

CONCEPTO 4 DE 4

El viaje de una request HTTP

Qué pasa exactamente desde que apretás un botón hasta que ves la respuesta en pantalla

HTTP: EL IDIOMA COMÚN DE LA WEB

HTTP (HyperText Transfer Protocol) es el protocolo estándar que usan todos los navegadores y aplicaciones web del mundo para comunicarse con servidores. Nació en 1991 junto con la World Wide Web — Tim Berners-Lee los diseñó juntos. Hoy sigue siendo el protocolo sobre el que corre prácticamente todo el tráfico de internet.

Todo mensaje HTTP tiene estas tres partes:

MÉTODO ¿Qué querés hacer? Leer datos, crear algo nuevo, modificar, o borrar. Es un verbo que le dice al servidor cuál es la intención de la request.

URL / ENDPOINT ¿Sobre qué? La dirección exacta del recurso.
`api.banco.com/cuenta/1234/saldo` — cada segmento de la URL tiene un significado.

HEADERS **Metadatos.** Información extra que va con el mensaje: quién sos (token de autenticación), qué formato de respuesta esperás (`Accept: application/json`), de qué tipo son los datos que mandás.

BODY **Los datos (opcional).** En requests tipo POST/PUT que crean o modifican algo, el body lleva el contenido. Por ejemplo, al registrarte, el body lleva tu email y contraseña en formato JSON.

ANTES DE LA REQUEST: EL DNS

Escribís `mercadolibre.com` en el navegador. El navegador no sabe a qué servidor ir — las computadoras se comunican por **direcciones IP** numéricas, no por nombres. El primer paso es una consulta al **DNS** (Domain Name System), el "directorio telefónico" de internet:

Vos escribís: `mercadolibre.com`

|



Consulta DNS → ¿Qué IP tiene `mercadolibre.com`?

|



Respuesta DNS → `18.205.X.X` (IP del servidor)

|



Ahora el navegador sabe a dónde enviar la request HTTP

En AWS, el servicio **Route 53** es justamente eso — un servicio de DNS. Cuando desplegamos una app, una de las tareas es configurar Route 53 para que el dominio apunte al servidor correcto.

LOS MÉTODOS HTTP

HTTP define varios verbos. Los cinco más usados en APIs REST modernas:

GET

Obtener información. **No modifica nada**. Se puede repetir sin consecuencias.

GET `/productos/42`

POST

Crear un recurso nuevo. Lleva datos en el **body**. Cada llamada puede crear uno nuevo.

POST `/ordenes`

PUT

Reemplazar un recurso completo. Mandás el objeto entero con los nuevos valores.

PUT `/usuario/77`

PATCH

Modificar **solo algunos campos** de un recurso existente. Más eficiente que PUT.

PATCH `/usuario/77/email`

DELETE

Eliminar un recurso. Si se repite sobre el mismo ID, el resultado es el mismo: ya no existe.

DELETE `/sesion/actual`

Idempotencia

GET, PUT, DELETE son *idempotentes*: repetirlos con los mismos parámetros da el mismo resultado. POST no lo es — cada llamada puede crear un nuevo recurso.

QUERY PARAMS VS BODY: ¿CÓMO VAN LOS DATOS?

Hay dos formas de mandar datos en una request:

Query params – en la URL

`/productos?categoria=ropa&precio_max=5000&orden=desc`

Se usan para **filtrar o paginar**. Son visibles en la URL, no van encriptados por sí mismos (aunque HTTPS encripta la URL completa). Se usan principalmente con GET.

Body – en el cuerpo del mensaje

```
{ "email": "...", "password": "..." }
```

Se usan para **enviar datos sensibles o grandes** (formularios, archivos). Van en el body del mensaje, no en la URL. Típicamente en POST, PUT, PATCH.

EL VIAJE COMPLETO — EL CAJERO AUTOMÁTICO

Estás frente al cajero y apretás **"Consultar saldo"**. Todo lo que viste arriba sucede en menos de un segundo:

1

DNS lookup

El cajero resuelve el nombre `api.mibanco.com` a una dirección IP. Si ya lo hizo antes y la respuesta está en cache, se salta este paso.

2

Conexión y handshake TLS

El cajero establece una conexión con el servidor. Si es HTTPS (y siempre debería serlo), antes de enviar nada hay un *TLS handshake*: el servidor le muestra su certificado, el cajero lo valida, y ambos acuerdan una clave de encriptación en tiempo real. Nadie más puede descifrar lo que sigue.

3

El cajero arma y envía la request

`GET api.mibanco.com/cuenta/saldo`

En los headers va la identidad: `Authorization: Bearer eyJhbG...` — un token JWT que el servidor puede verificar sin consultar la base de datos. El mensaje viaja encriptado.

4

El backend verifica y procesa

El servidor recibe el mensaje. Primero valida el token: ¿es auténtico? ¿está vencido? Si algo falla, responde de inmediato con `401` sin tocar la base de datos. Si pasa, consulta: `SELECT saldo FROM cuentas WHERE id = ?`

5

La respuesta vuelve al cajero

El backend empaqueta el resultado con un código de estado y lo manda de vuelta. El cajero recibe la respuesta, lee el código, y decide qué mostrar en pantalla.

CÓDIGOS DE ESTADO HTTP: LAS FAMILIAS

Los códigos tienen tres dígitos. El primero define la familia:

2xx

Éxito

`200 OK` — lo que pediste, acá está
`201 Created` — se creó el recurso
`204 No Content` — ok, pero no hay nada que devolver (ej: un DELETE exitoso)

3xx

Redirección

`301 Moved Permanently` — la URL cambió para siempre
`302 Found` — redirección temporal
`304 Not Modified` — tenés la versión en cache, no cambió nada

4xx

Error del cliente

`400 Bad Request` — la request está mal formada
`401 Unauthorized` — no te autenticaste
`403 Forbidden` — estás autenticado, pero no tenés permiso
`404 Not Found` — ese recurso no existe
`429 Too Many Requests` — rate limit superado

5xx

Error del servidor

`500 Internal Server Error` — algo falló en el servidor
`502 Bad Gateway` — el servidor recibió respuesta inválida de otro servicio
`503 Service Unavailable` — el servidor está caído o sobrecargado

401 vs 403: La diferencia importa. **401** significa "no sé quién sos — inició sesión". **403** significa "sé quién sos, pero no podés hacer esto". El primero es falta de autenticación; el segundo, falta de autorización.

SESIONES Y COOKIES: CÓMO EL SERVIDOR TE "RECUERDA"

HTTP es por naturaleza **stateless** (sin estado): cada request es independiente, el servidor no recuerda la anterior. ¿Cómo funcionan entonces el "estás logueado"?

Hay dos mecanismos principales:

Cookies de sesión

El servidor crea un ID de sesión, lo guarda en su base de datos, y se lo manda al navegador como cookie. En cada request siguiente, el navegador manda la cookie automáticamente. El servidor busca el ID y sabe quién sos.

Tokens JWT

Un string encriptado que el servidor genera al loguearte. El cliente lo guarda y lo manda en el header de cada request. El servidor puede verificar el token *sin consultar la base de datos*. Más escalable, ideal para APIs.

EL DIAGRAMA COMPLETO

```
[ Cajero – Frontend ]
|
| 1. DNS lookup: api.mibanco.com → 18.X.X.X
| 2. TLS handshake (acuerdo de encriptación)
|
| GET /cuenta/saldo
| Headers: { Authorization: Bearer eyJhbG... }
| (viaja encriptado – HTTPS 🔒)
|
[ Servidor del Banco – Backend ]
|  — ¿Token válido y no vencido? → no → 401 Unauthorized
|  — ¿Tiene permiso para /saldo? → no → 403 Forbidden
|  — ¿La cuenta existe?          → no → 404 Not Found
|  — Todo ok → consulta la base de datos
|
[ Base de Datos ]
|  — SELECT saldo FROM cuentas WHERE id = ?
|  — → 47350
|
[ Servidor del Banco – Backend ]
|
| HTTP Response 200 OK
| { "saldo": 47350, "moneda": "UYU" }
|
[ Cajero – Frontend ] → muestra $47.350 en pantalla
```

PARA RECORDAR

Todo eso — DNS, TLS handshake, request, validación de token, consulta a base de datos, respuesta — sucede en **menos de un segundo**. Ahora imaginá miles de personas haciendo lo mismo simultáneamente. Los servidores tienen que escalar para absorber ese volumen sin degradarse. Esa capacidad de escalar bajo demanda es una de las razones por las que las apps modernas se construyen en cloud.