PuigFlows

HTTP Caching en Producción: Cache-Control, ETags, stale-while-revalidate y CDNs

Guía · Dificultad: Intermedio · Proyectos Reales · Lectura: ~36 min
Edición bilingüe — Español & English · puigflows.com · Agosto 2026

PREMIUM

Versión en Español

Guía completa en español. The English version starts after this section.

La petición más rápida es la que nunca llega a tu servidor

Antes de añadir Redis, réplicas de lectura o más pods, hay una capa de caché que ya tienes desplegada y que probablemente estás desaprovechando: la del propio protocolo HTTP. El navegador de cada usuario, los proxies intermedios y tu CDN saben guardar y reutilizar respuestas; solo necesitan que tu servidor les diga cómo, con un puñado de cabeceras. Bien configurada, esta capa elimina peticiones enteras (ni siquiera hay ida y vuelta por la red) o las reduce a un 304 sin cuerpo. Mal configurada, sirve los datos de un usuario a otro o deja a tus clientes viendo una versión vieja de la aplicación durante horas después de un deploy.

El comportamiento está estandarizado en la RFC 9111 y es idéntico en HTTP/1.1, HTTP/2 y HTTP/3, así que todo lo que sigue aplica a cualquier stack.

Dos decisiones independientes: frescura y validación

Todo el caching HTTP se reduce a dos preguntas, y cada una se responde con cabeceras distintas:

- **¿Durante cuánto tiempo puedo reutilizar esta respuesta sin preguntar?** Eso es la *frescura*, y la controla Cache-Control.
- **Cuando caduque, ¿cómo compruebo si sigue siendo válida sin volver a descargarla?** Eso es la *validación*, y la resuelven ETag y Last-Modified junto con las peticiones condicionales.

Separarlas evita la confusión más común del protocolo: no-cache no significa «no guardes esto en caché». Significa «guárdalo, pero revalida con el origen antes de cada uso». La directiva que de verdad prohíbe almacenar es no-store.

Cache-Control: las directivas que de verdad usarás

- max-age=N: la respuesta es fresca durante N segundos. Mientras dure, el navegador la sirve desde disco o memoria sin tocar la red.
- s-maxage=N: igual que max-age, pero solo para cachés compartidas (CDN, proxies). Te permite dar TTLs largos al CDN, que puedes purgar, y TTLs cortos al navegador, que no puedes.
- public / private: private restringe el almacenamiento al navegador del usuario; imprescindible en respuestas por usuario. public permite guardarla en cachés compartidas incluso cuando normalmente no se haría.
- no-cache: almacena, pero revalida siempre antes de usar. Ideal para HTML.
- no-store: no guardes nada, en ningún sitio. Para datos sensibles.
- immutable: promete que el contenido de esa URL no cambiará jamás, así el navegador ni siquiera revalida al recargar. Solo tiene sentido con nombres de fichero versionados por hash.
- must-revalidate: cuando la respuesta caduque, prohíbe servirla en modo «stale» aunque el origen esté caído; mejor un error que un dato viejo. Útil en datos financieros o de inventario.

Cómo calcula la caché la edad de una respuesta: Age, Date y la aritmética de la frescura

Para razonar sobre incidentes de caché necesitas saber cómo decide una caché si algo sigue fresco. La regla de la RFC 9111 es simple: una respuesta es fresca mientras su *edad* sea menor que su *vida útil* (el max-age o s-maxage aplicable). La edad no empieza a contar cuando tu navegador recibe la respuesta, sino cuando la generó el origen.

Ahí entra la cabecera Age: cada caché compartida que reenvía una respuesta añade (o incrementa) Age con los segundos que esa respuesta lleva almacenada aguas arriba. Si tu CDN te entrega Age: 45 con Cache-Control: max-age=60, tu navegador solo la considerará fresca 15 segundos más. Esto explica un fenómeno que despista a mucha gente: pones max-age=60, y aun así ves peticiones al origen a los pocos segundos desde algunos clientes — sus copias «nacieron viejas» porque venían de un nodo que ya llevaba tiempo guardándolas.

Puedes inspeccionarlo con curl en dos líneas:

```
curl -sI https://tu-sitio.com/api/catalog | grep -i -E 'age|cache-control|etag|x-cache|cf-cache-status'

# Age: 41                    ← segundos almacenada en la CDN
# Cache-Control: public, s-maxage=60, stale-while-revalidate=300
# CF-Cache-Status: HIT       ← Cloudflare sirvió desde caché
```

Dos consecuencias prácticas. Primera: si encadenas varias capas (CDN regional, caché intermedia, navegador), los TTLs no se suman; la edad viaja con la respuesta y todas las capas la descuentan. Segunda: si tu origen emite una cabecera Date desviada (reloj mal sincronizado), las cachés calcularán edades negativas o infladas y tu política se vuelve impredecible. Mantén NTP en orden: es infraestructura de caching aunque no lo parezca.

stale-while-revalidate: velocidad de caché con datos casi frescos

Definida en la RFC 5861, stale-while-revalidate=N es la directiva más infrautilizada del protocolo. Cuando la respuesta caduca, la caché puede seguir sirviendo la copia vieja durante N segundos más mientras la refresca en segundo plano. El usuario ve siempre respuestas instantáneas y el dato nunca envejece más que unos segundos. Su hermana stale-if-error=N hace lo mismo cuando el origen responde 5xx o no responde: tu CDN se convierte en una red de seguridad durante un incidente.

```
Cache-Control: public, max-age=0, s-maxage=60, stale-while-revalidate=300, stale-if-error=86400
```

Esa línea le dice al CDN: considera esto fresco 60 segundos; hasta 5 minutos después, sírvelo viejo mientras lo renuevas por detrás; y si mi origen se cae, sigue sirviéndolo hasta un día entero.

Un matiz importante que casi nadie documenta: el soporte difiere por capa. En el navegador, Chrome (75+) y Firefox (68+) aplican stale-while-revalidate a subrecursos, pero Safari no lo implementa en su caché HTTP, y no se aplica a navegaciones de documento. En CDNs, Cloudflare, Fastly y CloudFront lo soportan, aunque CloudFront exige habilitarlo en la cache policy. La actitud correcta es tratarlo como una optimización progresiva: quien lo entiende vuela, quien no, simplemente usa max-age.

ETags y peticiones condicionales: el 304 que ahorra ancho de banda

Cuando una respuesta caduca, la caché no tiene por qué descargarla de nuevo. Si el servidor incluyó un ETag (un identificador opaco del contenido), el cliente puede enviar If-None-Match con ese valor. Si el contenido no cambió, el servidor responde 304 Not Modified sin cuerpo, la caché renueva la frescura de su copia y el usuario se ahorra la transferencia completa.

Dos detalles importan en producción. Primero: genera el ETag a partir del contenido (un hash del cuerpo), no de

metadatos del sistema de ficheros. El ETag clásico de Apache incluía el inode y el de nginx usa mtime y tamaño: con varios servidores detrás de un balanceador, cada uno emite un ETag distinto para el mismo fichero y tu tasa de 304 se hunde sin que ningún dashboard te lo cuente. Segundo: un 304 debe reenviar las cabeceras que actualizan la respuesta almacenada (ETag, Cache-Control), no solo el código de estado.

Así se ve un endpoint con ETag y revalidación condicional en FastAPI:

```
import hashlib, json
from fastapi import FastAPI, Request, Response

app = FastAPI()
CACHE_HEADERS = {"Cache-Control": "private, max-age=30, stale-while-revalidate=120"}

@app.get("/api/products")
async def list_products(request: Request):
    products = await fetch_products()  # tu consulta real
    body = json.dumps(products, sort_keys=True)

    # ETag fuerte derivado del contenido: estable entre servidores
    etag = "" + hashlib.sha256(body.encode()).hexdigest()[:32] + ""

    if request.headers.get("if-none-match") == etag:
        # 304: sin cuerpo, pero reenviando las cabeceras que refrescan la copia
        return Response(status_code=304, headers={"ETag": etag, **CACHE_HEADERS})

    return Response(
        content=body,
        media_type="application/json",
        headers={"ETag": etag, **CACHE_HEADERS},
    )
```

El hash se calcula sobre el JSON ya serializado con claves ordenadas, de modo que dos servidores cualesquiera produzcan el mismo ETag para los mismos datos. El coste de calcular un SHA-256 es despreciable comparado con transferir el cuerpo entero.

ETags fuertes, ETags débiles y Last-Modified: cuándo usar cada validador

HTTP define dos tipos de validador y conviene saber elegir. Un ETag *fuerte* (`"abc123"`) afirma que el cuerpo es byte a byte idéntico; es el único válido para reanudar descargas con `Range`. Un ETag *débil* (`W/"abc123"`) solo promete equivalencia semántica: útil cuando el cuerpo cambia de forma irrelevante entre servidores, por ejemplo un JSON con timestamps de renderizado o compresión que produce bytes distintos para el mismo contenido.

`Last-Modified` con `If-Modified-Since` es el validador veterano: más débil (resolución de un segundo, y cualquier reescritura del fichero lo «invalida» aunque el contenido sea igual), pero gratis si sirves ficheros estáticos. La práctica recomendada es enviar ambos cuando puedas; si el cliente manda los dos, `If-None-Match` tiene prioridad.

Reglas rápidas que evitan sustos:

- Contenido generado (JSON, HTML renderizado): ETag fuerte por hash del cuerpo. Determinista entre servidores.
- Ficheros estáticos servidos por nginx/Apache en un solo servidor: `Last-Modified` + ETag por defecto están bien; con varios servidores, sincroniza mtimes en el deploy (rsync los preserva, un `COPY` de Docker puede no hacerlo) o desactiva el ETag de metadatos.

- Si activas compresión dinámica, recuerda que nginx degrada el ETag a débil al comprimir al vuelo: coherente con la semántica, pero puede romper descargas con rangos.

Concurrencia optimista: If-Match y el 412 que salva tus escrituras

Los validadores no solo sirven para leer más barato: también evitan el *lost update* en escrituras. El patrón es concurrencia optimista sobre HTTP: el cliente lee un recurso y recibe su ETag; cuando envía el `PUT` / `PATCH`, incluye `If-Match` con ese ETag; si otro cliente modificó el recurso entre medias, el ETag ya no coincide y el servidor responde `412 Precondition Failed` en lugar de pisar los cambios.

```
from fastapi import FastAPI, Request, Response, HTTPException

@app.put("/api/documents/{doc_id}")
async def update_document(doc_id: str, request: Request):
    doc = await get_document(doc_id)
    if doc is None:
        raise HTTPException(404)

    current_etag = compute_etag(doc)        # mismo hash que usas en el GET
    client_etag = request.headers.get("if-match")

    if client_etag is None:
        # Obligar a los clientes a mandar If-Match evita pisadas accidentales
        raise HTTPException(428, detail="If-Match requerido")
    if client_etag != current_etag:
        raise HTTPException(412, detail="El documento cambió; recarga y reintenta")

    payload = await request.json()
    updated = await save_document(doc_id, payload)
    return Response(headers={"ETag": compute_etag(updated)})
```

El código 428 (`Precondition Required`) existe precisamente para declarar que tu API exige peticiones condicionales. Este patrón convierte una condición de carrera silenciosa —dos editores guardando a la vez, el segundo borra lo del primero— en un error explícito que la interfaz puede resolver mostrando un diff o pidiendo recargar. Si ya generas ETags para el caching, la concurrencia optimista te sale casi gratis.

El patrón de oro para una SPA: assets inmutables, HTML que siempre revalida

Los bundlers modernos (Vite, webpack, esbuild) escriben el hash del contenido en el nombre de cada fichero: `app.3f9c1b.js`. Si el contenido cambia, cambia la URL. Eso habilita la división de trabajo más eficaz de todo el caching web:

- **Assets con hash**: `Cache-Control: public, max-age=31536000, immutable`. Un año de caché y ni una sola revalidación. La URL es el versionado.
- **El HTML**: `Cache-Control: no-cache`. Se revalida en cada navegación, así cada deploy llega al instante: el HTML nuevo referencia los bundles nuevos.

```
# nginx
location /assets/ {
    add_header Cache-Control "public, max-age=31536000, immutable";
}
```

```
location = /index.html {
    add_header Cache-Control "no-cache";
}
```

El error inverso es la avería clásica de las SPA: HTML con max-age largo. Tras un deploy, los usuarios conservan un index.html viejo que pide bundles que ya no existen en el servidor, y el resultado es la pantalla en blanco hasta que a alguien se le ocurre «vaciar la caché».

APIs: qué cachear, qué no, y el papel de Vary

- **Datos sensibles o efímeros** (sesiones, tokens, datos médicos o de pago): Cache-Control: no-store.
- **Respuestas por usuario** (su carrito, su perfil): private, max-age=30 o similar. Nunca public: una caché compartida serviría el carrito de un usuario a otro.
- **Datos compartidos y semi-estáticos** (catálogo, configuración, contenido editorial): public, s-maxage=60, stale-while-revalidate=300 y deja que el CDN absorba el tráfico.

La cabecera Vary le dice a la caché qué cabeceras de la petición forman parte de la clave. Si sirves respuestas comprimidas, Vary: Accept-Encoding; si la respuesta cambia por idioma, añade Accept-Language. Y cuidado con el extremo opuesto: Vary: Cookie o Vary: * fragmentan la caché en tantas variantes que el hit rate cae a cero.

El mismo patrón completo en Express, cubriendo assets, HTML y una API cacheable:

```
const express = require("express");
const crypto = require("node:crypto");
const app = express();

// Assets con hash en el nombre: cachear un año, sin revalidar
app.use("/assets", express.static("dist/assets", {
  immutable: true,
  maxAge: "1y",
}));

// HTML: revalidar siempre, para que cada deploy llegue al instante
app.get("/", (req, res) => {
  res.set("Cache-Control", "no-cache");
  res.sendFile("index.html", { root: "dist" });
});

// API semi-estática: 60 s de frescura en CDN + margen de revalidación
app.get("/api/catalog", async (req, res) => {
  const body = JSON.stringify(await getCatalog());
  const etag =
    '"' + crypto.createHash("sha256").update(body).digest("hex").slice(0, 32) + '"';

  res.set({
    ETag: etag,
    "Cache-Control": "public, max-age=0, s-maxage=60, stale-while-revalidate=300",
    Vary: "Accept-Encoding",
  });

  if (req.get("If-None-Match") === etag) return res.status(304).end();
  res.type("application/json").send(body);
});
```

La clave de caché: query strings, normalización y por qué el orden de los parámetros importa

Toda caché indexa por una *clave*: normalmente método + host + ruta + query string, más lo que añada `Vary`. Entender la clave explica la mitad de los hit rates decepcionantes.

El caso típico: `/api/catalog?page=2&size=20` y `/api/catalog?size=20&page=2` son la misma consulta para tu backend, pero dos entradas distintas para la mayoría de cachés. Añade un parámetro de tracking (`utm_source`, `fbclid`, `gclid`) y cada campaña de marketing fragmenta tu caché en miles de variantes que jamás se reutilizan. Las defensas, por orden de eficacia:

- **Normaliza en el borde.** Los CDNs permiten definir qué parámetros forman parte de la clave (CloudFront cache policies, Cloudflare cache rules, Fastly VCL). Declara la lista blanca de parámetros que de verdad cambian la respuesta y ordénalos alfabéticamente; el resto (tracking) se ignora para la clave y se reenvía al origen si hace falta.
- **Genera URLs canónicas.** Si tu frontend siempre construye la query en el mismo orden, la fragmentación desaparece en origen. Una función utilitaria que ordene `URLSearchParams` cuesta cinco líneas.
- **No metas datos por usuario en la URL de recursos compartidos.** Un `?token=...` en una URL de imagen convierte cada usuario en una variante de caché (y filtra el token a logs y Referers de paso).

La regla mental: la clave de caché define qué se comparte. Todo lo que esté en la clave fragmenta; todo lo que afecte a la respuesta y NO esté en la clave (ni en `Vary`) es un bug de datos cruzados esperando su momento.

Seguridad: cache poisoning y cache deception, los dos ataques que debes conocer

Una caché compartida es una superficie de ataque: si un atacante consigue que la caché guarde una respuesta manipulada bajo una clave legítima, cada visitante posterior la recibirá. Son los dos ataques documentados extensamente por la investigación de PortSwigger, y se previenen con disciplina de configuración, no con un WAF.

Web cache poisoning explota *entradas sin clave* (unkeyed inputs): cabeceras como `X-Forwarded-Host` que el backend usa para generar la respuesta (por ejemplo, URLs absolutas en el HTML) pero que la caché no incluye en la clave. El atacante envía una petición con la cabecera envenenada; el backend genera HTML que apunta a un dominio malicioso; la caché lo guarda bajo la URL normal; y todos los usuarios siguientes reciben ese HTML. Defensas: no uses cabeceras de la petición para construir respuestas cacheables (o inclúyelas en `Vary`), desactiva el soporte de cabeceras de sobrescritura que no necesites, y cachea solo lo que declares explícitamente cacheable.

Web cache deception es el inverso: engañar a la caché para que guarde contenido privado. El clásico: el atacante hace que la víctima visite `/mi-cuenta/perfil.css`; el backend ignora el sufijo y responde con el HTML privado del perfil; el CDN ve la extensión `.css`, aplica su regla de «los estáticos se cachean» y guarda la página privada de la víctima, que el atacante recupera después visitando la misma URL. Defensas: marca todo lo dinámico con `Cache-Control: private, no-store` explícito, configura el CDN para que respete las cabeceras del origen por encima de reglas por extensión, y rechaza (404/redirección) rutas con sufijos que tu aplicación no reconoce en lugar de responder «con buena voluntad».

La moraleja común: sé explícito. Una respuesta sin `Cache-Control` es una invitación a que cada capa intermedia decida por ti, y las decisiones por defecto de una caché nunca incorporan tu modelo de amenazas.

Authorization, contenido privado y cachés compartidas

La RFC 9111 trae una regla que sorprende: por defecto, una caché compartida **no debe almacenar** respuestas a peticiones que llevaban cabecera `Authorization` … salvo que la respuesta incluya directivas que lo re-permitan explícitamente: `public`, `s-maxage` o `must-revalidate`. Es decir, ese `public` que añadiste «por si acaso» a un endpoint autenticado está anulando la protección del protocolo. Audita tus APIs autenticadas: si la respuesta depende del usuario, la combinación correcta es `private` (o `no-store`), nunca `public` ni `s-maxage`.

¿Y cuando quieres servir contenido privado *a escala* desde el CDN — vídeos de un curso de pago, descargas de clientes? Ahí el patrón no es cachear la respuesta autenticada, sino separar la autorización de la entrega:

- **URLs firmadas**: tu backend genera una URL con firma y caducidad (CloudFront signed URLs, S3 presigned, tokens de Fastly/Cloudflare); el CDN valida la firma en el borde y sirve el objeto cacheado. La autorización ocurre una vez en tu servidor; la entrega, millones de veces en el borde.
- **Cookies firmadas** para proteger familias enteras de rutas (`/curso-premium/*`) sin firmar cada URL.

Este diseño mantiene el objeto cacheado una sola vez (la clave no incluye nada por usuario) mientras el acceso sigue controlado. Es la diferencia entre «cachear contenido privado» (peligroso) y «controlar el acceso a contenido cacheado» (correcto).

Request collapsing y micro-caching: sobrevivir a la estampida

¿Qué pasa cuando una URL popular expira y 10.000 clientes la piden en el mismo segundo? Sin protección, todas las peticiones fallan la caché a la vez y golpean tu origen: es el *cache stampede* aplicado a HTTP. Las cachés serias lo mitigan con **request collapsing** (coalescing): la primera petición viaja al origen y las demás esperan y comparten esa única respuesta. CloudFront lo hace automáticamente entre peticiones que comparten clave de caché, y su Origin Shield añade una capa regional centralizada que colapsa peticiones entre nodos, dejando tu origen en «tan pocas como una petición por objeto». Varnish y Fastly lo llevan en el ADN del diseño; en Cloudflare, el tiered caching cumple un papel análogo.

En tu propio nginx, el mismo concepto se activa con dos directivas:

```
proxy_cache_path /var/cache/nginx keys_zone=api_cache:10m max_size=1g inactive=10m;

server {
  location /api/ {
    proxy_pass http://backend;
    proxy_cache api_cache;

    # Micro-caching: 1 segundo de caché convierte 5.000 req/s en ~1 req/s al backend
    proxy_cache_valid 200 1s;

    # Collapsing: solo una petición viaja al origen por clave; el resto espera
    proxy_cache_lock on;
    proxy_cache_lock_timeout 5s;

    # Sirve stale mientras se refresca en segundo plano, y también ante errores
    proxy_cache_use_stale updating error timeout http_500 http_502 http_503;
    proxy_cache_background_update on;

    add_header X-Cache-Status $upstream_cache_status;
  }
}
```

El **micro-caching** — TTLs de 1 a 5 segundos — es la técnica más rentable para contenido «dinámico pero no personalizado»: portadas, listados, resultados públicos. Un segundo de caché es invisible para el usuario, pero bajo carga significa que tu backend responde una vez por segundo en lugar de miles. Combinado con proxy_cache_lock (collapsing) y proxy_cache_use_stale updating (SWR casero), tienes un escudo de tres capas en veinte líneas de configuración.

CDNs: s-maxage, invalidación y cabeceras dirigidas

Un CDN es una caché compartida, así que obedece s-maxage por encima de max-age. La estrategia habitual es asimétrica: TTL corto o nulo para el navegador (no puedes purgarlo) y TTL generoso para el CDN (sí puedes). Al desplegar, invalida por URL o por etiquetas de caché (cache tags / surrogate keys) en lugar de purgarlo todo, y confirma que tu CDN respeta stale-while-revalidate: Cloudflare, Fastly y CloudFront lo soportan, pero en algunos requiere activarlo en la configuración de la política de caché.

Cuando necesites dar instrucciones distintas al CDN y al navegador sin trucos, existe una cabecera estándar para ello: CDN-Cache-Control (RFC 9213), que las cachés de CDN obedecen con prioridad sobre Cache-Control. Cloudflare ofrece además la variante con ámbito de proveedor Cloudflare-CDN-Cache-Control, que solo consumen sus nodos, y Fastly, por historia, usa también Surrogate-Control. Un ejemplo que lo junta todo:

```
HTTP/1.1 200 OK
Cache-Control: private, max-age=0, must-revalidate     # navegador: revalida siempre
CDN-Cache-Control: max-age=300, stale-while-revalidate=600   # CDN: 5 min + SWR
Surrogate-Key: product-123 catalog                    # tags para purga selectiva (Fastly)
```

Las **surrogate keys / cache tags** merecen un párrafo propio: etiquetas cada respuesta con los identificadores de las entidades que contiene (product-123, category-shoes), y cuando el producto 123 cambia, purgas la etiqueta y caen a la vez la ficha, los listados y los feeds que lo incluían — en Fastly la purga por key se propaga globalmente en unos 150 ms. Cloudflare (planes Enterprise) y otros ofrecen purge by tag equivalente. Integra la purga en el evento de dominio («producto actualizado») y no en el pipeline de deploy: la invalidación pertenece al dato, no a la release.

Y recuerda: configura TTLs explícitos en cabeceras en lugar de depender del «default TTL» del panel del CDN, que convierte tu política de caché en configuración invisible fuera del repositorio.

La caché del navegador por dentro: memoria, disco y bfcache

«La caché del navegador» son en realidad varias cachés apiladas, y distinguirlas ayuda a depurar:

- **Memory cache**: vive dentro de la pestaña, dura lo que dura la página y es la razón de que una imagen repetida en el mismo documento solo se pida una vez. No respeta del todo tus cabeceras: es una optimización interna del motor.
- **Disk cache (la caché HTTP)**: la que gobiernan Cache-Control y los validadores. Persiste entre sesiones y está particionada por sitio de nivel superior en los navegadores modernos (un mismo recurso cacheado en el contexto de sitio-a.com no se reutiliza embebido en sitio-b.com; el «cache probing» entre sitios murió con esa partición).
- **Back/forward cache (bfcache)**: al navegar atrás o adelante, el navegador puede restaurar la página entera congelada en memoria — DOM, JS y scroll incluidos — sin tocar la red ni la caché HTTP. Lo rompen cosas como unload handlers o Cache-Control: no-store en el documento. Si tu HTML puede permitírselo, evita no-store a nivel de documento y usa no-cache: mantiene el control de frescura sin sacrificar el bfcache.

En DevTools, la columna Size de la pestaña Network te dice de dónde salió cada respuesta: (memory cache), (disk cache) o el tamaño transferido real. Y un matiz que confunde en las pruebas: el botón de recarga envía revalidaciones (max-age=0) para el documento y sus subrecursos aunque estén frescos — el comportamiento «puro» de caché solo lo ves navegando con enlaces o abriendo la URL en una pestaña nueva. Probar caching a base de F5 es la receta clásica para concluir, erróneamente, que «no funciona».

Una herramienta más del navegador: la cabecera de respuesta Clear-Site-Data: "cache" vacía la caché HTTP de tu origen en el cliente. Es el botón de emergencia si has cacheado algo con un TTL desastroso: no llega a los CDN, pero sí a cada navegador que vuelva a visitarte.

Service workers: cuando necesitas programar la caché

Cuando las cabeceras se quedan cortas — offline, precarga de una shell de aplicación, políticas por tipo de petición — el service worker te da una caché programable (la Cache Storage API) interpuesta entre la página y la red. Las tres estrategias que cubren el 95 % de los casos:

- **Cache-first** para assets versionados: mira la caché, y solo si falta, ve a la red.
- **Network-first** para HTML o datos que deben estar al día: intenta la red con timeout, cae a la caché si no hay conexión.
- **Stale-while-revalidate** para todo lo intermedio: responde desde caché al instante y actualiza en segundo plano.

Con Workbox, cada estrategia es una línea de configuración:

```
import { registerRoute } from "workbox-routing";
import { CacheFirst, NetworkFirst, StaleWhileRevalidate } from "workbox-strategies";
import { ExpirationPlugin } from "workbox-expiration";

// Assets con hash: cache-first, máximo 60 entradas
registerRoute(
  ({ url }) =&gt; url.pathname.startsWith("/assets/"),
  new CacheFirst({
    cacheName: "static-assets",
    plugins: [new ExpirationPlugin({ maxEntries: 60, maxAgeSeconds: 30 * 86400 })],
  })
);

// API de catálogo: SWR programático, funciona incluso offline
registerRoute(
  ({ url }) =&gt; url.pathname.startsWith("/api/catalog"),
  new StaleWhileRevalidate({ cacheName: "api-catalog" })
);

// Navegaciones (HTML): red primero con timeout de 3 s, caché como fallback
registerRoute(
  ({ request }) =&gt; request.mode === "navigate",
  new NetworkFirst({ cacheName: "pages", networkTimeoutSeconds: 3 })
);
```

Dos advertencias de producción. Primera: el service worker no reemplaza a la caché HTTP; se apoya en ella (sus fetch pasan por la caché del navegador salvo que lo impidas). Diseña las cabeceras primero y añade el service worker como capa de control extra. Segunda: un service worker mal versionado es la forma más eficaz de congelar tu aplicación en el pasado — respeta el ciclo de actualización (instala, espera, activa) y jamás caches el propio fichero sw.js con TTL largo.

Compresión y caché: Vary: Accept-Encoding, brotli y zstd

La compresión interactúa con la caché en la clave: si el origen comprime, la respuesta gzip y la respuesta brotli del mismo recurso son variantes distintas, y sin Vary: Accept-Encoding una caché intermedia puede servir brotli a un cliente que no lo soporta. Los CDNs modernos lo resuelven normalizando: agrupan los infinitos valores posibles de Accept-Encoding en dos o tres clases (sin compresión, gzip, brotli/zstd) para no fragmentar la caché, y muchos descomprimen/recomprimen en el borde según el cliente.

El estado del arte en 2026: brotli es universal en navegadores y CDNs (mejor ratio que gzip, especialmente en texto), y zstd — Content- Encoding: zstd — está soportado en Chrome/ Edge (desde la 123) y Firefox, con compresión notablemente más rápida en el servidor a ratios comparables. La configuración rentable: comprime estáticos *en build* con brotli a nivel máximo (y sirve los ficheros precomprimidos), usa compresión dinámica con nivel moderado para APIs, y deja siempre Vary: Accept-Encoding en todo lo comprimible. Un detalle que muerde: no comprimas respuestas ya comprimidas (imágenes, vídeo, fuentes woff2) — quemas CPU para ganar bytes negativos y, de paso, degradas ETags fuertes a débiles en nginx.

GraphQL y otras APIs difíciles de cachear

El caching HTTP está diseñado alrededor de GET sobre URLs estables, y GraphQL rompe ambas cosas: todo viaja por POST a /graphql y la «identidad» de la consulta vive en el cuerpo. Las cachés compartidas no cachean POST, así que un API GraphQL queda, por defecto, fuera de toda la infraestructura de este artículo. Las salidas conocidas:

- **GET para consultas**: la spec permite enviar la query en la URL (/graphql?query=...&variables=...). Funciona con cachés estándar, pero las URLs se hacen enormes y filtran la consulta a logs.
- **Persisted queries / APQ**: el cliente registra (o acuerda por hash) las consultas y luego pide GET /graphql?extensions={"persistedQuery":{"sha256Hash":"..."}}. URLs cortas, estables y cacheables en CDN — es el patrón que usan Apollo y Relay en producción, y la única vía razonable de poner un CDN delante de GraphQL.
- **Cachear detrás del resolver**: cuando ni eso es viable, la caché baja a la capa de datos (DataLoader, Redis), fuera del alcance de HTTP.

La misma lógica aplica a cualquier «RPC sobre POST»: si quieres que la Web cachee por ti, dale URLs estables y método GET a tus lecturas. Es un argumento de arquitectura, no solo de protocolo.

Observabilidad: medir el hit ratio antes de presumir de él

Una política de caché sin métricas es una hipótesis. Lo mínimo que quieres ver en un dashboard:

- **Hit ratio del CDN**, global y por ruta. La cabecera de diagnóstico te la da cada proveedor: CF- Cache- Status (Cloudflare), X-Cache (CloudFront), X-Served-By + X-Cache (Fastly). HIT, MISS, EXPIRED, REVALIDATED, STALE — cada estado cuenta una historia distinta; un pico de EXPIRED tras un deploy es normal, un MISS crónico en una ruta caliente es dinero.
- **Tasa de 304 en el origen**: si tus ETags funcionan, una fracción sana de las revalidaciones debe terminar en 304. Una caída repentina delata un ETag inestable (¿un timestamp que se coló en el JSON?) o un Vary nuevo que fragmentó la clave.
- **Age servida**: percentiles de la cabecera Age en respuestas HIT te dicen cuánto «viven» de verdad tus objetos — si el p95 de Age es 4 segundos con un s-maxage de 300, tu caché está siendo purgada o desalojada antes de tiempo (¿claves fragmentadas? ¿capacidad?).

Y convierte las cabeceras en tests. Un contrato de caching se rompe en silencio con un refactor cualquiera; un test de integración lo convierte en un fallo de CI ruidoso:

```
import pytest
from httpx import AsyncClient
from app import app

@pytest.mark.anyio
async def test_catalog_es_cacheable_y_revalidable():
    async with AsyncClient(app=app, base_url="http://test") as client:
        r1 = await client.get("/api/catalog")
        assert r1.status_code == 200
        assert "s-maxage=60" in r1.headers["cache-control"]
        assert "stale-while-revalidate" in r1.headers["cache-control"]
        etag = r1.headers["etag"]

        # La revalidación condicional debe devolver 304 sin cuerpo
        r2 = await client.get("/api/catalog", headers={"If-None-Match": etag})
        assert r2.status_code == 304
        assert r2.headers["etag"] == etag

@pytest.mark.anyio
async def test_perfil_jamas_es_public():
    async with AsyncClient(app=app, base_url="http://test") as client:
        r = await client.get("/api/me", headers={"Authorization": "Bearer test"})
        cc = r.headers.get("cache-control", "")
        assert "private" in cc or "no-store" in cc
        assert "public" not in cc
```

El segundo test es el que evita el incidente de «estoy viendo la cuenta de otro»: cinco líneas que valen una postmortem.

Caso práctico: el catálogo que pasó de 900 a 40 req/s en el origen

Para aterrizarlo todo, un caso realista: una tienda con un catálogo de 30.000 productos, picos de 900 req/s en las páginas de listado y una API JSON que las alimenta. Antes: sin cabeceras explícitas, el CDN aplicaba su TTL por defecto de 2 horas a los estáticos y nada al JSON; cada deploy purgaba todo el CDN «por si acaso»; y las páginas de producto tardaban un p95 de 1,8 s con el origen al 70 % de CPU.

El rediseño, ruta a ruta:

- `/assets/*` (JS, CSS, imágenes con hash): `public, max-age=31536000, immutable`. Hit ratio del 99,6 %.
- `/` y páginas HTML: `no-cache` para el navegador, `CDN-Cache-Control: max-age=60, stale-while-revalidate=300` para el borde. Los deploys llegan al minuto sin purga global.
- `/api/catalog*` (listados, público): `public, max-age=0, s-maxage=120, stale-while-revalidate=600, stale-if-error=86400` + ETag fuerte + `Surrogate-Key` por categoría. La purga por tag al editar un producto invalida exactamente los listados afectados.
- `/api/cart`, `/api/me`: `private, no-store`. Tests de CI que lo verifican.

Resultado tras dos sprints: origen de 900 a ~40 req/s sostenidas (collapsing de CloudFront + Origin Shield absorbió los picos de expiración), p95 de las páginas de listado de 1,8 s a 210 ms, factura de egress del origen reducida un 88 %, y — el efecto que nadie esperaba — el incidente semanal de «catálogo caído a las 9:00» desapareció: era la estampida del TTL por defecto expirando en hora punta, y `stale-while-revalidate` la disolvió.

La lección transversal: nada de esto requirió infraestructura nueva. Fueron cabeceras, una lista blanca de parámetros de query en la cache policy y purga por tags conectada al evento de «producto actualizado».

Depuración sistemática: por qué esta respuesta no se está cacheando

Cuando una ruta no se cachea como esperas, resiste la tentación de tocar cabeceras al azar. Este es el árbol de decisión que resuelve el 90 % de los casos, en orden:

- **1. ¿Qué dice la respuesta?** `curl -sI` contra el origen directamente (saltándote el CDN, con el host apuntado al balanceador). Si no hay `Cache-Control` explícito o hay un `no-store` heredado de un middleware global, ya está el culpable. Los frameworks añaden cabeceras por defecto: Express con `helmet`, Django con middleware de sesión, Rails con `Cache-Control: no-cache` en todo lo que pasa por sesión.
- **2. ¿Hay Set-Cookie?** Es el asesino silencioso número uno: la mayoría de los CDNs no cachean respuestas con `Set-Cookie`. Un middleware de sesión que toca la cookie en cada petición inhabilita el caching de todo el sitio. Sepáralo: la sesión se crea en el login, no en cada GET del catálogo.
- **3. ¿Qué dice el CDN?** Repite el curl contra el CDN y lee su cabecera de diagnóstico (`CF-Cache-Status`, `X-Cache`). `BYPASS` o `DYNAMIC` significan que una regla del panel decidió no cachear (en Cloudflare, el HTML no se cachea por defecto: necesitas una cache rule); `MISS` permanente con cabeceras correctas apunta a fragmentación de clave.
- **4. ¿Fragmentación?** Compara dos peticiones «idénticas» reales (cópialas de los logs, no las inventes). ¿Difieren en query params de tracking, en `Accept-Language` con un `Vary` amplio, en una cookie que entra en la clave? El desglose de variantes del CDN o un `sort | uniq -c` sobre las URLs de los logs lo delata.
- **5. ¿Expulsión temprana?** Si entra en caché pero el `Age` nunca crece, el objeto está siendo desalojado (caché llena, poca demanda por nodo en un CDN muy distribuido) o purgado por una automatización que nadie recuerda. El historial de purgas del proveedor es el último lugar donde todos miran y el primero donde deberían.

Documenta el diagnóstico cuando lo cierres: los bugs de caché reinciden, y el árbol que hoy tardas dos horas en recorrer, la próxima vez son diez minutos.

Frameworks y caché: Next.js, ISR y las cabeceras que emiten por ti

Los metaframeworks toman decisiones de caching por ti, y conviene saber cuáles. Next.js es el ejemplo más elaborado: los assets de `/_next/static` salen con `public, max-age=31536000, immutable` automáticamente (el patrón de oro, gratis); las páginas estáticas prerenderizadas se sirven con `s-maxage=31536000, stale-while-revalidate` entre bastidores; y el modo ISR (Incremental Static Regeneration) es, conceptualmente, stale-while-revalidate aplicado al render: la página se regenera en segundo plano cuando expira su `revalidate` y los visitantes nunca esperan. En Vercel, la cabecera `Vercel-CDN-Cache-Control` permite dirigir instrucciones solo a su edge, con `CDN-Cache-Control` disponible para CDNs adicionales por encima — la jerarquía de la RFC 9213 aplicada en producto real.

La contrapartida: cuando el framework decide por ti, tus cabeceras manuales pueden ser ignoradas o pisadas. En Next.js, un `Cache-Control` puesto a mano en una ruta prerenderizada es sobrescrito en producción; el control real está en `revalidate`, `dynamic` y las opciones de `fetch` (`next: { revalidate: 60 }`). La regla práctica con cualquier metaframework: antes de escribir cabeceras, lee qué emite por defecto (un `curl -sI` a cada tipo de ruta en producción tarda un minuto) y trabaja con su modelo, no contra él. Y si migras de framework, re-audita: los defaults de caching son de las diferencias menos visibles y más caras.

103 Early Hints, preload y el primer viaje sin caché

La caché no ayuda al primer visitante: su navegador no tiene nada. Para ese viaje frío, el protocolo ofrece otra palanca: 103 Early Hints, una respuesta informativa que el servidor (o el CDN) envía *antes* del 200 definitivo, mientras el backend aún está generando el HTML. En ella viajan cabeceras Link con rel=preload y rel=preconnect, de modo que el navegador empieza a descargar el CSS y el JS críticos en paralelo con el render del servidor:

```
HTTP/1.1 103 Early Hints
Link: &lt;/assets/app.3f9c1b.css&gt;; rel=preload; as=style
Link: &lt;/assets/app.3f9c1b.js&gt;; rel=preload; as=script
Link: &lt;https://cdn.imagenes.com&gt;; rel=preconnect

HTTP/1.1 200 OK
Content-Type: text/html
...
```

Cloudflare y Fastly pueden emitir los Early Hints desde el borde (memorizando los Link de respuestas anteriores), con lo que el aviso llega en milisegundos aunque tu origen tarde 400 ms en renderizar. La sinergia con el caching es directa: los recursos que el hint precarga son justo los assets inmutables con hash — el 103 acelera su *primera* descarga y la caché elimina todas las siguientes. Chrome y Firefox lo soportan sobre HTTP/2 y HTTP/3; Safari llegó más tarde pero lo ha incorporado. Es de las pocas optimizaciones que mejoran el viaje frío sin tocar el código de aplicación: se configura en el borde y se mide en el Largest Contentful Paint.

Imágenes y negociación de contenido: el caso especial de Accept y los Client Hints

Las imágenes son el mayor volumen de bytes cacheables de un sitio típico y tienen su propia complicación: la *negociación*. Un mismo /foto.jpg puede servirse como AVIF a un navegador moderno y como JPEG a uno antiguo, decidido por la cabecera Accept de la petición. Eso obliga a Vary: Accept — y de nuevo a la fragmentación de clave, porque cada navegador envía un Accept ligeramente distinto. Los CDNs de imágenes (Cloudflare Images/Polish, Fastly IO, imgix, Cloudinary) resuelven esto normalizando la negociación en el borde: agrupan los Accept en tres o cuatro clases (AVIF, WebP, JPEG) y cachean una variante por clase, no por User-Agent.

Si sirves imágenes responsive con srcset, cada tamaño es una URL distinta (foto-800w.avif) y el problema desaparece: URLs explícitas cachean mejor que negociación implícita, exactamente el mismo principio que los assets con hash. Los Client Hints (Sec-CH-DPR, Sec-CH-Width) permiten al servidor elegir tamaño según el dispositivo, pero cada hint que uses debe entrar en Vary — poder no es deber, y cada dimensión extra multiplica variantes. La recomendación pragmática: URLs explícitas para los tamaños (decide el navegador con srcset), negociación de *formato* normalizada en el CDN, TTLs largos porque una imagen con URL versionada es inmutable de facto, y stale-if-error generoso — una imagen vieja siempre es mejor que un hueco roto en la página.

Errores comunes que sirven datos equivocados o entierran tu hit rate

- public **en respuestas personalizadas.** El clásico incidente de «estoy viendo la cuenta de otra persona». Si la respuesta depende de quién pregunta, es private o no-store, sin excepciones.
- **Confiar en la frescura heurística.** Sin Cache-Control ni Expires, las cachés pueden inventarse un TTL (típicamente el 10 % de la edad según Last-Modified). Contenido viejo servido sin que nadie lo haya pedido. Sé siempre explícito.
- **ETags de metadatos con múltiples servidores.** Cada servidor emite un validador distinto y los 304

desaparecen. Hashea el contenido.

- **Olvidar** `Vary: Accept-Encoding` cuando el origen comprime: una caché intermedia puede entregar gzip a un cliente que no lo entiende.
- `Set-Cookie` **en respuestas cacheables.** Muchas CDNs se niegan a cachearlas por seguridad, y si alguna la guarda, estás compartiendo cookies entre usuarios. Sepáralo.
- **Purga total en cada deploy.** Vacías el CDN entero, el origen recibe la estampida y el TTFB se dispara. Invalida solo lo que cambió, con purge por URL o por tags.

Varnish y la caché reversa en tu propia infraestructura

No todo el mundo quiere (o puede) delegar la capa compartida en un CDN comercial: tráfico interno, requisitos de residencia de datos, o simplemente control fino. Varnish es el proxy de caché especializado de referencia, y dos de sus ideas merecen conocerse aunque nunca lo despliegues, porque nombran conceptos que ya has visto en este artículo.

La primera es el **grace mode**: el equivalente configurable de `stale-while-revalidate`, anterior a la RFC. Cada objeto guarda, además de su TTL, un periodo de «gracia» durante el cual Varnish puede servirlo caducado mientras una única petición en segundo plano lo renueva. TTL corto + gracia larga es la combinación clásica para contenido semi-dinámico: frescura de segundos, resiliencia de horas.

```
sub vcl_backend_response {
  set beresp.ttl = 60s;     # fresco un minuto
  set beresp.grace = 6h;    # servible en gracia hasta 6 horas si hace falta
}
```

La segunda es que el **request coalescing es el comportamiento por defecto**: las peticiones concurrentes a un objeto ausente esperan en una lista mientras una sola viaja al backend. La estampida que en nginx activas con `proxy_cache_lock`, en Varnish viene de serie. Añade la purga por etiquetas (con `xkey`, el módulo del que descienden las surrogate keys de Fastly, que está construido sobre Varnish) y tienes, en tu propio rack, el mismo modelo operativo que un CDN: TTLs agresivos, gracia generosa e invalidación quirúrgica por evento de dominio. El coste es operarlo tú: memoria dimensionada, monitorización del hit ratio con `varnishstat` y disciplina de VCL en revisión de código, porque un VCL es código de producción con capacidad de servir datos equivocados a todos tus usuarios a la vez.

Glosario rápido

- **Frescura (freshness)**: periodo en que una respuesta cacheada puede servirse sin consultar al origen.
- **Validación**: comprobación condicional (ETag/Last-Modified) de que una copia caducada sigue siendo válida, resuelta con un 304.
- **Caché privada**: la del navegador de un usuario. Caché **compartida**: CDN o proxy que sirve a muchos usuarios; cambia las reglas de qué es seguro almacenar.
- **Clave de caché**: identidad bajo la que se guarda una respuesta (método + URL + variantes de `Vary`, más lo que configure el CDN).
- **Variante**: cada copia alternativa de una misma URL creada por `Vary` (gzip vs brotli, idiomas).
- **TTL**: tiempo de vida de la frescura (`max-age`, `s-maxage` o TTL configurado en el CDN).
- **Purga / invalidación** : expulsión explícita de objetos de una caché compartida, por URL o por etiqueta

(surrogate key / cache tag).

- **Request collapsing / coalescing**: técnica por la que peticiones concurrentes al mismo objeto ausente comparten un único viaje al origen.
- **Stampede (estampida)**: avalancha de peticiones al origen cuando un objeto popular expira sin protección de collapsing ni SWR.
- **Micro-caching**: TTLs de segundos sobre contenido dinámico no personalizado para desacoplar el backend del volumen de tráfico.
- **Origin Shield / tiered caching**: capa intermedia centralizada de un CDN que agrega los misses de muchos nodos antes de llegar al origen.
- **bfcache**: restauración instantánea de páginas completas al navegar atrás/ adelante, independiente de la caché HTTP.

Preguntas frecuentes

¿Qué gana `Expires` frente a `max-age`? Nada: `max-age` tiene prioridad si aparecen ambas, y `Expires` depende de relojes absolutos. Existe por compatibilidad; escríbelo solo si debes soportar clientes HTTP/1.0, es decir, casi nunca.

¿Puedo cachear respuestas de error? Con cuidado. Un 404 cacheado unos segundos (o `stale-if-error` para 5xx) protege al origen; un 500 cacheado una hora convierte un blip en una caída de una hora. Muchos CDNs cachean errores con TTL corto por defecto — revisa ese valor, es de los que nadie mira hasta el incidente.

¿Cachear HTML de páginas con sesión iniciada? Como regla, no en cachés compartidas. El patrón sólido es cachear el HTML anónimo (o el shell de la SPA) en el CDN y personalizar vía API autenticada con `private`, o con edge computing que ensambla fragmentos. Mezclar «HTML con datos del usuario» y «caché compartida» es jugar con fuego.

¿`no-cache` o `max-age=0, must-revalidate`? Prácticamente equivalentes para cachés que cumplen la norma. `no-cache` es más corto y expresa la intención; `must-revalidate` añade la garantía explícita de no servir stale jamás tras expirar.

¿Por qué mi hit rate es bajo con todo bien configurado? Sospecha de la clave: parámetros de tracking fragmentando variantes, `Vary` demasiado amplio, cookies en la clave del CDN, o un long tail de URLs que simplemente no se repiten (una caché no ayuda a lo que solo se pide una vez). El desglose de MISS por URL del CDN responde esto en diez minutos.

¿El caching HTTP sustituye a Redis? No: se apilan. HTTP cachea *respuestas completas* cerca del usuario; Redis cachea *datos* dentro de tu infraestructura para componer respuestas nuevas. La combinación habitual: Redis recorta el coste de generar la respuesta, y HTTP evita generarla siquiera.

¿Cómo pruebo el caching en local sin desplegar? Levanta un nginx o Varnish en Docker delante de tu app y usa curl con `--resolve` para simular el dominio real. Lo que no puedes reproducir en local es el comportamiento del CDN comercial: para eso, un subdominio de staging detrás del mismo CDN con la misma configuración es la única prueba fiable. Presupuesta esa configuración duplicada; es más barata que depurar caching solo en producción.

¿Qué pasa con los datos personales y el RGPD? Una caché compartida es un almacenamiento más: si guardas respuestas con datos personales en un CDN, eso es un encargado de tratamiento y una transferencia que documentar. La solución técnica y legal a la vez: los datos personales viajan siempre con `private`/`no-store`, y lo que se cachea en el borde es contenido anónimo. La frontera de la caché es también una frontera de

cumplimiento.

¿Vale la pena cachear respuestas de LLM o APIs de IA? Las respuestas generativas rara vez son cacheables por HTTP (cada prompt es único y suelen ir por POST con streaming), pero sus *insumos* sí: catálogos de modelos, embeddings de documentos públicos, resultados de moderación por hash del contenido. Y si expones un endpoint de IA con entradas repetidas (autocompletado, clasificación de textos cortos), una clave derivada del hash de la entrada con `s-maxage` corto puede ahorrar llamadas caras al proveedor. El principio no cambia: identifica la parte determinista y dale una URL estable.

¿En qué orden aplico todo esto en un sistema ya en marcha? Primero lo que no puede salir mal: `no-store`/`private` explícito en todo lo autenticado y personalizado, con tests que lo fijen. Segundo, los assets con hash e `immutable`: máximo beneficio, riesgo nulo. Tercero, ETags de contenido en las APIs de lectura para cosechar 304. Cuarto, `s-maxage` + `stale-while-revalidate` en las dos o tres rutas públicas de más tráfico — no en todas: cada ruta cacheada es un contrato que mantener. Y solo al final, purga por tags y afinado de claves, cuando las métricas te digan dónde duele. Cada paso es reversible y observable antes de dar el siguiente; el caching se despliega igual que cualquier otro cambio de riesgo: gradualmente y mirando los dashboards.

Checklist de producción

- Toda respuesta lleva un `Cache-Control` explícito; nada depende de heurísticas.
- Assets versionados por hash: `public, max-age=31536000, immutable`. HTML: `no-cache`.
- Respuestas por usuario: `private`; sensibles: `no-store`. Con `Authorization`: jamás `public` ni `s-maxage`.
- APIs compartidas de lectura intensiva: `s-maxage` + `stale-while-revalidate`, y `stale-if-error` como red de seguridad.
- ETags fuertes derivados del contenido, idénticos entre servidores, con manejo correcto de `If-None-Match` y 304 que reenvía cabeceras.
- `Vary` mínimo pero suficiente: `Accept-Encoding` casi siempre; jamás `*` ni `Cookie`.
- Clave de caché normalizada en el CDN: lista blanca de parámetros de query, tracking fuera.
- Rutas dinámicas marcadas `no-store`/`private` explícito: defensa contra cache deception.
- Escrituras concurrentes protegidas con `If-Match`/412 si expones ETags.
- Invalidación del CDN por URL o cache tags integrada en los eventos de dominio, no solo en el deploy.
- Monitoriza hit ratio, tasa de 304 y percentiles de `Age`; tests de CI para los contratos de cabeceras.

La caché HTTP no sustituye a tu caché de aplicación: la complementa desde una capa que ya pagas y que está más cerca del usuario que cualquier Redis. Configúrala con la misma seriedad que el resto de tu infraestructura y notarás la diferencia en latencia, en factura de egress y en la tranquilidad de los deploys.

Referencias y lecturas

Las fuentes primarias caben en una tarde y valen más que cien posts: la RFC 9111 (HTTP Caching, la semántica completa de este artículo), la RFC 5861 (`stale-while-revalidate` y `stale-if-error`), la RFC 9213 (`CDN-Cache-Control` y las cabeceras dirigidas) y la RFC 8246 (`immutable`). Para la parte ofensiva, la investigación de PortSwigger sobre web cache poisoning y web cache deception es el material de referencia, con laboratorios prácticos incluidos. Y la documentación de caching de tu CDN concreto — cache policies de CloudFront, cache rules de Cloudflare, VCL y surrogate keys de Fastly — es de las pocas documentaciones de proveedor que conviene leer entera: ahí viven los defaults que este artículo te ha enseñado a no aceptar a ciegas.

English Version

Full guide in English. La versión en español precede a esta sección.

The fastest request is the one that never reaches your server

Before you add Redis, read replicas, or more pods, there is a caching layer you have already deployed and are probably underusing: the one built into HTTP itself. Every user's browser, the proxies along the way, and your CDN all know how to store and reuse responses; they just need your server to tell them how, with a handful of headers. Configured well, this layer eliminates entire requests (no network round trip at all) or shrinks them to a body-less 304. Configured badly, it serves one user's data to another, or leaves your customers staring at a stale version of the app for hours after a deploy.

The behavior is standardized in RFC 9111 and is identical across HTTP/1.1, HTTP/2, and HTTP/3, so everything that follows applies to any stack.

Two independent decisions: freshness and validation

All of HTTP caching boils down to two questions, each answered by different headers:

- **How long may I reuse this response without asking?** That is *freshness*, controlled by Cache-Control.
- **Once it expires, how do I check it is still valid without downloading it again?** That is *validation*, handled by ETag and Last-Modified together with conditional requests.

Keeping them separate avoids the protocol's most common confusion: no-cache does not mean "don't cache this." It means "store it, but revalidate with the origin before every use." The directive that actually forbids storage is no-store.

Cache-Control: the directives you will actually use

- max-age=N: the response is fresh for N seconds. While it lasts, the browser serves it from disk or memory without touching the network.
- s-maxage=N: same as max-age, but only for shared caches (CDNs, proxies). It lets you give long TTLs to the CDN, which you can purge, and short TTLs to the browser, which you cannot.
- public / private: private restricts storage to the user's own browser; essential for per-user responses. public allows shared caches to store the response even when they normally would not.
- no-cache: store it, but always revalidate before use. Ideal for HTML.
- no-store: store nothing, anywhere. For sensitive data.
- immutable: promises the content at this URL will never change, so the browser skips revalidation even on reload. Only makes sense with hash-versioned file names.
- must-revalidate: once the response expires, forbids serving it stale even if the origin is down; better an error than an outdated value. Useful for financial or inventory data.

How caches compute a response's age: Age, Date, and freshness arithmetic

To reason about caching incidents you need to know how a cache decides whether something is still fresh. The RFC 9111 rule is simple: a response is fresh while its *age* is lower than its *freshness lifetime* (the applicable max-

age or s-maxage). The age does not start counting when your browser receives the response, but when the origin generated it.

That is where the Age header comes in: every shared cache that forwards a response adds (or increments) Age with the seconds that response has been stored upstream. If your CDN hands you Age: 45 with Cache-Control: max-age=60, your browser will only consider it fresh for 15 more seconds. This explains a phenomenon that puzzles a lot of people: you set max-age=60, yet some clients hit the origin again within seconds — their copies were "born old" because they came from a node that had already been storing them for a while.

You can inspect it with curl in two lines:

```
curl -sI https://your-site.com/api/catalog | grep -i -E 'age|cache-control|etag|x-cache|cf-cache-status'

# Age: 41                    ← seconds stored at the CDN
# Cache-Control: public, s-maxage=60, stale-while-revalidate=300
# CF-Cache-Status: HIT       ← Cloudflare served from cache
```

Two practical consequences. First: if you chain several layers (regional CDN, intermediate cache, browser), TTLs do not add up; the age travels with the response and every layer subtracts it. Second: if your origin emits a skewed Date header (badly synced clock), caches will compute negative or inflated ages and your policy becomes unpredictable. Keep NTP healthy: it is caching infrastructure even if it does not look like it.

stale-while-revalidate: cache speed with nearly fresh data

Defined in RFC 5861, stale-while-revalidate=N is the protocol's most underused directive. When a response expires, the cache may keep serving the old copy for up to N more seconds while it refreshes it in the background. Users always see instant responses, and the data never ages more than a few seconds. Its sibling stale-if-error=N does the same when the origin returns 5xx or stops answering: your CDN becomes a safety net during an incident.

```
Cache-Control: public, max-age=0, s-maxage=60, stale-while-revalidate=300, stale-if-error=86400
```

That line tells the CDN: treat this as fresh for 60 seconds; for up to 5 minutes afterwards, serve it stale while renewing it behind the scenes; and if my origin goes down, keep serving it for up to a full day.

One nuance almost nobody documents: support differs per layer. In the browser, Chrome (75+) and Firefox (68+) apply stale-while-revalidate to subresources, but Safari does not implement it in its HTTP cache, and it does not apply to document navigations. Among CDNs, Cloudflare, Fastly, and CloudFront support it, though CloudFront requires enabling it in the cache policy. The right attitude is to treat it as a progressive enhancement: caches that understand it fly, and those that do not simply fall back to max-age.

ETags and conditional requests: the 304 that saves bandwidth

When a response expires, the cache does not have to download it again. If the server included an ETag (an opaque identifier of the content), the client can send If-None-Match with that value. If the content has not changed, the server answers 304 Not Modified with no body, the cache renews the freshness of its copy, and the user skips the full transfer.

Two details matter in production. First: derive the ETag from the content (a hash of the body), not from filesystem metadata. Apache's classic ETag included the inode, and nginx's uses mtime plus size: with several servers behind a load balancer, each one emits a different ETag for the same file and your 304 rate collapses without any dashboard telling you. Second: a 304 must resend the headers that update the stored response

(ETag, Cache-Control), not just the status code.

Here is an endpoint with ETag and conditional revalidation in FastAPI:

```
import hashlib, json
from fastapi import FastAPI, Request, Response

app = FastAPI()
CACHE_HEADERS = {"Cache-Control": "private, max-age=30, stale-while-revalidate=120"}

@app.get("/api/products")
async def list_products(request: Request):
    products = await fetch_products()  # your real query
    body = json.dumps(products, sort_keys=True)

    # Strong ETag derived from content: stable across servers
    etag = "" + hashlib.sha256(body.encode()).hexdigest()[:32] + ""

    if request.headers.get("if-none-match") == etag:
        # 304: no body, but resending the headers that refresh the stored copy
        return Response(status_code=304, headers={"ETag": etag, **CACHE_HEADERS})

    return Response(
        content=body,
        media_type="application/json",
        headers={"ETag": etag, **CACHE_HEADERS},
    )
```

The hash is computed over the JSON serialized with sorted keys, so any two servers produce the same ETag for the same data. The cost of a SHA-256 is negligible compared with transferring the whole body.

Strong ETags, weak ETags, and Last-Modified: choosing your validator

HTTP defines two kinds of validator and it pays to choose deliberately. A *strong* ETag ("abc123") asserts the body is byte-for-byte identical; it is the only validator usable for resuming downloads with Range. A *weak* ETag (W/"abc123") only promises semantic equivalence: useful when the body changes in irrelevant ways across servers, for example JSON with render timestamps, or compression that produces different bytes for the same content.

Last-Modified with If-Modified-Since is the veteran validator: weaker (one-second resolution, and any rewrite of the file "invalidates" it even if the content is identical), but free if you serve static files. The recommended practice is to send both when you can; if the client sends both, If-None-Match takes precedence.

Quick rules that prevent surprises:

- Generated content (JSON, rendered HTML): strong ETag from a hash of the body. Deterministic across servers.
- Static files served by nginx/Apache on a single server: default Last-Modified + ETag are fine; with multiple servers, keep mtimes in sync at deploy time (rsync preserves them, a Docker COPY may not) or disable metadata ETags.
- If you enable dynamic compression, remember nginx downgrades the ETag to weak when compressing on the fly: consistent with the semantics, but it can break ranged downloads.

Optimistic concurrency: If-Match and the 412 that saves your writes

Validators are not just for cheaper reads: they also prevent the *lost update* on writes. The pattern is optimistic concurrency over HTTP: the client reads a resource and receives its ETag; when it sends the PUT / PATCH, it includes If-Match with that ETag; if another client modified the resource in between, the ETag no longer matches and the server answers 412 Precondition Failed instead of overwriting the changes.

```
from fastapi import FastAPI, Request, Response, HTTPException

@app.put("/api/documents/{doc_id}")
async def update_document(doc_id: str, request: Request):
    doc = await get_document(doc_id)
    if doc is None:
        raise HTTPException(404)

    current_etag = compute_etag(doc)        # same hash you use in the GET
    client_etag = request.headers.get("if-match")

    if client_etag is None:
        # Forcing clients to send If-Match prevents accidental overwrites
        raise HTTPException(428, detail="If-Match required")
    if client_etag != current_etag:
        raise HTTPException(412, detail="Document changed; reload and retry")

    payload = await request.json()
    updated = await save_document(doc_id, payload)
    return Response(headers={"ETag": compute_etag(updated)})
```

Status 428 (Precondition Required) exists precisely to declare that your API demands conditional requests. This pattern turns a silent race condition — two editors saving at once, the second erasing the first's work — into an explicit error the UI can resolve by showing a diff or asking to reload. If you already generate ETags for caching, optimistic concurrency comes almost for free.

The golden pattern for SPAs: immutable assets, HTML that always revalidates

Modern bundlers (Vite, webpack, esbuild) write a content hash into every file name: app.3f9c1b.js. If the content changes, the URL changes. That enables the most effective division of labor in all of web caching:

- **Hashed assets** : Cache- Control: public, max- age=31536000, immutable . A year of caching and not a single revalidation. The URL is the versioning.
- **The HTML**: Cache-Control: no-cache. Revalidated on every navigation, so each deploy lands instantly: new HTML references the new bundles.

```
# nginx
location /assets/ {
    add_header Cache-Control "public, max-age=31536000, immutable";
}

location = /index.html {
    add_header Cache-Control "no-cache";
}
```

The inverse mistake is the classic SPA outage: HTML with a long max- age. After a deploy, users keep an old index.html that requests bundles which no longer exist on the server, and the result is a blank screen until someone thinks to "clear the cache."

APIs: what to cache, what not to, and the role of Vary

- **Sensitive or ephemeral data** (sessions, tokens, medical or payment data): `Cache-Control: no-store`.
- **Per-user responses** (their cart, their profile): `private, max-age=30` or similar. Never `public`: a shared cache would serve one user's cart to another.
- **Shared, semi-static data** (catalog, configuration, editorial content): `public, s-maxage=60, stale-while-revalidate=300`, and let the CDN absorb the traffic.

The `Vary` header tells caches which request headers are part of the cache key. If you serve compressed responses, use `Vary: Accept-Encoding`; if the response depends on language, add `Accept-Language`. Beware the opposite extreme: `Vary: Cookie` or `Vary: *` fragment the cache into so many variants that the hit rate drops to zero.

The same complete pattern in Express, covering assets, HTML, and a cacheable API:

```
const express = require("express");
const crypto = require("node:crypto");
const app = express();

// Hash-named assets: cache for a year, never revalidate
app.use("/assets", express.static("dist/assets", {
  immutable: true,
  maxAge: "1y",
}));

// HTML: always revalidate, so every deploy lands instantly
app.get("/", (req, res) => {
  res.set("Cache-Control", "no-cache");
  res.sendFile("index.html", { root: "dist" });
});

// Semi-static API: 60 s of CDN freshness + revalidation window
app.get("/api/catalog", async (req, res) => {
  const body = JSON.stringify(await getCatalog());
  const etag =
    '"' + crypto.createHash("sha256").update(body).digest("hex").slice(0, 32) + '"';

  res.set({
    ETag: etag,
    "Cache-Control": "public, max-age=0, s-maxage=60, stale-while-revalidate=300",
    Vary: "Accept-Encoding",
  });

  if (req.get("If-None-Match") === etag) return res.status(304).end();
  res.type("application/json").send(body);
});
```

The cache key: query strings, normalization, and why parameter order matters

Every cache indexes by a *key*: usually method + host + path + query string, plus whatever `Vary` adds. Understanding the key explains half of all disappointing hit rates.

The typical case: `/api/catalog?page=2&size=20` and `/api/catalog?size=20&page=2` are the same query for your backend, but two different entries for most caches. Add a tracking parameter (`utm_source`, `fbclid`, `gclid`) and every marketing campaign fragments your cache into thousands of variants that are never reused. The defenses, in order of effectiveness:

- **Normalize at the edge.** CDNs let you define which parameters belong to the key (CloudFront cache policies, Cloudflare cache rules, Fastly VCL). Declare the allowlist of parameters that actually change the response and sort them alphabetically; the rest (tracking) is ignored for the key and forwarded to the origin if needed.
- **Generate canonical URLs.** If your frontend always builds the query string in the same order, fragmentation disappears at the source. A utility function that sorts `URLSearchParams` costs five lines.
- **Never put per-user data in the URL of shared resources.** A `?token=...` on an image URL turns every user into a cache variant (and leaks the token into logs and Referers while it is at it).

The mental rule: the cache key defines what is shared. Everything in the key fragments; everything that affects the response and is NOT in the key (or in `Vary`) is a cross-user data bug waiting for its moment.

Security: cache poisoning and cache deception, the two attacks you must know

A shared cache is an attack surface: if an attacker can get the cache to store a manipulated response under a legitimate key, every subsequent visitor receives it. These are the two attacks documented extensively by PortSwigger's research, and they are prevented with configuration discipline, not with a WAF.

Web cache poisoning exploits *unkeyed inputs*: headers like `X-Forwarded-Host` that the backend uses to generate the response (for example, absolute URLs in the HTML) but that the cache does not include in the key. The attacker sends a request with the poisoned header; the backend generates HTML pointing at a malicious domain; the cache stores it under the normal URL; and every following user receives that HTML. Defenses: do not use request headers to build cacheable responses (or include them in `Vary`), disable support for override headers you do not need, and cache only what you explicitly declare cacheable.

Web cache deception is the inverse: tricking the cache into storing private content. The classic: the attacker gets the victim to visit `/my-account/profile.css`; the backend ignores the suffix and responds with the private profile HTML; the CDN sees the `.css` extension, applies its "static files are cacheable" rule, and stores the victim's private page, which the attacker later retrieves by visiting the same URL. Defenses: mark everything dynamic with an explicit `Cache-Control: private, no-store`, configure the CDN to honor origin headers over extension-based rules, and reject (404/redirect) paths with suffixes your application does not recognize instead of answering "helpfully."

The shared moral: be explicit. A response without `Cache-Control` is an invitation for every intermediate layer to decide for you, and a cache's default decisions never incorporate your threat model.

Authorization, private content, and shared caches

RFC 9111 contains a rule that surprises people: by default, a shared cache **must not store** responses to requests that carried an `Authorization` header… unless the response includes directives that explicitly re-allow it: `public`, `s-maxage`, or `must-revalidate`. In other words, that `public` you added "just in case" to an authenticated endpoint is disabling the protocol's built-in protection. Audit your authenticated APIs: if the response depends on the user, the correct combination is `private` (or `no-store`), never `public` or `s-maxage`.

And when you do want to serve private content *at scale* from the CDN — videos of a paid course, customer downloads? The pattern there is not to cache the authenticated response but to separate authorization from delivery:

- **Signed URLs**: your backend generates a URL with a signature and an expiry (CloudFront signed URLs, S3 presigned, Fastly/ Cloudflare tokens); the CDN validates the signature at the edge and serves the cached object. Authorization happens once on your server; delivery happens millions of times at the edge.
- **Signed cookies** to protect whole families of routes (`/premium-course/*`) without signing each URL.

This design keeps the object cached exactly once (the key contains nothing per-user) while access remains controlled. It is the difference between "caching private content" (dangerous) and "controlling access to cached content" (correct).

Request collapsing and micro-caching: surviving the stampede

What happens when a popular URL expires and 10,000 clients request it in the same second? Without protection, all requests miss the cache at once and hit your origin: it is the *cache stampede* applied to HTTP. Serious caches mitigate it with **request collapsing** (coalescing): the first request travels to the origin and the rest wait and share that single response. CloudFront does it automatically among requests that share a cache key, and its Origin Shield adds a centralized regional layer that collapses requests across nodes, leaving your origin with "as few as one request per object." Varnish and Fastly have it in their design DNA; on Cloudflare, tiered caching plays the analogous role.

In your own nginx, the same concept is enabled with two directives:

```
proxy_cache_path /var/cache/nginx keys_zone=api_cache:10m max_size=1g inactive=10m;

server {
  location /api/ {
    proxy_pass http://backend;
    proxy_cache api_cache;

    # Micro-caching: 1 second of cache turns 5,000 req/s into ~1 req/s at the backend
    proxy_cache_valid 200 1s;

    # Collapsing: only one request travels to the origin per key; the rest wait
    proxy_cache_lock on;
    proxy_cache_lock_timeout 5s;

    # Serve stale while refreshing in the background, and on errors too
    proxy_cache_use_stale updating error timeout http_500 http_502 http_503;
    proxy_cache_background_update on;

    add_header X-Cache-Status $upstream_cache_status;
  }
}
```

Micro-caching — TTLs of 1 to 5 seconds — is the most profitable technique for content that is "dynamic but not personalized": home pages, listings, public results. One second of cache is invisible to the user, but under load it means your backend answers once per second instead of thousands of times. Combined with `proxy_cache_lock` (collapsing) and `proxy_cache_use_stale updating` (homegrown SWR), you get a three-layer shield in twenty lines of configuration.

CDNs: s-maxage, invalidation, and targeted headers

A CDN is a shared cache, so it honors `s-maxage` over `max-age`. The usual strategy is asymmetric: a short or zero TTL for the browser (which you cannot purge) and a generous TTL for the CDN (which you can). On deploy, invalidate by URL or by cache tags (surrogate keys) instead of purging everything, and confirm your CDN honors `stale-while-revalidate`: Cloudflare, Fastly, and CloudFront support it, but some require enabling it in the cache policy configuration.

When you need to give the CDN and the browser different instructions without tricks, there is a standard header for it: `CDN-Cache-Control` (RFC 9213), which CDN caches obey with priority over `Cache-Control`. Cloudflare

additionally offers the vendor-scoped variant Cloudflare-CDN-Cache-Control, consumed only by its own nodes, and Fastly, for historical reasons, also uses Surrogate-Control. One example that puts it all together:

```
HTTP/1.1 200 OK
Cache-Control: private, max-age=0, must-revalidate     # browser: always revalidate
CDN-Cache-Control: max-age=300, stale-while-revalidate=600   # CDN: 5 min + SWR
Surrogate-Key: product-123 catalog                    # tags for selective purge (Fastly)
```

Surrogate keys / cache tags deserve their own paragraph: tag every response with the identifiers of the entities it contains (product-123, category-shoes), and when product 123 changes, you purge the tag and the product page, the listings, and the feeds that included it all drop at once — on Fastly, a purge by key propagates globally in about 150 ms. Cloudflare (Enterprise plans) and others offer equivalent purge-by-tag. Wire the purge into the domain event ("product updated") rather than the deploy pipeline: invalidation belongs to the data, not to the release.

And remember: set explicit TTLs in headers instead of relying on the CDN dashboard's "default TTL," which turns your caching policy into invisible configuration living outside the repository.

Inside the browser cache: memory, disk, and the bfcache

"The browser cache" is actually several stacked caches, and telling them apart helps when debugging:

- **Memory cache**: lives inside the tab, lasts as long as the page does, and is the reason an image repeated in the same document is only requested once. It does not fully honor your headers: it is an internal engine optimization.
- **Disk cache (the HTTP cache)**: the one governed by Cache-Control and validators. It persists across sessions and is partitioned by top-level site in modern browsers (the same resource cached in the context of site-a.com is not reused when embedded in site-b.com; cross-site "cache probing" died with that partitioning).
- **Back/forward cache (bfcache)**: when navigating back or forward, the browser can restore the entire page frozen in memory — DOM, JS, and scroll included — without touching the network or the HTTP cache. Things like unload handlers or Cache-Control: no-store on the document break it. If your HTML can afford it, avoid no-store at the document level and use no-cache: you keep freshness control without sacrificing the bfcache.

In DevTools, the Size column of the Network tab tells you where each response came from: (memory cache), (disk cache) , or the actual transferred size. And one nuance that confuses testing: the reload button sends revalidations (max-age=0) for the document and its subresources even if they are fresh — you only see "pure" cache behavior by navigating through links or opening the URL in a new tab. Testing caching by hammering F5 is the classic recipe for wrongly concluding it "doesn't work."

One more browser tool: the Clear-Site-Data: "cache" response header empties your origin's HTTP cache on the client. It is the emergency button if you cached something with a disastrous TTL: it does not reach CDNs, but it does reach every browser that visits you again.

Service workers: when you need to program the cache

When headers fall short — offline support, precaching an app shell, per-request-type policies — a service worker gives you a programmable cache (the Cache Storage API) interposed between the page and the network. The three strategies that cover 95% of cases:

- **Cache-first** for versioned assets: check the cache, and only if missing, go to the network.

- **Network-first** for HTML or data that must be current: try the network with a timeout, fall back to the cache when offline.
- **Stale-while-revalidate** for everything in between: answer from cache instantly and update in the background.

With Workbox, each strategy is one line of configuration:

```
import { registerRoute } from "workbox-routing";
import { CacheFirst, NetworkFirst, StaleWhileRevalidate } from "workbox-strategies";
import { ExpirationPlugin } from "workbox-expiration";

// Hashed assets: cache-first, at most 60 entries
registerRoute(
  ({ url }) =&gt; url.pathname.startsWith("/assets/"),
  new CacheFirst({
    cacheName: "static-assets",
    plugins: [new ExpirationPlugin({ maxEntries: 60, maxAgeSeconds: 30 * 86400 })],
  })
);

// Catalog API: programmatic SWR, works even offline
registerRoute(
  ({ url }) =&gt; url.pathname.startsWith("/api/catalog"),
  new StaleWhileRevalidate({ cacheName: "api-catalog" })
);

// Navigations (HTML): network first with a 3 s timeout, cache as fallback
registerRoute(
  ({ request }) =&gt; request.mode === "navigate",
  new NetworkFirst({ cacheName: "pages", networkTimeoutSeconds: 3 })
);
```

Two production warnings. First: the service worker does not replace the HTTP cache; it builds on it (its fetch calls go through the browser cache unless you prevent it). Design the headers first and add the service worker as an extra control layer. Second: a badly versioned service worker is the most effective way to freeze your application in the past — respect the update cycle (install, wait, activate) and never cache the sw.js file itself with a long TTL.

Compression and caching: Vary: Accept-Encoding, brotli, and zstd

Compression interacts with the cache at the key: if the origin compresses, the gzip response and the brotli response of the same resource are different variants, and without Vary: Accept-Encoding an intermediate cache can serve brotli to a client that does not support it. Modern CDNs solve this by normalizing: they group the infinite possible Accept-Encoding values into two or three classes (no compression, gzip, brotli/zstd) to avoid fragmenting the cache, and many decompress/recompress at the edge according to the client.

The state of the art in 2026: brotli is universal across browsers and CDNs (better ratio than gzip, especially on text), and zstd — Content-Encoding: zstd — is supported in Chrome/Edge (since 123) and Firefox, with markedly faster server-side compression at comparable ratios. The profitable setup: compress static files *at build time* with brotli at maximum level (and serve the precompressed files), use dynamic compression at a moderate level for APIs, and always keep Vary: Accept-Encoding on everything compressible. One detail that bites: do not compress already-compressed responses (images, video, woff2 fonts) — you burn CPU to gain negative bytes and, along the way, downgrade strong ETags to weak in nginx.

GraphQL and other hard-to-cache APIs

HTTP caching is designed around GET on stable URLs, and GraphQL breaks both: everything travels via POST to `/graphql` and the "identity" of the query lives in the body. Shared caches do not cache POST, so a GraphQL API sits, by default, outside all the infrastructure in this article. The known ways out:

- **GET for queries**: the spec allows sending the query in the URL (`/graphql?query=...&variables=...`). It works with standard caches, but URLs get huge and leak the query into logs.
- **Persisted queries / APQ**: the client registers (or agrees by hash on) the queries and then requests `GET /graphql?extensions={"persistedQuery":{"sha256Hash":"..."}}`. Short, stable, CDN-cacheable URLs — the pattern Apollo and Relay use in production, and the only reasonable way to put a CDN in front of GraphQL.
- **Caching behind the resolver**: when even that is not viable, caching moves down to the data layer (DataLoader, Redis), out of HTTP's reach.

The same logic applies to any "RPC over POST": if you want the Web to cache for you, give your reads stable URLs and the GET method. It is an architecture argument, not just a protocol one.

Observability: measure your hit ratio before bragging about it

A caching policy without metrics is a hypothesis. The minimum you want on a dashboard:

- **CDN hit ratio**, global and per route. Each provider exposes its diagnostic header: `CF-Cache-Status` (Cloudflare), `X-Cache` (CloudFront), `X-Served-By` + `X-Cache` (Fastly). HIT, MISS, EXPIRED, REVALIDATED, STALE — each state tells a different story; a spike of EXPIRED after a deploy is normal, a chronic MISS on a hot route is money.
- **304 rate at the origin**: if your ETags work, a healthy fraction of revalidations must end in 304. A sudden drop betrays an unstable ETag (did a timestamp sneak into the JSON?) or a new `Vary` that fragmented the key.
- **Served Age**: percentiles of the `Age` header on HIT responses tell you how long your objects really "live" — if the p95 of Age is 4 seconds with an s-maxage of 300, your cache is being purged or evicted early (fragmented keys? capacity?).

And turn headers into tests. A caching contract breaks silently with any refactor; an integration test turns that into a noisy CI failure:

```
import pytest
from httpx import AsyncClient
from app import app

@pytest.mark.anyio
async def test_catalog_is_cacheable_and_revalidatable():
    async with AsyncClient(app=app, base_url="http://test") as client:
        r1 = await client.get("/api/catalog")
        assert r1.status_code == 200
        assert "s-maxage=60" in r1.headers["cache-control"]
        assert "stale-while-revalidate" in r1.headers["cache-control"]
        etag = r1.headers["etag"]

        # Conditional revalidation must return a body-less 304
        r2 = await client.get("/api/catalog", headers={"If-None-Match": etag})
        assert r2.status_code == 304
        assert r2.headers["etag"] == etag

@pytest.mark.anyio
async def test_profile_is_never_public():
    async with AsyncClient(app=app, base_url="http://test") as client:
```

```
    r = await client.get("/api/me", headers={"Authorization": "Bearer test"})
    cc = r.headers.get("cache-control", "")
    assert "private" in cc or "no-store" in cc
    assert "public" not in cc
```

The second test is the one that prevents the "I'm seeing someone else's account" incident: five lines worth a postmortem.

Case study: the catalog that went from 900 to 40 req/s at the origin

To make it all concrete, a realistic case: a store with a 30,000-product catalog, peaks of 900 req/s on listing pages, and a JSON API feeding them. Before: no explicit headers, the CDN applied its 2-hour default TTL to static files and nothing to the JSON; every deploy purged the whole CDN "just in case"; and product pages had a p95 of 1.8 s with the origin at 70% CPU.

The redesign, route by route:

- `/assets/*` (hashed JS, CSS, images): `public, max-age=31536000, immutable`. 99.6% hit ratio.
- `/` and HTML pages: `no-cache` for the browser, `CDN-Cache-Control: max-age=60, stale-while-revalidate=300` for the edge. Deploys land within a minute with no global purge.
- `/api/catalog*` (listings, public): `public, max-age=0, s-maxage=120, stale-while-revalidate=600, stale-if-error=86400` + strong ETag + `Surrogate-Key` per category. Purging the tag when a product is edited invalidates exactly the affected listings.
- `/api/cart`, `/api/me`: `private, no-store`. CI tests that verify it.

Result after two sprints: origin traffic from 900 to ~40 sustained req/s (CloudFront's collapsing + Origin Shield absorbed the expiration peaks), listing-page p95 from 1.8 s to 210 ms, origin egress bill down 88%, and — the effect nobody expected — the weekly "catalog down at 9:00" incident disappeared: it was the default-TTL stampede expiring at rush hour, and `stale-while-revalidate` dissolved it.

The cross-cutting lesson: none of this required new infrastructure. It was headers, a query-parameter allowlist in the cache policy, and tag-based purging wired to the "product updated" event.

Systematic debugging: why is this response not being cached?

When a route does not cache the way you expect, resist the temptation to tweak headers at random. This is the decision tree that solves 90% of cases, in order:

- **1. What does the response say?** `curl -sI` against the origin directly (bypassing the CDN, with the host pointed at the load balancer). If there is no explicit `Cache-Control`, or a `no-store` inherited from a global middleware, you have your culprit. Frameworks add default headers: Express with `helmet`, Django with session middleware, Rails with `Cache-Control: no-cache` on everything that touches the session.
- **2. Is there a Set-Cookie?** It is the number-one silent killer: most CDNs will not cache responses with `Set-Cookie`. A session middleware that touches the cookie on every request disables caching for the whole site. Separate it: the session is created at login, not on every catalog GET.
- **3. What does the CDN say?** Repeat the curl against the CDN and read its diagnostic header (`CF-Cache-Status`, `X-Cache`). `BYPASS` or `DYNAMIC` mean a dashboard rule decided not to cache (on Cloudflare, HTML is not cached by default: you need a cache rule); a permanent `MISS` with correct headers points at key fragmentation.
- **4. Fragmentation?** Compare two real "identical" requests (copy them from the logs, do not invent them). Do

they differ in tracking query params, in `Accept-Language` with a broad `Vary`, in a cookie that is part of the key? The CDN's variant breakdown, or a `sort | uniq -c` over the logged URLs, gives it away.

- **5. Early eviction?** If it enters the cache but `Age` never grows, the object is being evicted (cache full, low per-node demand on a highly distributed CDN) or purged by an automation nobody remembers. The provider's purge history is the last place everyone looks and the first place they should.

Document the diagnosis when you close it: cache bugs reoffend, and the tree that takes you two hours today takes ten minutes next time.

Frameworks and caching: Next.js, ISR, and the headers emitted for you

Metaframeworks make caching decisions on your behalf, and it pays to know which ones. Next.js is the most elaborate example: assets under `/_next/static` ship with `public, max-age=31536000, immutable` automatically (the golden pattern, for free); prerendered static pages are served with `s-maxage=31536000, stale-while-revalidate` behind the scenes; and ISR (Incremental Static Regeneration) is, conceptually, stale-while-revalidate applied to rendering: the page regenerates in the background when its `revalidate` window expires and visitors never wait. On Vercel, the `Vercel-CDN-Cache-Control` header lets you target instructions at their edge only, with `CDN-Cache-Control` available for additional CDNs above it — the RFC 9213 hierarchy applied in a real product.

The flip side: when the framework decides for you, your manual headers may be ignored or overwritten. In Next.js, a hand-set `Cache-Control` on a prerendered route is overridden in production; the real control lives in `revalidate`, `dynamic`, and the `fetch` options (`next: { revalidate: 60 }`). The practical rule with any metaframework: before writing headers, learn what it emits by default (a `curl -sI` against each route type in production takes a minute) and work with its model, not against it. And if you migrate frameworks, re-audit: caching defaults are among the least visible and most expensive differences.

103 Early Hints, preload, and the cold first visit

The cache does not help your first-time visitor: their browser has nothing. For that cold trip, the protocol offers another lever: `103 Early Hints`, an informational response the server (or the CDN) sends *before* the final 200, while the backend is still generating the HTML. It carries `Link` headers with `rel=preload` and `rel=preconnect`, so the browser starts downloading the critical CSS and JS in parallel with the server render:

```
HTTP/1.1 103 Early Hints
Link: &lt;/assets/app.3f9c1b.css&gt;; rel=preload; as=style
Link: &lt;/assets/app.3f9c1b.js&gt;; rel=preload; as=script
Link: &lt;https://cdn.images.com&gt;; rel=preconnect

HTTP/1.1 200 OK
Content-Type: text/html
...
```

Cloudflare and Fastly can emit Early Hints from the edge (memorizing the `Link` headers of previous responses), so the hint arrives in milliseconds even if your origin takes 400 ms to render. The synergy with caching is direct: the resources the hint preloads are precisely the immutable hashed assets — the 103 speeds up their *first* download and the cache eliminates all the following ones. Chrome and Firefox support it over HTTP/2 and HTTP/3; Safari arrived later but has adopted it. It is one of the few optimizations that improve the cold visit without touching application code: you configure it at the edge and measure it in Largest Contentful Paint.

Images and content negotiation: the special case of Accept and Client Hints

Images are the largest volume of cacheable bytes on a typical site and come with their own complication: *negotiation*. The same /photo.jpg can be served as AVIF to a modern browser and as JPEG to an old one, decided by the request's Accept header. That forces Vary: Accept — and once again key fragmentation, because every browser sends a slightly different Accept. Image CDNs (Cloudflare Images/Polish, Fastly IO, imgix, Cloudinary) solve this by normalizing negotiation at the edge: they group Accept values into three or four classes (AVIF, WebP, JPEG) and cache one variant per class, not per User-Agent.

If you serve responsive images with srcset, each size is a distinct URL (photo-800w.avif) and the problem disappears: explicit URLs cache better than implicit negotiation, exactly the same principle as hashed assets. Client Hints (Sec-CH-DPR, Sec-CH-Width) let the server pick a size per device, but every hint you use must go into Vary — capability is not obligation, and every extra dimension multiplies variants. The pragmatic recommendation: explicit URLs for sizes (let the browser decide via srcset), *format* negotiation normalized at the CDN, long TTLs because a version-named image is de facto immutable, and a generous stale-if-error — an old image always beats a broken hole in the page.

Common mistakes that serve the wrong data or bury your hit rate

- public **on personalized responses.** The classic "I'm seeing someone else's account" incident. If the response depends on who is asking, it is private or no-store, no exceptions.
- **Relying on heuristic freshness.** Without Cache-Control or Expires, caches may invent a TTL (typically 10% of the age per Last-Modified). Stale content served without anyone asking for it. Always be explicit.
- **Metadata-based ETags with multiple servers.** Each server emits a different validator and 304s vanish. Hash the content.
- **Forgetting** Vary: Accept-Encoding when the origin compresses: an intermediate cache can hand gzip to a client that cannot decode it.
- Set-Cookie **on cacheable responses.** Many CDNs refuse to cache them for safety, and if one does store it, you are sharing cookies between users. Keep them separate.
- **Full purge on every deploy.** You empty the entire CDN, the origin takes the stampede, and TTFB spikes. Invalidate only what changed, with per-URL or tag-based purging.

Varnish and the reverse cache in your own infrastructure

Not everyone wants (or can) delegate the shared layer to a commercial CDN: internal traffic, data-residency requirements, or simply fine-grained control. Varnish is the reference specialized caching proxy, and two of its ideas are worth knowing even if you never deploy it, because they name concepts you have already seen in this article.

The first is **grace mode**: the configurable equivalent of stale-while-revalidate, predating the RFC. Each object stores, besides its TTL, a "grace" period during which Varnish may serve it expired while a single background request renews it. Short TTL + long grace is the classic combination for semi-dynamic content: freshness measured in seconds, resilience measured in hours.

```
sub vcl_backend_response {
  set beresp.ttl = 60s;     # fresh for one minute
  set beresp.grace = 6h;    # servable in grace for up to 6 hours if needed
}
```

The second is that **request coalescing is the default behavior**: concurrent requests for a missing object wait on a list while a single one travels to the backend. The stampede protection you enable in nginx with proxy_cache_lock comes standard in Varnish. Add tag-based purging (with xkey, the module from which Fastly's surrogate keys descend — Fastly is built on Varnish) and you have, in your own rack, the same operating model as a CDN: aggressive TTLs, generous grace, and surgical invalidation per domain event. The cost is operating it yourself: properly sized memory, hit-ratio monitoring with varnishstat, and VCL discipline in code review, because a VCL is production code with the power to serve wrong data to all your users at once.

Quick glossary

- **Freshness**: the period during which a cached response may be served without consulting the origin.
- **Validation**: the conditional check (ETag/Last-Modified) that an expired copy is still valid, resolved with a 304.
- **Private cache**: a single user's browser cache. **Shared cache**: a CDN or proxy serving many users; it changes the rules of what is safe to store.
- **Cache key**: the identity under which a response is stored (method + URL + Vary variants, plus whatever the CDN configures).
- **Variant**: each alternative copy of the same URL created by Vary (gzip vs brotli, languages).
- **TTL**: the freshness lifetime (max-age, s-maxage, or a TTL configured at the CDN).
- **Purge / invalidation**: explicit eviction of objects from a shared cache, by URL or by tag (surrogate key / cache tag).
- **Request collapsing / coalescing**: the technique by which concurrent requests for the same missing object share a single trip to the origin.
- **Stampede**: the avalanche of origin requests when a popular object expires without collapsing or SWR protection.
- **Micro-caching**: seconds-long TTLs on non-personalized dynamic content to decouple the backend from traffic volume.
- **Origin Shield / tiered caching**: a centralized intermediate CDN layer that aggregates misses from many nodes before they reach the origin.
- **bfcache**: instant restoration of full pages on back/forward navigation, independent of the HTTP cache.

Frequently asked questions

What does Expires offer over max-age? Nothing: max-age wins if both appear, and Expires depends on absolute clocks. It exists for compatibility; write it only if you must support HTTP/1.0 clients, which is to say, almost never.

Can I cache error responses? Carefully. A 404 cached for a few seconds (or stale-if-error for 5xx) protects the origin; a 500 cached for an hour turns a blip into an hour-long outage. Many CDNs cache errors with a short TTL by default — review that value, it is one nobody looks at until the incident.

Caching HTML for logged-in pages? As a rule, not in shared caches. The solid pattern is to cache the anonymous HTML (or the SPA shell) at the CDN and personalize via an authenticated API with private, or with edge computing that assembles fragments. Mixing "HTML with user data" and "shared cache" is playing with fire.

no-cache or max-age=0, must-revalidate? Practically equivalent for spec-compliant caches. no-cache is shorter and

expresses the intent; must-revalidate adds the explicit guarantee of never serving stale after expiry.

Why is my hit rate low with everything configured correctly? Suspect the key: tracking parameters fragmenting variants, an overly broad Vary, cookies in the CDN key, or a long tail of URLs that simply never repeat (a cache cannot help with what is only requested once). The CDN's per-URL MISS breakdown answers this in ten minutes.

Does HTTP caching replace Redis? No: they stack. HTTP caches *complete responses* close to the user; Redis caches *data* inside your infrastructure to compose new responses. The usual combination: Redis cuts the cost of generating the response, and HTTP avoids generating it at all.

How do I test caching locally without deploying? Run an nginx or Varnish in Docker in front of your app and use curl with --resolve to simulate the real domain. What you cannot reproduce locally is the commercial CDN's behavior: for that, a staging subdomain behind the same CDN with the same configuration is the only reliable test. Budget for that duplicated configuration; it is cheaper than debugging caching only in production.

What about personal data and the GDPR? A shared cache is just another storage system: if you store responses with personal data on a CDN, that is a data processor and a transfer to document. The solution is technical and legal at once: personal data always travels with private/no-store, and what is cached at the edge is anonymous content. The cache boundary is also a compliance boundary.

Is it worth caching LLM or AI API responses? Generative responses are rarely HTTP-cacheable (every prompt is unique and they usually go over POST with streaming), but their *inputs* are: model catalogs, embeddings of public documents, moderation results keyed by content hash. And if you expose an AI endpoint with repeated inputs (autocomplete, classification of short texts), a key derived from the input hash with a short s-maxage can save expensive provider calls. The principle does not change: identify the deterministic part and give it a stable URL.

In what order do I apply all this to a running system? First, what must never go wrong: explicit no-store/private on everything authenticated and personalized, with tests pinning it. Second, hashed assets with immutable: maximum benefit, zero risk. Third, content ETags on read APIs to harvest 304s. Fourth, s-maxage + stale-while-revalidate on the two or three highest-traffic public routes — not on all of them: every cached route is a contract to maintain. And only at the end, tag-based purging and key tuning, when the metrics tell you where it hurts. Each step is reversible and observable before taking the next; caching ships like any other risky change: gradually, watching the dashboards.

Production checklist

- Every response carries an explicit Cache-Control; nothing relies on heuristics.
- Hash-versioned assets: public, max-age=31536000, immutable. HTML: no-cache.
- Per-user responses: private; sensitive ones: no-store. With Authorization: never public or s-maxage.
- Read-heavy shared APIs: s-maxage + stale-while-revalidate, with stale-if-error as a safety net.
- Strong content-derived ETags, identical across servers, with correct If-None-Match handling and 304s that resend headers.
- Minimal but sufficient Vary: Accept-Encoding almost always; never * or Cookie.
- Cache key normalized at the CDN: query-parameter allowlist, tracking out.
- Dynamic routes marked with explicit no-store/private: defense against cache deception.
- Concurrent writes protected with If-Match/412 if you expose ETags.
- CDN invalidation by URL or cache tags wired into domain events, not just the deploy.

- Monitor hit ratio, 304 rate, and Age percentiles; CI tests for header contracts.

HTTP caching does not replace your application cache: it complements it from a layer you already pay for, one that sits closer to the user than any Redis ever will. Configure it with the same seriousness as the rest of your infrastructure and you will see the difference in latency, in your egress bill, and in how calm your deploys become.

References and further reading

The primary sources fit in an afternoon and are worth more than a hundred blog posts: RFC 9111 (HTTP Caching, the full semantics behind this article), RFC 5861 (stale-while-revalidate and stale-if-error), RFC 9213 (CDN-Cache-Control and targeted headers), and RFC 8246 (immutable). For the offensive side, PortSwigger's research on web cache poisoning and web cache deception is the reference material, with hands-on labs included. And your specific CDN's caching documentation — CloudFront cache policies, Cloudflare cache rules, Fastly's VCL and surrogate keys — is one of the few pieces of vendor documentation worth reading end to end: that is where the defaults live that this article has taught you not to accept blindly.