

Sessions vs. Tokens: Cookies, JWTs, and Where Each One Bites

Thu, Aug 6, 9:00 AM EDT · <https://scottslab.io/posts/sessions-vs-tokens-cookies-jwt>



TL;DR

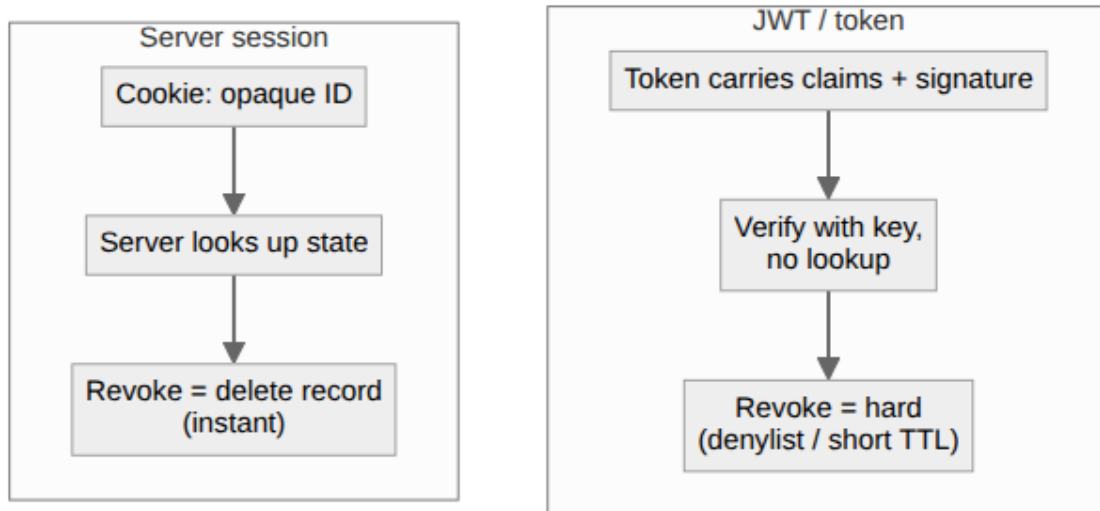
Server sessions are remembered state you can revoke instantly; JWTs are self-contained and stateless but hard to take back before they expire. Where you store the credential is its own trap. HttpOnly cookies dodge XSS but invite CSRF, localStorage does the reverse. For a normal web app, boring server sessions in HttpOnly cookies win; reach for tokens when you actually have the service-to-service problem they solve.

Once someone has authenticated, you have a smaller and more permanent problem: proving it's still them on the next request without asking them to log in every time. There are two honest answers: a server-side session or a self-contained token. Most of the auth bugs I see come from picking one without understanding what it costs.

Sessions: state the server remembers

A session is server-remembered state. You log in, the server stores a record and hands the browser an opaque ID in a cookie; every request presents the cookie, the server looks it up. The nice property is control: logging someone out, killing a compromised session, or forcing re-auth is a

single delete on the server, effective immediately. The cost is that the server has to keep and check that state. That's a non-issue for one app, and a real design question once you have a fleet of services that all need to agree on who's logged in.



Tokens: proof that travels with the request

A token, usually a JWT, flips it around. Instead of a lookup key, the token itself carries the claims (who you are, when it expires) and a signature the server can verify without storing anything. That statelessness is genuinely useful: any service holding the public key can validate the token, no shared session store required, which is why it took over APIs and microservices. But the same property is the trap. Because nothing is looked up, you can't easily take a valid token back. It's good until it expires, and "just revoke it" means rebuilding the very server-side state you adopted JWTs to avoid: a denylist, or short lifetimes plus refresh tokens, which is most of a session with extra steps.

Where you store it, and which bug you accept

Then there's where you put it, which is its own footgun. A cookie can be `HttpOnly`, so JavaScript can't read it, which takes token theft via XSS off the table. But cookies ride along automatically on every request, which is exactly what CSRF abuses, so now you need `SameSite` and anti-CSRF tokens. Put a JWT in `localStorage` to dodge CSRF and you've made it readable by any script that runs on your page, so one XSS and the token walks out the door. There's no placement that dodges both; you're choosing which class of bug you'd rather defend against.



What I actually use

My rule of thumb: for a normal web app with a browser and a login, server sessions in `HttpOnly` cookies are the boring correct answer, and it's what I use here. Instant revocation matters more than saving a lookup. Reach for tokens when you actually have the problem they solve: service-to-service calls, or enough backends that they can't share a session store. Both are fine tools. The mistakes come from choosing the stateless one because it sounds modern, then reinventing state the first time you need to log somebody out.