Engineer

Caching

⏭ keeping an answer so the work is not repeated

What it is

Caching is the practice of saving the result of some piece of work, such as an HTTP response, a query result or a rendered page, and handing that saved result to later requests under rules that say how long it can be trusted and what retires it. The verb is to cache, and a result kept this way is cached; caching the return values of one function inside a program is usually called memoisation.

The store that holds the copies is the cache. Caching is the policy around it, and the policy has three questions. How long is a copy fresh, meaning fit to serve without asking anyone? Once it is stale, how is it checked? And when the original changes, what removes or replaces the copy?

HTTP answers all three in headers. RFC 9111 calls a stored response fresh while its age is below its freshness lifetime, which the origin sets with the `max-age` directive of `Cache-Control` or with an `Expires` date. Past that point the response is stale, and a cache normally has to validate it before reuse. It sends a conditional request carrying a validator it kept, usually the response's `ETag` in an `If-None-Match` header, and if nothing has changed the origin answers 304 Not Modified with no body. Revalidation still costs a round trip, but it saves the transfer and the work of building the body again.

```
HTTP/1.1 200 OK
Cache-Control: max-age=3600, stale-while-revalidate=60
ETag: "v42"
```

Inside an application the same questions take other shapes. In cache-aside, the code reads the cache first, and on a miss reads the database and fills the cache itself. In write-through, every write updates the cache together with the

database, so a read finds the current value. An expiry on each entry, its time to live or TTL, plays the part of `max-age`. Even a database can cache: a PostgreSQL materialized view stores a query's result as a table, and that result changes only when someone runs `REFRESH MATERIALIZED VIEW`.

Use the concept in a decision

Put "What is caching?" to work in a real decision. We can help relate the concept to the system you are building or buying. Finding the queries and the missing indexes behind a system that has got heavier every year, and fixing the ones that matter.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. 30 days of electricity data in 180 ms — 180 ms to aggregate a month of meter readings

30 days of electricity data in 180 ms Get in touch →

Why it matters

Every caching rule trades freshness for speed, and the right trade differs by kind of data. A logo can be a year old. A product price can be a minute old. A balance or a seat count should be confirmed with the source before anyone acts on it. Caching most often goes wrong when one rule is applied to all three.

Static files get the cleanest answer. When a build writes a hash of each file's content into its name, a changed file gets a new URL, so the old URL can be cached for a year. RFC 8246 adds the `immutable` directive, which tells the browser the response will stay the same while it is fresh, so even a reload can skip revalidating it. The HTML page that names those files has to stay short-lived or be revalidated on every use, or visitors keep loading the old names.

Two directives are easy to confuse. `no-cache` lets a cache store the response but forbids reusing it without first validating it with the origin; `no-store` forbids storing it at all. For a page that changes often, `no-cache` with an `ETag` gives current content and still saves the transfer when nothing has changed.

Invalidation is the hard half. RFC 9111 requires a cache to invalidate a URL when an unsafe request to it, such as `POST`, `PUT` or `DELETE`, succeeds, and the same section points out that this reaches only the caches that request passes through. A browser, a CDN and an application cache are three separate places, and removing a copy from one leaves the other two in place. So most systems rely on expiry as the backstop and treat explicit invalidation as a way to be current sooner.

Stale data can also be useful. RFC 5861 defines `stale-while-revalidate`, which lets a cache serve a stale response for a stated number of seconds while it fetches a fresh one in the background, and `stale-if-error`, which lets it keep serving a stale copy when the origin returns an error. Both turn a hard expiry into a soft one: the visitor gets an answer at once, and the update follows.

What it looks like in practice

Many of one platform's users are seafarers and field staff at sea, on phones and often offline, and for them caching made the web app installable and usable at sea. The service worker is configured through next-pwa, and Workbox runtime caching serves fonts, images, audio, video and CDN assets cache first: from the device when a copy is there, from the network only on a miss. Those are the files that stay the same from one request to the next, and HTTP cache-control headers go with them. Because service workers need HTTPS, install and offline behaviour were tested over HTTPS on real devices. The outcome is an app that installs, keeps working offline and loads repeat visits from cache rather than the network. A cache-first rule skips the freshness check entirely while an entry exists, which is why it suits files whose content is tied to their address.

At the other end of a system sits a PostgreSQL database made ten times faster that holds users, tenants, geographic data and daily electricity readings. Caching was one measure among several. Next to targeted indexes, GiST for spatial queries and composite ones for tenant and date filters, and time-based partitioning of the electricity table, some query results were kept as materialized views, and query caching and connection pooling were set up to cut overhead. Response times improved by 10x, critical operations ran in milliseconds and resource use fell by 40%. Those gains belong to the work as a whole, so the share that caching contributed is not measured on its own.

Where it goes wrong

After a deploy, some visitors still get the old version. The response carried no explicit freshness, so caches estimated one. RFC 9111 lets them use a fraction of the time since `Last-Modified` and names 10% as a typical setting, so a file last changed a month ago can be reused for about three days without asking. Send an explicit `Cache-Control` on every response, put a content hash in static file names, and keep the HTML that points at them short-lived.

A purge clears the CDN, and browsers keep the old copy. A purge removes the CDN's copy and nothing else; each browser that stored the response keeps it until its own lifetime runs out. Give public responses a short `max-age` for browsers and a longer `s-maxage`, which only shared caches apply, so the copy that can be purged is the one that lives longest. When a browser copy must change at once, change the URL.

A page that was briefly missing stays missing. RFC 9110 makes a 404 heuristically cacheable, so a cache may keep one that came with no freshness information for a lifetime it estimates itself. A link followed a moment before a page was published can keep returning 404 after the page exists. Give error responses an explicit short lifetime, and purge the URL when the page goes live.

An old value is written back after an update. In cache-aside, a reader misses and reads the row; before it stores the value, a writer updates the row and deletes the entry; then the reader stores what it read, and the old value stays until something removes it. Delete the entry after the transaction commits, and give every entry a TTL so that a raced or missed invalidation ends on its own. In Redis, a plain `SET` discards any earlier expiry, so set the TTL in the same command:

```
SET price:42 "19.90" EX 300
```

Further reading

- RFC 9111, Section 3: Storing Responses in Caches — when a response may be stored, followed by the freshness, revalidation and invalidation rules.
- RFC 9110, Section 13: Conditional Requests — `ETag`, `If-None-Match` and the 304 response that make revalidation cheap.
- RFC 5861: HTTP Cache-Control Extensions for Stale Content — the two directives that let a cache serve stale responses on purpose.
- RFC 8246: HTTP Immutable Responses — the `immutable` directive for files whose URL changes with their content.
- Workbox: caching strategies — cache first, network first and stale-while-revalidate in a service worker, with code.
- PostgreSQL: Materialized Views — keeping a query result inside the database and refreshing it on demand.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is caching?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/caching/