Browser caching strategies

You have a site that loads like a drunk sloth. The fix isn't a faster server or a CDN—not always. The real lever, the one most devs ignore until their Lighthouse score screams, is how you tell the browser to stop re-downloading the same junk. **Browser caching strategies** are the set of rules you define so the client holds onto assets—images, CSS, JavaScript—and doesn't bug your server for them on every single page view. This isn't theory. This is the difference between a 2-second load and a 0.3-second load for returning visitors.

Most people screw this up by either caching nothing (default behavior) or caching everything forever (breaking their own site when they push an update). The middle ground is where you want to live.

The mental model: your site is a warehouse, the browser is a hoarder

Think of your server as a warehouse. Every time a browser visits, it walks to the warehouse, grabs a box, and carries it back. That walk takes time. If the browser already has a box from last week, and you tell it the box hasn't changed, it just uses the old one. No walk. That's caching.

The problem is that browsers, left to their own devices, are conservative hoarders. They'll keep a CSS file for days unless you explicitly say "throw this away after an hour." And if you never say anything, they'll still cache some things based on heuristics—but inconsistently across browsers and versions. You need to be explicit.

Cache-Control: the one header that does 90% of the work

Forget Expires headers. Forget Pragma. The HTTP/1.1 spec gave us Cache-Control, and it's the only tool you need for most static assets. The header is a string you send with your response. Here's the brutal truth: you only need a handful of directives.

- **public**: Any cache (browser, CDN, proxy) can store this. Use for static assets like images, fonts, and CSS.
- **private**: Only the browser can cache this. Use for HTML pages with user-specific content.
- **no-cache**: Misleading name. It means "check with the server before using the cached copy." It doesn't mean "don't cache."
- **no-store**: This actually means "don't cache anything." Use for sensitive data like bank balances.

- **max-age=[seconds]**: How long the browser can keep the asset without asking the server.

A typical setup for a production site looks like this: for your JavaScript bundle, set Cache-Control: public, max-age=31536000, immutable. That's one year. The immutable flag tells the browser "don't even bother asking if this changed—it won't." For your HTML pages, set Cache-Control: no-cache so the browser always revalidates. For API responses with user data, private, no-cache.

ETags and Last-Modified: the revalidation dance

You set max-age=3600. After an hour, the browser sends a request with If-None-Match (the ETag) or If-Modified-Since (the Last-Modified date). Your server checks if the file changed. If it didn't, you send back a 304 Not Modified response with an empty body. That's a tiny response—no payload. The browser uses its cached copy.

ETags are fingerprints—usually a hash of the file content. They're more precise than timestamps. If you're using a framework like Express or Django, ETags are often generated automatically. Don't disable them. They're your safety net for revalidation.

> Rule of thumb: use ETags for revalidation, but don't rely on them as your primary caching strategy. They're for the case when the cache expires, not for the case when the cache is still fresh.

Service workers: the nuclear option for offline and speed

Cache-Control headers are passive. The browser follows your rules. Service workers are active. They intercept every network request from your site and decide what to do: serve from cache, fetch from network, or a mix. This is where you get aggressive.

A common pattern is "stale-while-revalidate." You serve the cached version immediately (milliseconds), then fetch the latest version in the background and update the cache for next time. The user never waits for the network. This is what makes Progressive Web Apps feel native.

But service workers are complex. They run in a separate thread. They can break your site if you write bad fetch logic. And they require HTTPS. If you're not ready for that complexity, stick to Cache-Control headers. They cover 95% of use cases without the headache.

The three mistakes that kill your caching strategy

First mistake: caching the HTML page with a long max-age. You push a new blog post.

The browser doesn't know. It serves the old HTML from cache. Your readers don't see the new post for a day. Fix: always set no-cache on HTML, or use a very short max-age like 60 seconds.

Second mistake: not versioning your static assets. You have style.css with max-age=31536000. You change a color. The browser still has the old style.css in cache. It doesn't fetch the new one. Fix: add a hash to the filename—style.a1b2c3.css—or use a query parameter like style.css?v=2. When the file changes, the URL changes, so the browser fetches the new one.

Third mistake: caching everything with public when some assets are user-specific. You cache a profile image with public, max-age=86400. A CDN serves that image to another user. Now they see the wrong profile picture. Fix: use private for anything tied to a specific user session.

Real-world scenario: an e-commerce product page

You run a store selling custom furniture. Your product page has a hero image (2MB), a CSS file (50KB), a JavaScript bundle (200KB), and the HTML (15KB).

The hero image: Cache-Control: public, max-age=31536000, immutable. It never changes. The CSS and JS: same, but you version them with hashes in the build step. The HTML: Cache-Control: no-cache. Every time a user visits a product page, the browser checks with the server. The server sends a 304 if the product details haven't changed. The HTML loads in under 100ms because the heavy assets are cached for a year.

The trade-off: you need a build pipeline that generates hashed filenames. Tools like Webpack or Vite do this automatically. If you're on a shared hosting plan with no build step, you can't do this easily. In that case, set max-age=86400 on your CSS and JS and accept that some users will see stale versions for a day.

Myth vs reality about browser caching

Myth 1: "The browser caches everything by default."
Reality: Browsers cache based on heuristics, but they're conservative. Without explicit headers, many resources are re-fetched on every page load. You must set headers.

Myth 2: "Setting a long cache time means users never see updates."
Reality: Only if you don't version your assets. With versioned filenames, a long cache time is safe. The old URL is cached, but the new URL is fetched fresh.

Myth 3: "Service workers replace HTTP caching."
Reality: Service workers work on top of HTTP caching. They can override it, but

they're not a replacement. You still need Cache-Control headers for non-service-worker scenarios and for the initial fetch.

How to audit your current caching setup

Open Chrome DevTools, go to the Network tab, and reload your site. Click on any resource. Look at the Response Headers. Do you see Cache-Control? What's the max-age? If you see max-age=0 or no Cache-Control at all, you're not caching. If you see max-age=31536000 on your HTML, you're caching too aggressively.

Use the Lighthouse audit in Chrome. It will flag "Serve static assets with an efficient cache policy" if your cache times are too short. It recommends at least 180 days for static assets. That's a good baseline.

For a deeper check, use WebPageTest and look at the "Cache" section. It shows you exactly how long each resource is cached and whether it's being revalidated correctly.

Decision tree: which caching strategy fits your situation?

If you have a static site hosted on Netlify or Vercel: they handle caching for you. Set max-age=31536000 on all build artifacts and no-cache on HTML. Done.

If you have a dynamic site with a backend (Node, PHP, Python): you need to configure your server or framework. In Express, use res.set('Cache-Control', 'public, max-age=86400'). In Nginx, add add_header Cache-Control "public, max-age=86400"; in your server block. In Apache, use Header set Cache-Control "public, max-age=86400" in your .htaccess.

If you have a single-page application (React, Vue, Angular): build your app with hashed filenames. Serve the index.html with no-cache. Serve everything else with max-age=31536000, immutable. Your users get instant loads after the first visit.

If you have a WordPress site: use a caching plugin like WP Rocket or W3 Total Cache. They set the headers for you. But check that they're not caching the admin pages.

One final thing: don't overthink it

Browser caching is not complicated. It's a few headers. It's a versioning scheme. It's a decision about how long you're willing to serve stale content. Start with max-age=86400 on your static assets and no-cache on your HTML. Measure the difference. Then push the cache time to a year for anything that has a versioned URL. That's it. That's the strategy. Everything else is optimization around the edges.

Technical Verification Node

fast indexing tool

Report ID: 8A23B41A | Signature: 50db33884c6cc517ec406b70c9ca670a