1. What is HTTP Caching & Why is it needed?

HTTP Caching is a mechanism where the browser (and intermediary servers) stores a local copy of server responses.

- **The Need:** It reduces **latency** (faster loads), decreases **bandwidth** consumption, and lowers the **server load**.
- **The Performance Impact:** It is the difference between a **200 OK** (fetching from network, ~200ms-2s) and a **200 OK (from cache)** or **304 Not Modified** (~10ms-50ms).

2. Setting and Accessing: The Roles

- **Who Sets it?** The **Server** defines the caching policy via HTTP Response Headers (e.g., Cache-Control).
- **Who Accesses it?** The **Browser** (automatically). JavaScript/Frontend code **cannot** directly read or write to the HTTP Cache (unlike the Cache API or LocalStorage). It is managed by the browser's network stack.

3. Storage & Memory Location

- **Where it lives:**
  1. **Memory Cache (RAM):** Very fast, volatile. Stores resources used during the current session. Cleared when the tab/browser closes.
  2. **Disk Cache (Hard Drive):** Slower but persistent. Stores resources intended for long-term use.
- **Capacity:** Browsers typically allocate a percentage of available disk space (often hundreds of MBs to GBs). It is much larger than LocalStorage (5MB).

4. Eviction: How is it cleared?

- **Automatic:** The browser uses an **LRU (Least Recently Used)** algorithm. When the cache is full, the oldest/least accessed files are deleted.
- **Manual (Client):** User clears "Browsing Data."
- **Programmatic (Server):** The server cannot "reach into" a browser and delete a file. It can only "invalidate" it by changing the file's URL (Versioning/Hashing) or sending a Clear-Site-Data: "cache" header.

5. Various Ways to Set HTTP Cache (The Core Interview Topic)

There are two main categories: **Strong Caching** and **Validation (Consultative) Caching**.

A. Strong Caching (No Network Request)

The browser doesn't even talk to the server. It just grabs the file from the disk.

1. **Cache-Control (Modern/Recommended):**
   - max-age=31536000: Cache the file for 1 year (in seconds).
   - immutable: Tells the browser the file will *never* change (used with hashed filenames like main.a7b2.js).
   - no-store: Do not cache at all (used for sensitive data).
   - no-cache: This is a "gotcha." It **does not** mean "don't cache." It means "Cache it, but check with the server before using it" (Validation).
2. **Expires (Legacy):**
   - Sets a specific date: Expires: Wed, 21 Oct 2025 07:28:00 GMT.
   - **Problem:** If the user’s system clock is wrong, this fails. Cache-Control overrides this.

B. Validation Caching (Network Request + 304 Response)

The browser asks: "I have this file from yesterday, is it still good?" If yes, the server sends a **304 Not Modified** (no body, very fast).

1. **ETag (Entity Tag):**
   - A unique fingerprint/hash of the file content.
   - **Flow:** Server sends ETag: "v1.2". Next time, Browser sends If-None-Match: "v1.2".
2. **Last-Modified:**
   - A timestamp.
   - **Flow:** Server sends Last-Modified: [Date]. Next time, Browser sends If-Modified-Since: [Date].

---

6. Critical Concepts You Must Mention

Cache-Busting (Fingerprinting)

Since you might set a file to cache for 1 year (max-age=31536000), how do you update it when you fix a bug?

- **The Strategy:** You change the filename. Instead of style.css, you use style.v2.css or style.8h2f.css.
- **Why?** The browser sees a new URL and treats it as a completely new resource, bypassing the old cache.

Private vs. Public

- Cache-Control: public: Can be cached by the browser **and** CDN/Proxies (good for static assets).
- Cache-Control: private: Only the **user's browser** can cache it (good for user-specific HTML).

Vary Header

- Tells the cache that the response depends on a specific request header (usually Vary: Accept-Encoding to distinguish between Gzip and Brotli versions of a file).

---

7. How it helps Performance: The "Metric" View

- **TTFB (Time to First Byte):** Reduced significantly because cached hits don't wait for server processing.
- **LCP (Largest Contentful Paint):** Images (like hero banners) that are cached render almost instantly on the second visit.
- **Bandwidth Cost:** Drastically reduces data costs for users on mobile plans and reduces your cloud egress costs.

---

8. Most Asked Interview Q&A

Q: "What is the difference between no-cache and no-store?"
A: no-store is absolute—the browser must not keep any copy. no-cache allows the browser to keep a copy but forces it to validate with the server (via ETag) before showing it to the user.

Q: "Explain the 'Stale-While-Revalidate' (SWR) header."
A: This is a modern performance strategy. Cache-Control: max-age=60, stale-while-revalidate=3600.
The browser shows the cached (stale) version immediately if it's less than an hour old, but simultaneously triggers a background fetch to update the cache for the next visit.

Q: "If a response has both Cache-Control: max-age=10 and an ETag, what happens after 15 seconds?"
A: The max-age has expired. The browser will make a request to the server, sending the If-None-Match (ETag) header. If the file hasn't changed, the server returns 304, and the browser reuses the local file.

---

1. What exactly can be cached?

The HTTP Cache is designed for **Idempotent** requests (requests that don't change the state of the server).

- **Methods:** Primarily **GET** requests. While some browsers theoretically allow caching of HEAD or POST (if configured with explicit freshness headers), in 99% of production systems, **only GET requests are cached**.
- **Content Types:**
  - **Static Assets:** JS files, CSS files, Images (PNG, WebP, SVG), Fonts (WOFF2).
  - **Documents:** HTML files.
  - **Data:** API responses (JSON/XML) — though this is usually managed with shorter TTLs (Time to Live).
- **What NOT to cache:** POST, PUT, DELETE requests, and any URL containing highly sensitive PII (Personally Identifiable Information) unless using Cache-Control: private.

---

2. The Flow: Strong Caching (Cache-Control: max-age)

In this scenario, the browser is the "Decision Maker."

Scenario A: Cache Hit

- The User requests app.js.
- The Browser checks its internal Cache Map.
- **Check:** Is current_time < (received_time + max_age)?
- **Result: Yes.**
- **Action:** The Browser retrieves the file from **Disk/Memory**. No network request is ever sent.
- **DevTools:** You see Status: 200 OK (from disk cache).

Scenario B: Cache Miss (Expired or Missing)

- The User requests app.js.
- The Browser checks the cache.
- **Result:** File is missing **or** max_age has expired.
- **Action:** The Browser sends a full HTTP GET request to the Server.
- **Server Response:** The Server sends the file (200 OK) + a new Cache-Control: max-age=... header.
- **Update:** Browser saves the new version to the disk.

---

3. The Flow: Validation Caching (ETag / If-None-Match)

In this scenario, the Server is the "Decision Maker."

Scenario A: Cache Hit (Revalidation)

- The User requests style.css.
- The Browser sees it has a version but it's "stale" (expired max-age or no-cache was used). It sees an ETag: "xyz123" in its records.
- **Action:** The Browser sends a request with a header: If-None-Match: "xyz123".
- **Server Check:** The Server checks the current file hash. It still matches "xyz123".
- **Result:** The Server sends a **304 Not Modified** (no body/payload).
- **Action:** The Browser updates the "freshness" of its local copy and displays it. This is extremely fast because the response body is empty.

Scenario B: Cache Miss

3. The User requests style.css with If-None-Match: "xyz123".
4. **Server Check:** The file has changed! The new hash is "abc789".
5. **Result:** The Server sends a **200 OK** + the **entire new file** + the new ETag: "abc789".
6. **Action:** The Browser replaces the old file in the cache with the new one.

---

4. Summary Table for the Interviewer

Feature	Strong Cache (max-age)	Validation Cache (ETag)
Network Request?	**No** (if fresh)	**Yes** (always, to check)
Server Load	Zero	Minimal (Header processing only)
Use Case	Hashed assets (main.8h2f.js)	Mutable files (index.html)
Response Code	200 OK (from cache)	304 Not Modified

---

5. Pro-Tip: The "Hashed Asset" Pattern

In modern Frontend System Design, we combine these.

- ➢ **HTML (index.html):** Use Cache-Control: no-cache + ETag. We always check the server to see if a new version of the app exists.
- ➢ **Assets (bundle.js, logo.png):** Use **Content Hashing** (e.g., main.a1b2.js). Set Cache-Control: max-age=31536000, immutable.
- ➢ **The Benefit:** If the JS changes, the HTML will point to a new filename. The old file stays in cache but is ignored; the new file is fetched and cached forever.

This is a great clarifying question. In an interview, knowing the **default behavior** shows you understand the "magic" that happens under the browser's hood.

1. Which types of responses get cached?

Technically, the browser looks at the **HTTP Method** and the **Status Code**.

- **HTTP Methods:** Only **GET** is cached by default. POST, PUT, PATCH, and DELETE are considered "unsafe" or "non-idempotent" (they change data on the server), so the browser will **never** cache them by default.
- **Status Codes:** The browser typically only caches "Successful" responses.
  - **Commonly Cached:** 200 (OK), 203 (Non-Authoritative Info), 204 (No Content), 300 (Multiple Choices), 301 (Moved Permanently), 410 (Gone).
  - **Rarely/Never Cached:** 404 (Not Found), 500 (Internal Server Error), 503 (Service Unavailable).

---

2. Heuristic Caching: What happens if no headers are specified?

If the server sends a GET response but **does not** include Cache-Control, Expires, ETag, or Last-Modified, the browser doesn't just give up. It performs **Heuristic Caching**.

The browser thinks: *"The server didn't tell me what to do, but this looks like a static file. I'll guess how long it's safe to keep."*

The "10% Rule" (Standard Heuristic)

Most modern browsers (Chrome, Firefox) use a simple formula if they see a Last-Modified header but no Cache-Control:

Cache Duration = (Date of Request - Last-Modified Date) * 0.10

- **Example:** If you fetch a file today (Jan 6, 2026) and the server says it was last modified 100 days ago, the browser will cache it for **10 days** automatically.
- **The Danger:** If the server provides **no headers at all** (not even Last-Modified), the

browser behavior is "implementation-defined." Usually, it won't cache it for long, or it might treat it as a session-only cache.

---

3. The Flow: Cache Hit vs. Cache Miss

To make this crystal clear for your interview, here is the step-by-step logic for both **Strong** and **Validation** caching.

Scenario 1: Strong Caching (The "Speed" Path)

This uses Cache-Control: max-age=....

- **Cache HIT:**
  - Browser: "I need hero.jpg."
  - Browser checks cache: "I have it. It was fetched 1 minute ago. Its max-age is 1 hour."
  - **Result:** Browser takes it from disk. **Network activity: 0.**
- **Cache MISS:**
  - Browser: "I need hero.jpg."
  - Browser checks cache: "It's expired (stale)" or "I don't have it."
  - **Result:** Browser sends a full request to the Server. Server returns 200 OK + the file.

Scenario 2: Validation Caching (The "Integrity" Path)

This uses ETag or Last-Modified.

- ➢ **Cache HIT (Revalidation):**
  - Browser: "I need index.html. My version is stale, but it has ETag: 'v1'."
  - Browser sends: GET /index.html with header If-None-Match: 'v1'.
  - Server: Checks its file. It’s still 'v1'.
  - **Result:** Server sends **304 Not Modified** (no body). Browser uses its local copy.
- ➢ **Cache MISS:**
  - Browser sends: GET /index.html with If-None-Match: 'v1'.
  - Server: "The file changed. The new version is 'v2'."
  - **Result:** Server sends **200 OK** + the entire new index.html file + ETag: 'v2'.

---

4. Summary of "No Header" Consequences

In an FE System Design interview, if you are asked about a site with no cache headers, your answer should be:

2. **Inconsistency:** Different browsers will guess different durations (Heuristic Caching).
3. **Performance Risk:** A browser might cache a file for too long, and the user won't see updates (Stale Data).
4. **Server Stress:** Or, the browser might never cache it, causing the server to be hit with requests for every single image and icon on every page load.