

OAuth 2.0 and OIDC, Explained by Wiring Up My Own Admin Login

Wed, Aug 5, 4:00 PM EDT · <https://scottslab.io/posts/oauth-oidc-explained-admin-login>



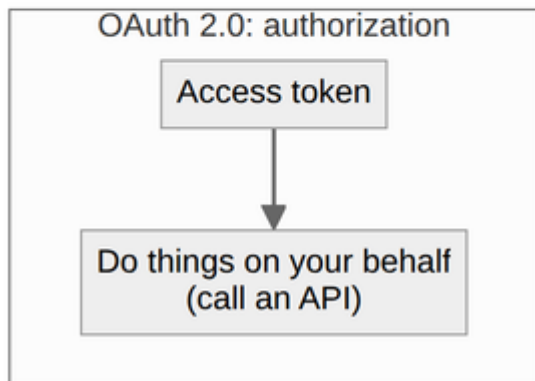
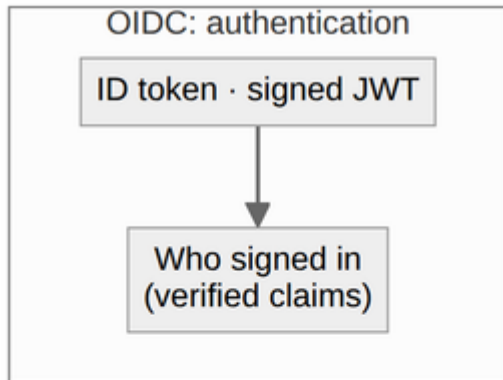
TL;DR

OAuth 2.0 is authorization (a token to do things on your behalf); OIDC is the identity layer on top that actually tells you who signed in. Using this site's Google login as the example: the authorization-code flow keeps the password between you and Google and does the sensitive token exchange server-to-server, and pinning the verified email claim to one address is what makes "only I can log in" true. Because these are bearer tokens, the redirect allow-list and client secret are controls, not paperwork.

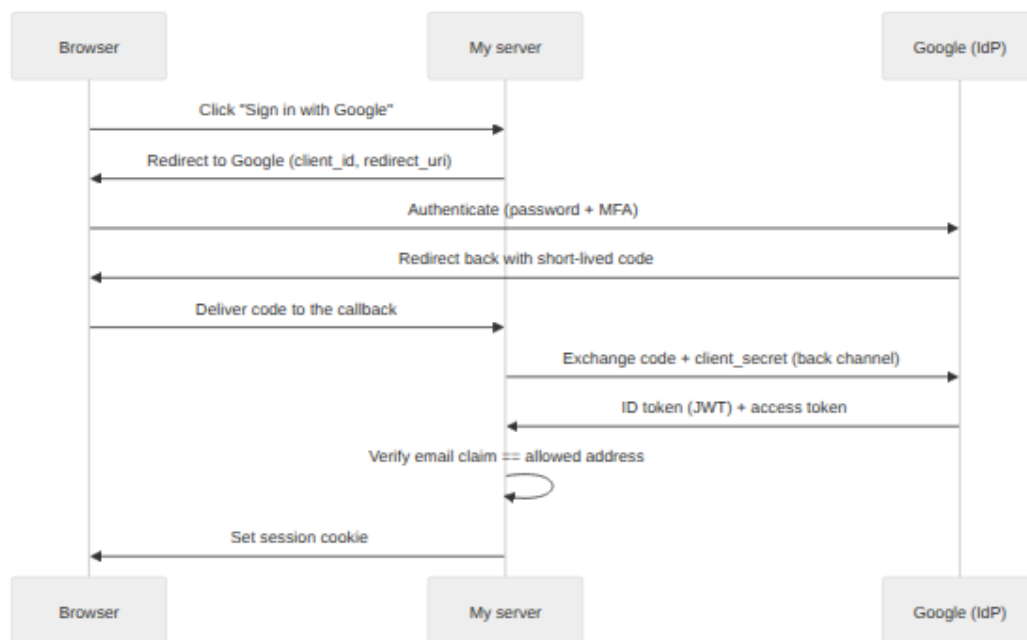
I put Google sign-in on the admin panel of this site, and it's the smallest possible version of a system people find intimidating. The whole feature is: only I can log in, and I never want to store a password to make that true. OAuth 2.0 and OIDC are what turn that into a two-paragraph problem instead of a security project.

The confusion usually starts with conflating the two. OAuth 2.0 is about authorization: getting a token that lets an app do something on your behalf. It was never designed to tell an app *who you are*, and the early years of people bolting identity onto it by hand produced a lot of subtle bugs. OIDC is the thin, standardized identity layer on top: same flow, but you also get an ID token, a

signed JWT that actually asserts "this is who signed in." For a login button you want OIDC, because you're authenticating, not just authorizing.



What actually happens when I click "Sign in with Google" is the authorization code flow. My server sends the browser to Google with my client ID and a callback URL. Google authenticates me. That part is entirely between me and Google, my server never sees the password. Then it redirects back to the callback with a short-lived code. My server then exchanges that code, plus its client secret, directly with Google for tokens. The reason for the two-step dance instead of just handing a token to the browser is that the sensitive exchange happens server-to-server, where the client secret lives and the code can't be lifted out of a URL or someone's browser history.



The part I care about as the person running the thing: I pinned it to exactly one identity. Google hands back a verified email claim, and my callback checks it against a single allowed address before it establishes a session. Anyone else authenticates to Google perfectly well and still gets bounced at my door. And because these are bearer tokens, where whoever holds the token can use it, the redirect-URI allow-list and the client secret aren't paperwork; they're the controls that stop the code from being redeemed by someone else. One modern addition worth calling out: the current OAuth security best practice (RFC 9700, 2025) makes PKCE the baseline. The client commits to a random secret hashed into the authorization request and only reveals the original at the token exchange, so an intercepted code is worthless without the matching verifier. It's mandatory for public clients and recommended even for confidential ones like this that already hold a secret.

I also kept a local password login as a break-glass path for when Google is unreachable, which nicely illustrates the trade-off. OIDC moves the hard parts (credential storage, MFA, breach monitoring) onto an identity provider who does it full-time, at the cost of a dependency on them being up. For a one-person admin panel that's an easy trade. For a product with millions of users it's the same flow and a much longer list of things to get right, but the shape never changes: redirect, authenticate somewhere trusted, exchange a code in the back channel, verify the claims, start a session.