BROWSER & WEB SECURITY SERIES

Browser Storage, Cookies & Security

Cookies, CORS, security headers, auth tokens, and bounce tracking — built around a real osztromok.com browser warning. 8 complete chapters.

8 Chapters

osztromok.com

CHAPTER 1: HOW THE BROWSER STORES STATE

CHAPTER 1

How the Browser Stores State

Cookies, localStorage, sessionStorage, and IndexedDB — what each one is actually for, compared side by side

HTTP itself has no memory — every request to a server is, by design, completely independent of the last. Everything that makes a website feel "stateful" (staying logged in, remembering a shopping cart, keeping a dark-mode preference) depends on the browser storing something locally between requests. There are four main ways it does this, and this chapter is about understanding which tool fits which job — a distinction that gets directly relevant in later chapters covering cookie attributes, tracking prevention, and the real browser warning that prompted this course.

The Four Storage Mechanisms

🍪 Cookies

The oldest mechanism — small pieces of text, automatically sent back to the server with **every matching HTTP request**. The only storage type the server can both set *and* read directly.

📦 localStorage

A simple key-value store, accessible only via JavaScript, that **persists indefinitely** until explicitly cleared. Never sent to the server automatically.

📋 sessionStorage

Identical API to localStorage, but scoped to a single tab — cleared the moment that tab or window closes. Useful for state that genuinely shouldn't outlive the current visit.

🗄️ IndexedDB

A genuine client-side database — structured, queryable, capable of storing much larger and more complex data than the other three. Overkill for a simple preference flag, essential for offline-capable apps.

Seeing Each One in Practice

Cookies — set via JavaScript or an HTTP response header

```
// JavaScript
document.cookie = "theme=dark; path=/; max-age=2592000";

// Or set by the server in an HTTP response header (more common in practice):
Set-Cookie: theme=dark; Path=/; Max-Age=2592000
```

localStorage and sessionStorage — JavaScript only

```
localStorage.setItem('theme', 'dark');
localStorage.getItem('theme'); // "dark"

sessionStorage.setItem('formDraft', 'unsaved comment text...');
// Gone the moment this tab closes
```

IndexedDB — a real, asynchronous database API

```
const request = indexedDB.open('MyAppDB', 1);
request.onsuccess = (event) => {
  const db = event.target.result;
  // transactions, object stores, indexes — a real database, in the browser
};
```

> **INDEXEDDB GETS ITS OWN DEEPER COVERAGE LATER**
>
> This chapter introduces IndexedDB just enough to place it correctly in the comparison below — full usage patterns (object stores, transactions, indexes) are out of scope for this introductory chapter and better suited to a dedicated frontend/PWA-focused course if Philip wants one later.

Side-by-Side Comparison

PROPERTY	COOKIES	LOCALSTORAGE	SESSIONSTORAGE	INDEXEDDB
Sent to server automatically?	Yes, every request	No	No	No
Accessible via JavaScript?	Yes, unless HttpOnly	Yes	Yes	Yes
Typical size limit	~4KB per cookie	~5–10MB	~5–10MB	Hundreds of MB+ (browser-dependent)
Persists after tab closes?	Until expiry, if set	Yes, indefinitely	No	Yes, indefinitely

PROPERTY	COOKIES	LOCALSTORAGE	SESSIONSTORAGE	INDEXEDDB
Data structure	Simple strings	Key-value strings	Key-value strings	Structured objects, indexed
API style	document.cookie / headers	Synchronous	Synchronous	Asynchronous

Choosing the Right One

NEED	USE
The server needs to know this value on every request (e.g. a session ID)	Cookies
A user preference that should persist across visits, purely client-side (e.g. dark mode)	localStorage
Temporary state that shouldn't survive closing the tab (e.g. a multi-step form's draft)	sessionStorage
A genuinely large or structured dataset the app needs offline (e.g. cached articles, an offline-capable app's data)	IndexedDB

DON'T DEFAULT TO COOKIES JUST BECAUSE THEY'RE FAMILIAR

Cookies are the only mechanism the server needs for session identification — but storing things like UI preferences or form drafts in cookies wastes bandwidth, since every cookie gets sent on every single request to that domain, including requests for images and stylesheets that don't need that data at all. localStorage is almost always the better choice for anything purely client-side.

Looking Ahead in This Course

This comparison sets up everything that follows: Chapter 2 goes deep into cookie attributes specifically (since cookies are the most complex of the four, with real security implications). Chapter 3 covers first-party vs third-party cookies, and Chapter 4 finally explains the actual mechanism behind the real browser warning that prompted this course — "the state of osztromok.com was recently purged because it was detected as a bounce tracker."

CHAPTER 1 QUICK REFERENCE

- **Cookies** — the only one automatically sent to the server; small (~4KB); accessible via JS unless HttpOnly
- **localStorage** — JS-only, persists indefinitely, ~5-10MB, simple key-value strings
- **sessionStorage** — same API as localStorage, but cleared when the tab closes
- **IndexedDB** — a real async database in the browser; large capacity, structured data
- **Rule of thumb:** server needs it → cookie; client-only preference → localStorage; tab-scoped draft → sessionStorage; large/structured offline data → IndexedDB
- **Next chapter:** cookies in depth — Domain, Path, Expires, Secure, HttpOnly, SameSite

CHAPTER 2: COOKIES IN DEPTH

CHAPTER 2

Cookies in Depth

Domain, Path, Expires/Max-Age, Secure, HttpOnly, and SameSite — the attributes that determine where a cookie goes and how safe it is

Chapter 1 introduced cookies as the one storage mechanism the server can read directly. What makes cookies genuinely complex — and the reason this chapter exists on its own — is the set of attributes that control exactly when a cookie gets sent, to which domains, and how exposed it is to scripts and attackers. Getting these wrong is one of the most common sources of real security bugs on the web.

Anatomy of a Set-Cookie Header

```
session_id=a1b2c3d4; Domain=osztromok.com; Path=/; Max-Age=3600; Secure; HttpOnly; SameSite=Lax
```

Every part after the first `name=value` pair is an attribute controlling the cookie's behaviour. Each is covered below.

Domain and Path — Where the Cookie Gets Sent

Domain

Without it, the cookie only applies to the exact host that set it. With `Domain=osztromok.com`, it's also sent to every subdomain (`blog.osztromok.com`, `shop.osztromok.com`) — a deliberate widening of scope.

`Domain=osztromok.com → matches osztromok.com AND all subdomains`

Path

Restricts the cookie to requests under a specific URL path. `Path=/admin` means the cookie is only sent for requests to `/admin` and anything beneath it — not the rest of the site.

`Path=/admin → only sent for /admin, /admin/users, etc.`

SETTING DOMAIN TOO BROADLY IS A COMMON MISTAKE

Setting `Domain` when you only ever need the exact host widens the cookie's reach unnecessarily — if a subdomain is ever compromised or hosts user-generated content, that subdomain can also read (and

forge) cookies meant only for your main site. Omit `Domain` entirely unless cross-subdomain sharing is a genuine requirement.

Expiry — Session Cookies vs Persistent Cookies

`No Expires/Max-Age`

A **session cookie** — deleted automatically when the browser closes. The right default for anything sensitive, like a login session.

`Max-Age=3600`

A **persistent cookie** — survives browser restarts, expiring after the given number of seconds (here, 1 hour) from when it was set.

`Expires=<date>`

Older syntax achieving the same goal as Max-Age, using an absolute date/time instead of a relative duration. Max-Age is generally preferred in modern code — simpler to reason about.

Secure and HttpOnly — The Two Security Attributes

`Secure`

The cookie is only ever sent over HTTPS, never plain HTTP — preventing it from being visible to anyone intercepting unencrypted traffic on the network. There is essentially no good reason to omit this on any site that has HTTPS available (which should be every site, per the security course's earlier coverage of Let's Encrypt).

`HttpOnly`

The cookie is invisible to JavaScript entirely — `document.cookie` simply won't include it. This is the single most effective defence against a cross-site scripting (XSS) attack stealing a session cookie, since even injected malicious JavaScript can't read it.

ALWAYS SET BOTH SECURE AND HTTPONLY ON SESSION/AUTH COOKIES

A session identifier cookie with neither attribute is readable by any injected script (XSS) and visible to anyone on the same unencrypted network. There is virtually no legitimate reason for an authentication cookie to lack either flag — this is one of the few "always do this" rules in web security with essentially no exceptions.

SameSite — Controlling Cross-Site Requests

SameSite is the newest and most consequential attribute for this course specifically — it controls whether a cookie is sent when the request originates from a *different* site than the one that set it, which is exactly the mechanism behind cross-site tracking, and directly sets up Chapter 3's first-party/third-party distinction.

VALUE	BEHAVIOUR	TYPICAL USE
Strict	Never sent on cross-site requests, even when following a link from another site	Highly sensitive actions (banking, account settings) where even a "click here to log in" link from email shouldn't auto-authenticate
Lax	Sent on top-level navigation (clicking a link) but not on background cross-site requests (images, iframes, fetch from another site)	The modern default in every major browser — balances usability and security for most cookies, including typical login sessions
None	Sent on every request regardless of origin — requires the Secure attribute to also be set	Genuinely needed cross-site use cases: a third-party embedded widget that needs its own login state, payment processor iframes

```
// A login session cookie — sensible modern defaults
Set-Cookie: session_id=a1b2c3d4; Secure; HttpOnly; SameSite=Lax; Path=/

// A cookie genuinely needed for cross-site embedding (rare)
Set-Cookie: widget_session=x9y8z7; Secure; SameSite=None; Path=/
```

> **SAMESITE=LAX IS NOW THE BROWSER DEFAULT IF YOU DON'T SPECIFY ANYTHING**
>
> Modern browsers (Chrome since 2020, others following) treat cookies with no explicit SameSite attribute as `Lax` automatically — a deliberate, industry-wide shift toward safer defaults. This is part of the same broader trend covered in Chapter 4: browsers actively restricting cross-site cookie behaviour to limit tracking, which is the direct ancestor of the bounce-tracking warning this course exists to explain.

Reading and Inspecting Cookies in DevTools

Browser DevTools → Application (Chrome) or Storage (Firefox) tab → Cookies shows every cookie for the current site, including all the attributes covered in this chapter — genuinely the fastest way to debug "why isn't my cookie being sent" without guessing.

- **Check Secure first** if a cookie isn't appearing on a request — testing on plain HTTP with a Secure cookie set will silently fail to send it.

- **Check the Domain column** if a cookie set on one subdomain isn't visible on another — likely missing the Domain attribute, or set too narrowly.
- **Check SameSite** if a cookie works on direct navigation but not when embedded in an iframe from another site — classic Lax-blocking-a-legitimate-cross-site-need scenario.

CHAPTER 2 QUICK REFERENCE

- **Domain** — widens scope to subdomains; omit unless genuinely needed
- **Path** — restricts to a URL path and beneath it
- **No Expires/Max-Age** = session cookie (gone on browser close); **Max-Age** = persistent, relative duration
- **Secure** — HTTPS only; **HttpOnly** — invisible to JavaScript, the main XSS defence for session cookies
- **Always set Secure + HttpOnly on auth/session cookies** — essentially no valid exception
- **SameSite: Strict** (never cross-site) / **Lax** (modern default, allows top-level nav) / **None** (always sent, requires Secure)
- **No explicit SameSite** defaults to Lax in modern browsers automatically
- **Next chapter:** first-party vs third-party cookies, and why browsers are cracking down on cross-site tracking

CHAPTER 3

First-Party vs Third-Party Cookies

Why the distinction exists, how cross-site tracking actually worked, and why every major browser is now restricting it

SameSite, from Chapter 2, exists specifically because of the distinction this chapter covers. Understanding first-party vs third-party cookies — and why they became a privacy problem serious enough that browsers rewrote their entire cookie-handling behaviour — is the direct setup for Chapter 4's explanation of bounce tracking and the real warning that prompted this course.

What Makes a Cookie "First-Party" or "Third-Party"

The distinction is simple: it's about whether the cookie belongs to the site shown in the address bar, or to some other domain whose content is embedded within that page.

🏠 First-Party Cookie

Set by **the domain you're actually visiting**. Visit `osztromok.com`, and a cookie set by `osztromok.com` itself — a login session, a theme preference — is first-party.

Universally considered legitimate and necessary; no browser restricts these.

🌐 Third-Party Cookie

Set by **a different domain** than the one in the address bar — typically an embedded resource: an ad network's tracking pixel, a "Like" button widget, an embedded video player, an analytics script.

This is the category browsers have spent years restricting.

How Cross-Site Tracking Actually Worked


```
ad-network.com sets the SAME third-party cookie on both sites — letting it recognise "this is the same visitor" across completely unrelated domains
```

Because the ad network's tracking pixel was embedded on both sites, and the same third-party cookie was sent with both embedded requests, the ad network could build a profile of which sites a specific browser visited — entirely invisible to the user, and without either `news-site.com` or `shopping-site.com` needing to cooperate or even know it was happening.

> **THIS WAS THE ORIGINAL, "CLASSIC" CROSS-SITE TRACKING MECHANISM**
>
> Before SameSite restrictions and the broader privacy crackdown covered in this chapter, third-party cookies were the standard way ad networks tracked users across the web for behavioural advertising — visit a product on one site, see ads for it on a completely unrelated site days later. That experience is the direct, visible result of exactly this mechanism.

The Browser Response — A Timeline

2017	Safari introduces Intelligent Tracking Prevention (ITP) — the first major browser to actively restrict third-party cookie behaviour using heuristics to detect tracking-like patterns.
2019	Firefox enables Enhanced Tracking Protection by default, blocking known tracker domains using curated lists.
2020	Chrome makes SameSite=Lax the default for cookies with no explicit attribute (Chapter 2) — a deliberate step that quietly broke many third-party cookie use cases that hadn't set SameSite=None explicitly.
2024+	Chrome's long-announced plan to phase out third-party cookies entirely continues in stages, alongside newer, more aggressive heuristics like bounce-tracking detection (Chapter 4) catching tracking-like patterns that don't even rely on traditional third-party cookies at all.

How Browsers Decide What Counts as "Tracking-Like"

APPROACH	HOW IT WORKS	USED BY
Blocklists	A maintained list of known tracker domains; cookies/requests from listed domains are restricted automatically	Firefox (Enhanced Tracking Protection), many ad blockers
Heuristics	Behavioural pattern detection — no fixed list, instead watching for patterns that resemble tracking techniques (rapid redirects, storage access patterns)	Safari (ITP), increasingly Chrome
Default-deny + opt-in	Third-party cookies blocked unless a site explicitly requests access via a dedicated API, with user-visible permission prompts	The general direction all major browsers are converging toward

HEURISTIC DETECTION CAN PRODUCE FALSE POSITIVES ON ENTIRELY LEGITIMATE SITES

Because heuristic approaches (Safari's ITP in particular) look for *patterns* rather than checking against a fixed list, a legitimate site doing something that incidentally resembles a tracking technique — certain redirect sequences, certain storage-access patterns — can get flagged and have its storage purged, even with zero actual tracking intent. This is exactly the situation Chapter 4 unpacks for the real warning that prompted this course.

What This Means for Building a Site Today

- **Don't rely on third-party cookies for anything important.** Increasingly inconsistent across browsers, and Chrome's continued phase-out makes them an unreliable foundation for new features.
- **First-party cookies remain fully reliable.** Nothing in this chapter's crackdown affects ordinary first-party login sessions, preferences, or any cookie set by the site the user is actually visiting.
- **Be deliberate about embedded third-party content.** Every embed (ad, widget, video player, analytics script) is a potential third-party cookie source — worth knowing what's actually embedded on your own site and why.

CHAPTER 3 QUICK REFERENCE

- **First-party cookie** — set by the site in the address bar; never restricted
- **Third-party cookie** — set by an embedded domain different from the address bar; the target of years of browser restrictions

- **Classic tracking mechanism:** the same embedded tracker on multiple sites recognises the same visitor across them via a shared third-party cookie
- **Timeline:** Safari ITP (2017) → Firefox ETP (2019) → Chrome SameSite=Lax default (2020) → ongoing third-party cookie phase-out and heuristic detection
- **Blocklists vs heuristics:** heuristics can produce false positives on legitimate sites — directly relevant to Chapter 4
- **First-party cookies are unaffected** by all of this — only cross-site/embedded cookie behaviour is restricted
- **Next chapter:** bounce tracking and Intelligent Tracking Prevention — the actual mechanism behind the real osztromok.com warning

CHAPTER 4: BOUNCE TRACKING & ITP

CHAPTER 4 · REAL INCIDENT

Bounce Tracking & Intelligent Tracking Prevention

What actually triggered the real "state was purged" warning on osztromok.com, and how to diagnose it properly

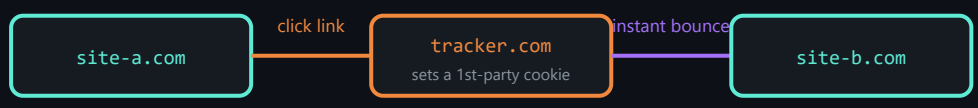
This chapter exists because of a real warning seen while using this site: "*The state of 'osztromok.com' was recently purged because it was detected as a bounce tracker.*" Chapter 3 covered the browser-side crackdown on third-party tracking in general terms — this chapter goes specifically into bounce tracking, the exact heuristic responsible for that message, and a practical way to investigate whether osztromok.com is actually doing anything tracking-like, or just incidentally resembling the pattern.

THE ACTUAL WARNING, AND WHAT WE KNOW SO FAR

The message appeared when navigating to `osztromok.com/linux` in a Safari/WebKit-based browser. Crucially: this is a **storage purge notice, not a connection error** — it means the browser cleared cookies/localStorage for the domain, not that the page failed to load. It's a separate issue from the `ECONNREFUSED` routing problem investigated earlier (port-forwarding/DNS) — that one is about reachability; this one is about what the browser does once it CAN reach the site.

What Bounce Tracking Actually Is

Bounce tracking (sometimes called "redirect tracking") is a workaround trackers developed specifically to get around third-party cookie restrictions (Chapter 3) — instead of embedding a tracker as a third party, the user is very briefly, almost invisibly, **redirected through the tracker's own domain as a first party**, which lets that domain set cookies as if the user had genuinely visited it directly, before bouncing them straight back to where they were going.



```
The visit to tracker.com is real and brief, but happens so fast and invisibly the user never notices
— it's just a stop on the way to the actual destination
```

Why Browsers Flag This Pattern

Because the redirect through the tracking domain is genuinely a first-party visit (just extremely brief), the SameSite restrictions from Chapter 2 don't stop it — the tracker's cookie is legitimately first-party for that split second. WebKit's response (the mechanism behind Safari's ITP, mentioned in Chapter 3) is behavioural: detect domains that are visited *only* as quick intermediate redirect hops, never as actual destinations users stay on, and purge their storage shortly after.

THIS IS EXACTLY THE HEURISTIC FALSE-POSITIVE RISK FLAGGED IN CHAPTER 3

The detection is pattern-based, not based on a list of known trackers — it's looking at *behaviour* (redirect-then-bounce, short visit duration, no genuine user engagement) rather than confirmed tracking intent. A legitimate site that happens to redirect through itself quickly, or gets visited briefly as an intermediate step in some navigation flow, can trigger the same detection with zero actual tracking happening.

Plausible Causes on osztromok.com — A Diagnostic Checklist

Without inspecting the live site's actual navigation flow, these are the most common legitimate (non-tracking) causes of this exact warning, ranked by how often they actually turn out to be the culprit:

🔁 A redirect chain on load

HTTP → HTTPS redirect, or a www/non-www redirect, or an old URL structure redirecting to a new one — if any of these happen in quick succession, especially combined with setting a cookie at each hop, it resembles the bounce pattern closely.

Most likely cause

🍪 A cookie set immediately, then the page navigates away quickly

If /linux itself (or a page it loads through) sets a cookie and then immediately redirects somewhere else — even for a legitimate reason like a login check or an old-URL redirect — that's structurally identical to the bounce pattern from WebKit's perspective.

Most likely cause

🔗 Arriving via a link/embed from another domain

📊 An embedded third-party analytics/widget script

If the warning appeared after following a link from search results, social media, or an embedded reference elsewhere, the "bounce" might be the click-through itself, especially if osztromok.com briefly redirects before settling on the final page.

Possible

If any analytics tool, ad, or embedded widget on the page itself performs its own redirect-based tracking technique, WebKit may be flagging the embedded tracker's behaviour rather than anything osztromok.com is doing directly.

Less likely, but worth ruling out

How to Actually Diagnose It

1. **Open Safari's Web Inspector → Network tab, then navigate to the affected page fresh**
   Look specifically for a chain of 3xx redirect responses before the final 200 — that chain, if it exists, is the prime suspect.

2. **Check the Application/Storage tab for what cookies osztromok.com actually sets, and when**
   If a cookie gets set during one of those redirect hops rather than on the final settled page, that's very likely the exact trigger.

3. **Check the server config (Apache/nginx) for the actual redirect rules in play**
   HTTP→HTTPS redirects, trailing-slash redirects, and old-URL redirects are all extremely common and entirely legitimate — the fix usually isn't removing the redirect, just not setting cookies during it (set cookies only on the final destination response).

4. **If a third-party script is implicated, check its documentation for known ITP-related guidance**
   Major analytics and ad platforms have published guidance for ITP compliance specifically — often a configuration flag rather than removing the integration outright.

THE PRACTICAL FIX, ONCE DIAGNOSED, IS USUALLY SMALL

In the most common case (a redirect chain that also sets a cookie), the fix is simply moving the cookie-setting logic to happen on the final destination page rather than during an intermediate redirect — the redirect itself can stay exactly as it is. This is a configuration/code change, not something that requires removing legitimate functionality.

CHAPTER 4 QUICK REFERENCE

- **Bounce tracking** — briefly redirecting through a domain as a first-party visit specifically to set tracking cookies, then bouncing back
- **The warning is a storage purge, not a connection error** — separate from the ECONNREFUSED routing issue investigated earlier
- **Detection is heuristic/behavioural**, not list-based — legitimate sites can trigger false positives
- **Most likely cause on osztromok.com:** a redirect chain (HTTP→HTTPS, www, old-URL) that also sets a cookie during one of the hops
- **Diagnose with:** Web Inspector Network tab (look for a 3xx chain) + Storage tab (check when cookies actually get set)
- **Likely fix:** set cookies only on the final destination response, not during intermediate redirects — the redirect itself is fine to keep
- **Next chapter:** CORS — Same-Origin Policy, preflight requests, and reading real CORS error messages

CHAPTER 5: CORS

CHAPTER 5

CORS — Cross-Origin Resource Sharing

The Same-Origin Policy, preflight requests, and how to actually read and fix a real CORS error

CORS is one of the most commonly misunderstood parts of web development — not because it's conceptually difficult, but because the error message it produces in the console rarely explains what's actually wrong. This chapter covers the underlying policy CORS exists to relax, what a "preflight" request actually is, and how to read a real CORS error well enough to fix it directly rather than guessing.

The Same-Origin Policy — What CORS Exists to Relax

Browsers enforce a default security rule called the Same-Origin Policy: JavaScript running on one origin cannot read the response of a request made to a different origin, unless that other origin explicitly allows it. An "origin" is the combination of scheme, host, and port — all three have to match exactly.

URL	COMPARED TO HTTPS://OSZTROMOK.COM	SAME ORIGIN?
`https://osztromok.com/linux`	Different path only	✅ Same origin
`http://osztromok.com`	Different scheme (http vs https)	❌ Different origin
`https://blog.osztromok.com`	Different host (subdomain)	❌ Different origin
`https://osztromok.com:8080`	Different port	❌ Different origin

THIS IS WHY A SUBDOMAIN FEELS LIKE "THE SAME SITE" BUT ISN'T, TO THE BROWSER

`osztromok.com` and `blog.osztromok.com` are different origins as far as the Same-Origin Policy is concerned, even though they're clearly part of the same overall website. This is exactly why Chapter 2's `Domain` cookie attribute exists — it's one of the few mechanisms that deliberately bridges origins within the same site.

What CORS Actually Does

CORS is a set of HTTP headers that let a server explicitly opt in to allowing requests from other origins — without it, the Same-Origin Policy blocks the response from being readable by the requesting page's JavaScript, even if the request itself technically succeeded on the server's end.

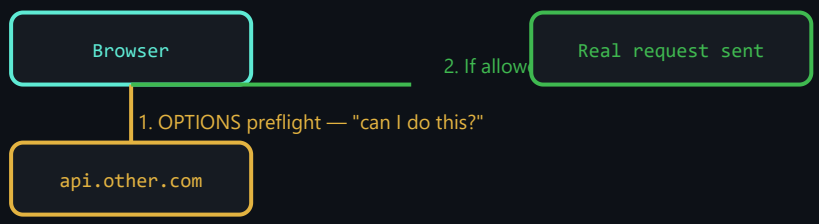
```
// The server includes this header in its response to allow cross-origin reads:
Access-Control-Allow-Origin: https://osztromok.com

// Or, less restrictively (use with caution — see warning below):
Access-Control-Allow-Origin: *
```

CORS IS ENFORCED BY THE BROWSER, NOT THE SERVER

A common misconception: the request to the server still happens, and the server still processes it — CORS only controls whether the browser lets the requesting page's JavaScript *read the response*. This is why CORS errors only ever show up in browser-based JavaScript (fetch, XMLHttpRequest) and never affect server-to-server requests, curl, or Postman, which don't enforce this policy at all.

Preflight Requests — The OPTIONS Request You Didn't Write

For anything beyond a simple GET/POST with standard headers, the browser sends an automatic `OPTIONS` request first — a "preflight" — asking the server's permission before sending the real request at all.




```
The preflight happens automatically, invisibly to your code — you never write the OPTIONS request yourself
```

What triggers a preflight

- **Non-GET/HEAD/POST methods** — PUT, DELETE, PATCH always trigger a preflight.
- **Custom headers** — anything beyond the small set of "CORS-safelisted" headers (like a custom `Authorization` or `X-Custom-Header`) triggers a preflight.

- **Content-Type other than the simple set** — `application/json` (extremely common for APIs) triggers a preflight; `text/plain` doesn't.

```
// The server's response to the preflight OPTIONS request needs these:
Access-Control-Allow-Origin: https://osztromok.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type, Authorization
```

Reading Real CORS Error Messages

`No 'Access-Control-Allow-Origin' header is present`

The server's response didn't include the header at all — either CORS isn't configured server-side, or the request hit an error page (404, 500) that doesn't have CORS headers configured either.

Fix: add the header server-side; check the request isn't actually failing for an unrelated reason first.

`has been blocked by CORS policy: Response to preflight request doesn't pass`

The OPTIONS preflight itself got an unsatisfactory response — often the server doesn't handle OPTIONS requests at all, or restricts allowed methods/headers too narrowly.

Fix: ensure the server explicitly handles OPTIONS and returns the right Allow-Methods/Allow-Headers.

`The value of the 'Access-Control-Allow-Origin' header... must not be the wildcard '*' when credentials mode is 'include'`

A specific, common combination: sending cookies/credentials cross-origin requires an exact origin in the header — the wildcard isn't allowed in that case.

Fix: set Access-Control-Allow-Origin to the exact requesting origin, and Access-Control-Allow-Credentials: true.

NEVER SET ACCESS-CONTROL-ALLOW-ORIGIN: * ON AN ENDPOINT THAT USES COOKIE-BASED AUTH

Combined with credentialed requests, a wildcard origin would let literally any website read authenticated responses on a user's behalf — exactly the kind of cross-site request forgery scenario covered conceptually in this course's earlier SameSite chapter. Browsers actively prevent this exact combination (the error above), which is a safety net, not a bug to work around.

Debugging a Real CORS Issue, Step by Step

- **Check the Network tab first, not just the console error.** Find the actual failed request, check if there's a preceding OPTIONS request, and look at its response headers directly.
- **Confirm the request even reached the server successfully.** A 404 or 500 response often shows up looking like a CORS error in the console, even though the real problem is unrelated to CORS at all.
- **Match the origin exactly.** `http://` vs `https://`, or a trailing slash difference, is enough to make an otherwise-correct Allow-Origin header not match.

CHAPTER 5 QUICK REFERENCE

- **Same-Origin Policy** — scheme + host + port must all match for two URLs to share an origin
- **CORS** — server-sent headers that explicitly opt in to cross-origin reads; enforced by the browser, not the server
- **Preflight (OPTIONS)** — automatic browser check before non-simple requests (custom methods/headers, JSON content-type)
- **Access-Control-Allow-Origin/Methods/Headers** — the core response headers a server needs for CORS to work
- **Wildcard (*) + credentials don't mix** — browsers block this combination deliberately, for good security reasons
- **Debug via the Network tab** — check for a preflight, confirm the request actually succeeded server-side, verify exact origin matching
- **Next chapter:** security headers from the browser's perspective — CSP, HSTS, X-Frame-Options

CHAPTER 6: SECURITY HEADERS

CHAPTER 6

Security Headers — The Browser's Perspective

CSP, HSTS, and X-Frame-Options — what each header actually tells the browser to refuse to do

The earlier "Securing Your Web Server" course covered these same headers from the server-configuration side — what to put in your Apache config. This chapter covers the same headers from the other end: what the browser actually does once it receives them, and why each one closes a specific, real attack rather than being generic best-practice advice.

Content-Security-Policy (CSP) — Restricting What a Page Can Load and Run

CSP tells the browser exactly which sources are allowed to provide scripts, styles, images, fonts, and other resources for a page — anything not explicitly allowed is simply blocked from loading or executing, no matter what HTML or JavaScript tries to request it.

```
Content-Security-Policy: default-src 'self'; script-src 'self' https://cdn.example.com; img-src 'self' data:;
```

DIRECTIVE	CONTROLS
`default-src`	Fallback for any resource type not given its own directive
`script-src`	Where JavaScript is allowed to load from — the most security-critical directive
`style-src`	Where CSS is allowed to load from
`img-src`	Where images are allowed to load from
`connect-src`	Which origins fetch/XMLHttpRequest/WebSocket can connect to — directly relevant to Chapter 5's CORS coverage
`frame-ancestors`	Which sites are allowed to embed this page in an iframe — the modern replacement for X-Frame-Options, covered below

`'self'` means "only this exact origin" (Chapter 5's definition of origin applies directly here). A strict CSP is one of the most effective defences against cross-site scripting (XSS) — even if an

attacker manages to inject a `<script>` tag pointing at their own domain, a correctly configured `script-src` simply refuses to execute it.

> **CSP VIOLATIONS SHOW UP IN THE CONSOLE, AND CAN BE REPORTED WITHOUT BLOCKING ANYTHING**
>
> `Content-Security-Policy-Report-Only` applies the same policy but only logs violations to the console (and optionally a reporting endpoint) without actually blocking anything — the standard way to test a new policy against real traffic before switching it to enforcing mode.

HSTS — Forcing HTTPS for Every Future Visit

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

Once a browser receives this header from a site, it remembers (for the given `max-age`, here one year) to **never attempt an HTTP connection to that domain again** — even if a user explicitly types `http://` or clicks an old http link, the browser silently rewrites it to https before ever sending a request.

Without HSTS	With HSTS
A user typing or clicking an http:// link makes one real (insecure) request to the server before any redirect to HTTPS can happen — that single request is interceptable on an untrusted network.	After the first HTTPS visit, the browser enforces HTTPS internally — no insecure request is ever sent again for that domain, closing the exact window the attack above relies on.
Vulnerable to: SSL-stripping attacks on the first connection	Protects: every subsequent visit, automatically

> **INCLUDESUBDOMAINS AND PRELOAD ARE COMMITMENTS, NOT CASUAL FLAGS**
>
> `includeSubDomains` applies HSTS to every subdomain too — meaning every subdomain must have valid HTTPS working, or they become entirely unreachable. `preload` submits the domain to a hardcoded list baked into browsers themselves, applying HSTS from a user's very first visit ever — genuinely difficult to reverse once submitted. Both are worth using deliberately, not by default, on a domain you're fully confident will keep working HTTPS indefinitely.

X-Frame-Options and frame-ancestors — Controlling Embedding

```
X-Frame-Options: SAMEORIGIN
// Modern equivalent, more flexible, part of CSP:
Content-Security-Policy: frame-ancestors 'self';
```

These headers control whether another site can embed your page inside an `<iframe>` — without this restriction, an attacker can layer invisible iframes of your real site under deceptive buttons on their own page, tricking a logged-in user into clicking something on your site without realising it (a technique called clickjacking).

DENY

No site, including your own, can embed this page in a frame at all.

SAMEORIGIN

Only the exact same origin (Chapter 5's definition) can embed it — your own site can, nobody else can.

frame-ancestors (CSP)

The modern, more flexible replacement — can specify a list of explicitly trusted origins allowed to embed the page, not just same-origin-or-nothing.

Checking What's Actually Configured

Browser DevTools → Network tab → click any request → Response Headers shows exactly which of these headers a site is actually sending — the fastest way to verify a server configuration change actually took effect, without guessing.

ONLINE HEADER-SCANNING TOOLS EXIST FOR A QUICK EXTERNAL CHECK

Several free online security-header scanners exist that fetch a URL and grade which of these headers are present and correctly configured — useful for a quick sanity check after deploying a configuration change, complementing the manual DevTools inspection above.

CHAPTER 6 QUICK REFERENCE

- **CSP** — whitelists allowed sources for scripts/styles/images/connections; the main defence against XSS at the browser level
- **Content-Security-Policy-Report-Only** — test a policy's violations without enforcing it yet
- **HSTS** — forces HTTPS for all future visits to a domain, closing the SSL-stripping window on a user's first connection

- **includeSubDomains / preload** — powerful but hard-to-reverse HSTS extensions; use deliberately
- **X-Frame-Options / frame-ancestors** — controls iframe embedding, the main defence against clickjacking
- **Check headers via DevTools Network tab** — Response Headers on any request
- **Next chapter:** session & auth tokens — cookies vs JWT vs server sessions, and where to safely store what

CHAPTER 7: SESSION & AUTH TOKENS

CHAPTER 7

Session & Auth Tokens

Cookies vs JWT vs server-side sessions, and where each piece of auth-related data is actually safe to store

Every chapter so far has been building toward this one — cookies (Ch1-3), security attributes (Ch2), cross-origin behaviour (Ch5), and headers (Ch6) all matter most in exactly this context: keeping a user logged in safely. This chapter compares the two dominant approaches to web authentication, and answers the question that trips up almost every developer at some point: where should a token actually live?

Server-Side Sessions — The Traditional Approach

The server creates a session record (in memory, a database, or Redis) and gives the browser only a short, random session ID — the actual user data and permissions stay entirely on the server, looked up by that ID on every request.

```
// What the browser actually holds — just an opaque ID, no user data
Set-Cookie: session_id=a1b2c3d4e5f6; Secure; HttpOnly; SameSite=Lax

// What the server holds, looked up by that ID:
{
  "user_id": 42,
  "role": "admin",
  "expires": "2026-07-01T12:00:00Z"
}
```

JWT (JSON Web Tokens) — The Stateless Approach

Instead of an opaque ID looked up server-side, a JWT contains the actual claims (user ID, role, expiry) directly, encoded and cryptographically signed — the server can verify it's genuine and unmodified without needing to look anything up in a database at all.

```
eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjo0Miwicm9sZSI6ImFkbWluIn0.SflKxwRJSMeKKF2QT4fwpMeJf36PC
```

Three parts, separated by dots: a header (algorithm info), a payload (the actual claims), and a signature (proving it hasn't been tampered with).

THE PAYLOAD IS ENCODED, NOT ENCRYPTED — ANYONE CAN READ IT

Decoding a JWT's payload is trivial (it's just base64, not encryption) — paste any JWT into a public JWT decoder and the claims are immediately readable by anyone who has the token. The signature prevents *tampering*, not *reading*. Never put a password, a secret, or genuinely sensitive personal data directly in a JWT's payload.

Comparing the Two Approaches

🗄️ Server-Side Sessions

Revocation: instant — delete the session record server-side, the user is immediately logged out everywhere.

Scaling: needs a shared session store (Redis, a database) if running multiple servers, so any server can look up any session.

Best for: traditional web apps with a single backend, anything needing instant logout/revocation (banking, admin panels).

🔑 JWT

Revocation: genuinely hard — a JWT is valid until it expires; there's no central place to "delete" it once issued, without adding back a server-side blocklist (which partly defeats the statelessness).

Scaling: any server can verify the signature independently — no shared session store needed.

Best for: APIs consumed by multiple independent services, mobile app backends, microservice architectures.

JWT'S REVOCATION PROBLEM IS USUALLY SOLVED WITH SHORT EXPIRY + REFRESH TOKENS

Rather than trying to revoke a JWT directly, the common pattern issues a short-lived JWT (minutes) for actual API access, alongside a separate, longer-lived "refresh token" used only to obtain a new JWT. Revoking access then just means invalidating the refresh token server-side — the short-lived JWT expires naturally within minutes regardless.

Where to Store a Token — The Question That Actually Matters

Regardless of which approach is used, the token has to live somewhere in the browser — and this decision has real security consequences, directly building on Chapter 2's Secure/HttpOnly coverage.

STORAGE LOCATION	VULNERABLE TO XSS?	VULNERABLE TO CSRF?	VERDICT
HttpOnly cookie	No — invisible to JS	Yes, without SameSite/CSRF tokens	Best overall, pair with SameSite=Lax/Strict
localStorage	Yes — fully readable by any script	No — never sent automatically	Common but risky if XSS is even slightly possible
sessionStorage	Yes — same exposure as localStorage	No	Same risk as localStorage, just shorter-lived
In-memory JS variable only	Reduced — gone if the script context is destroyed	No	Most XSS-resistant, but lost on every page refresh

THE XSS VS CSRF TRADEOFF IS GENUINELY UNAVOIDABLE, NOT A SIGN YOU'RE DOING IT WRONG

An HttpOnly cookie is safe from XSS (a script literally cannot read it) but needs SameSite protection against CSRF (a malicious site tricking the browser into sending it automatically). localStorage is naturally immune to CSRF (nothing sends it automatically) but fully exposed if XSS is ever possible anywhere on the page. There is no single storage location immune to both — the real defence is preventing XSS in the first place (Chapter 6's CSP) AND using SameSite correctly (Chapter 2), not picking a "safer" storage location as a substitute for either.

A Sensible Default for Most Projects

- **Store the actual session/auth token in an HttpOnly, Secure cookie with SameSite=Lax.** This is the combination covered across Chapters 2 and this one that resists the most realistic attack scenarios.
- **Use a strict CSP (Chapter 6)** as the primary defence against XSS, rather than relying on storage location alone.
- **If building a separate API consumed by a mobile app or third-party service** (where cookies don't naturally apply), JWTs with short expiry plus refresh tokens are the standard, well-understood pattern.

CHAPTER 7 QUICK REFERENCE

- **Server-side sessions** — opaque ID in a cookie, real data server-side; instant revocation, needs a shared store at scale

- **JWT** — self-contained, signed claims; no database lookup needed, but genuinely hard to revoke before expiry
- **JWT payload is readable by anyone** — signed against tampering, not encrypted against reading
- **JWT revocation pattern:** short-lived JWT + longer-lived refresh token, revoke the refresh token
- **HttpOnly cookie** — safe from XSS, needs SameSite against CSRF; **localStorage** — safe from CSRF, fully exposed to XSS
- **No storage location is immune to both** — prevent XSS via CSP, prevent CSRF via SameSite, rather than relying on storage choice alone
- **Sensible default:** HttpOnly + Secure + SameSite=Lax cookie for session tokens; JWT + refresh tokens for separate/mobile APIs
- **Next chapter (capstone):** debugging real browser console errors — a practical walkthrough of common storage/CORS/cookie messages

CHAPTER 8: DEBUGGING REAL CONSOLE ERRORS

CHAPTER 8 · FINAL CHAPTER · CAPSTONE

Debugging Real Browser Console Errors

A practical decoder for the storage, cookie, and CORS error messages you'll actually encounter — tying every chapter of this course together

Every previous chapter explained a mechanism in isolation. In practice, errors show up in the console without that context — just a red message and a stack trace. This final chapter works backward from real error messages to the right chapter's explanation, so the console becomes something you can actually read rather than just react to.

```
Refused to set unsafe header "Cookie"
```

WHAT'S HAPPENING

Code attempted to set the Cookie header manually via JavaScript (e.g. in a fetch request) — browsers refuse this for security reasons; cookies must go through `document.cookie` or be set by the server's response headers.

FIX

Don't try to set Cookie directly in a fetch/XHR request. If the goal is sending existing cookies with a cross-origin request, use `credentials: 'include'` in the fetch options instead — the browser attaches the actual cookie automatically.

Related: Chapter 1 (cookies are the only mechanism with automatic server transmission)

```
Cookie "session_id" has been rejected for invalid domain.
```

WHAT'S HAPPENING

The server tried to set a cookie with a Domain attribute that doesn't match (or isn't a valid parent domain of) the responding server's own hostname — browsers reject this outright as a security measure, since a server should never be able to set cookies for an unrelated domain.

FIX

Check the exact Domain value being set server-side — it must be the current domain or a true parent of it (e.g. a request from `api.osztromok.com` can set `Domain=osztromok.com`, but never `Domain=somethingelse.com`).

Related: Chapter 2 (Domain attribute scoping)

```
The state of "example.com" was recently purged because it was detected as a bounce tracker.
```

WHAT'S HAPPENING

WebKit's ITP flagged this domain as matching a redirect-then-bounce pattern and cleared its storage — the real warning this entire course was built around.

FIX

Check the Network tab for a 3xx redirect chain on this domain, and check whether a cookie is being set during one of the intermediate hops rather than the final destination page.

Related: Chapter 4 (full diagnostic walkthrough)

```
Access to fetch at 'https://api.example.com/data' from origin 'https://osztromok.com' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

WHAT'S HAPPENING

The requesting page's JavaScript made a cross-origin request, and the server's response didn't include permission for that origin to read it.

FIX

Confirm the request actually succeeded server-side first (check the Network tab's status code) — then add the appropriate Access-Control-Allow-Origin header server-side if it's genuinely missing.

Related: Chapter 5 (full CORS error decoder)

```
Refused to frame 'https://osztromok.com/' because an ancestor violates the following Content Security Policy directive: "frame-ancestors 'self'".
```

WHAT'S HAPPENING

Another site tried to embed this page in an iframe, and the page's own CSP explicitly disallows that — working exactly as intended, not a bug.

FIX

Only relevant to fix if the embedding was actually supposed to work — if so, add the embedding site's origin to `frame-ancestors`. If the embedding wasn't expected at all, this message means the clickjacking protection did its job.

Related: Chapter 6 (X-Frame-Options / frame-ancestors)

```
Uncaught (in promise) DOMException: Failed to execute 'setItem' on 'Storage': Setting the value of 'token' exceeded the quota.
```

WHAT'S HAPPENING

localStorage has a size limit (Chapter 1, roughly 5-10MB depending on browser) — this means the site is trying to store more than that limit allows, often from accumulated data over time rather than one large write.

FIX

Check what's actually accumulating in localStorage via DevTools → Application/Storage tab — often old cached data that was never cleaned up. Consider whether the data genuinely belongs in localStorage at all, or would suit IndexedDB's much larger capacity better.
Related: Chapter 1 (storage mechanism comparison)

A General Debugging Flow for Any Storage/Cookie/CORS Error

1. **Read the exact error message, don't skim it**
   The specific wording ("rejected for invalid domain" vs "blocked by CORS policy" vs "exceeded the quota") points to a completely different chapter and fix — they're not interchangeable.

2. **Open the Network tab and find the actual request involved**
   Check its status code, response headers, and whether it's part of a redirect chain — most of this course's errors become obvious once the real request/response is visible rather than just the console summary.

3. **Check the Application/Storage tab for the actual current state**
   What cookies/storage actually exist right now, with what attributes — confirms or rules out theories quickly rather than guessing from the error message alone.

4. **Identify which origin is involved, precisely**
   Chapter 5's exact origin definition (scheme + host + port) resolves a large fraction of "why doesn't this work" confusion — a mismatched scheme or subdomain is often the entire problem.

THIS DECODER IS A STARTING POINT, NOT AN EXHAUSTIVE LIST

New browser versions introduce new warnings and refine existing ones regularly — the value of this chapter isn't memorising every possible message, it's recognising the pattern (cookie scoping issue,

CORS, storage quota, tracking prevention) quickly enough to know which earlier chapter's concepts actually apply.

CHAPTER 8 QUICK REFERENCE — AND COURSE WRAP-UP

- **Read the exact wording** of console errors — different phrasings point to genuinely different problems and chapters
- **Network tab** — confirms actual request/response status, headers, and redirect chains
- **Application/Storage tab** — confirms actual current cookie/storage state directly
- **Most confusion traces back to an exact origin mismatch** — Chapter 5's scheme+host+port definition
- **Course recap:** storage mechanisms (Ch1) → cookies in depth (Ch2) → first/third-party (Ch3) → bounce tracking (Ch4) → CORS (Ch5) → security headers (Ch6) → auth tokens (Ch7) → debugging (Ch8)
- **This course was built around a real incident** — the osztromok.com bounce-tracker warning from Chapter 4 — and ties directly back into it here as one of the decoded error messages