.1**?

❌ **No es correcto.**

- En **HTTP/1.1**, la cabecera `Host` es **obligatoria**.
- Tu request incluye `User-Agent`, pero **falta el** `Host`.
- Un servidor HTTP/1.1 que cumpla el estándar debería responder con:

## HTTP/1.1 400 Bad Request

porque no puede saber a qué dominio corresponde el recurso `/index.php`.

Ejemplo correcto en HTTP/1.1

```
GET /index.php HTTP/1.1
Host: www.misitio.com
User-Agent: curl/7.54.0
```

👉 Ahora sí es válido, porque el servidor sabe qué sitio estás pidiendo.

C\_

Recordemos el request en HTTP/1.1

```
GET /index.php HTTP/1.1
Host: www.info.unlp.edu.ar
```

¿Qué cambia en HTTP/2?

1. HTTP/2 es **binario**, no de texto → los requests no viajan como líneas legibles, sino como *frames*.
2. Las cabeceras se transmiten usando **pseudo-headers** especiales:
  - `:method` → reemplaza el verbo HTTP.
  - `:path` → reemplaza la URI relativa.
  - `:scheme` → indica el esquema ( `http` o `https` ).
  - `:authority` → reemplaza a `Host`.
3. El request en HTTP/2 se representaría conceptualmente así:

```
:method: GET
:path: /index.php
:scheme: https
:authority: www.info.unlp.edu.ar
```

Diferencia clave

- En **HTTP/1.1** usás `Host` .
- En **HTTP/2** usás `:authority` .
- Además, como estás usando **HTTPS**, la capa de transporte también cifra todo con TLS