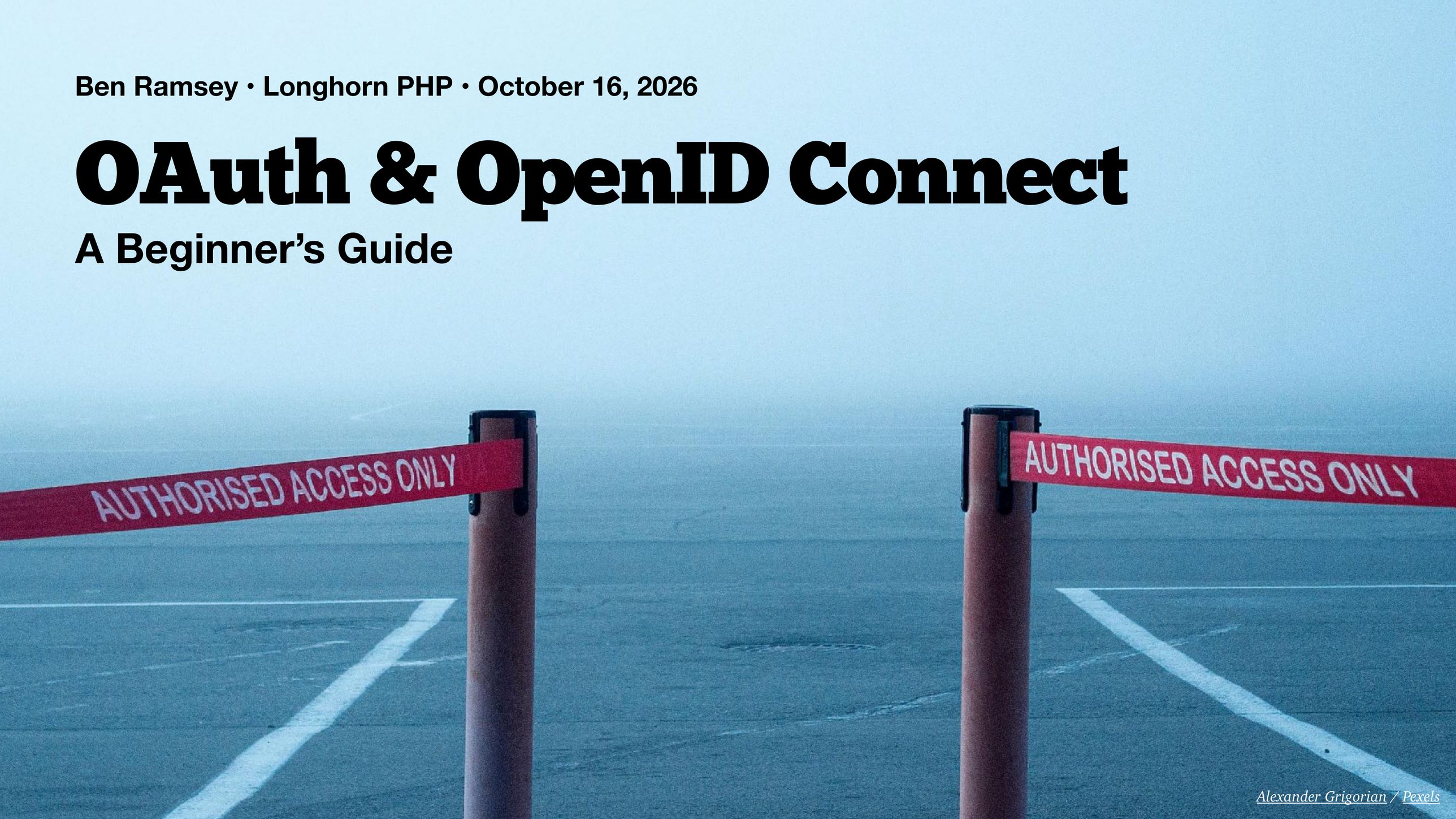
Ben Ramsey • Longhorn PHP • October 16, 2026

OAuth & OpenID Connect

A Beginner's Guide



Alexander Grigorian / Pexels

Who here has signed in with...

Apple?

Google?

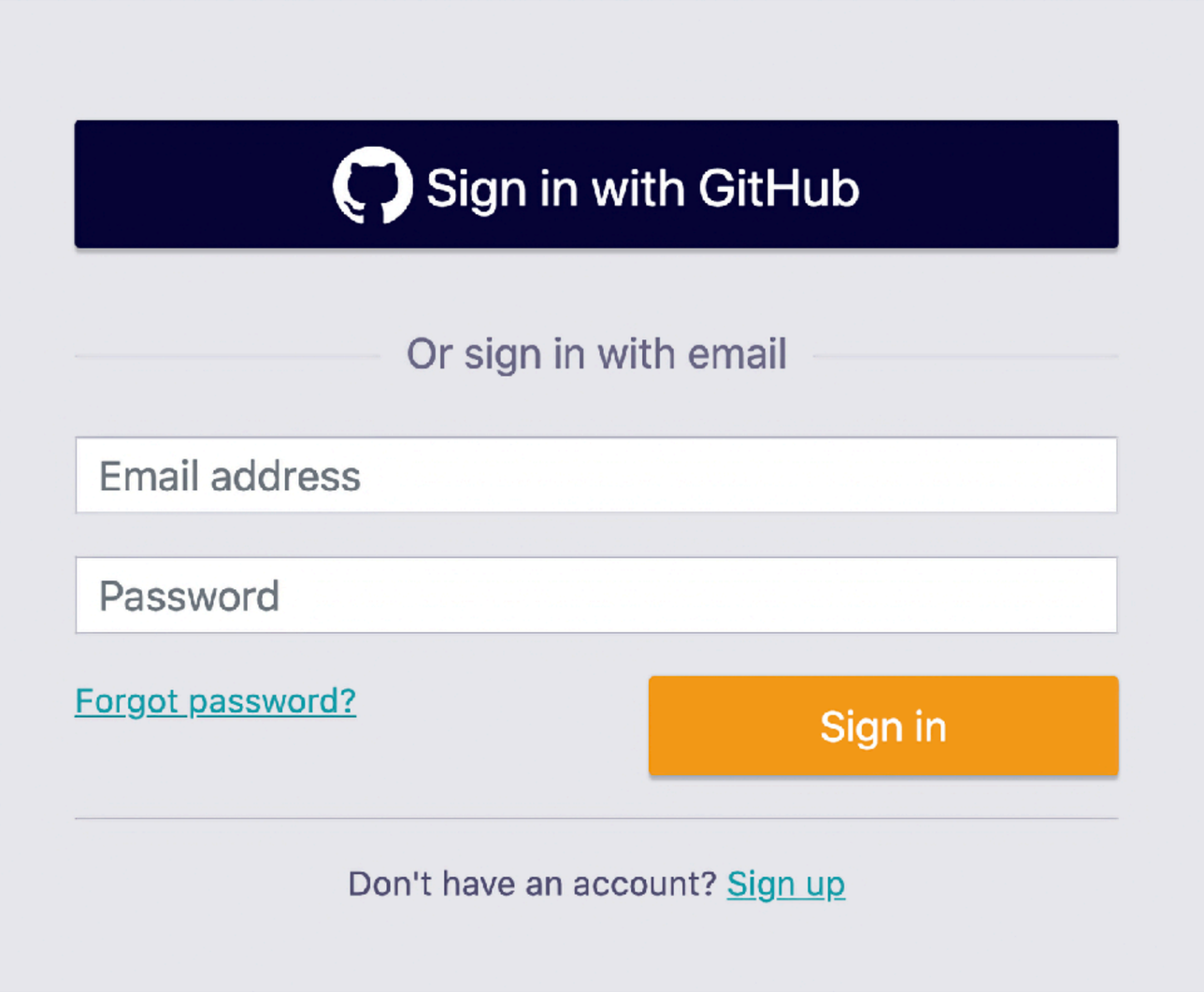
GitHub?

Elissa Garcia / Unsplash

You're trying out a new dev tool.

You click **"Sign in with GitHub."**

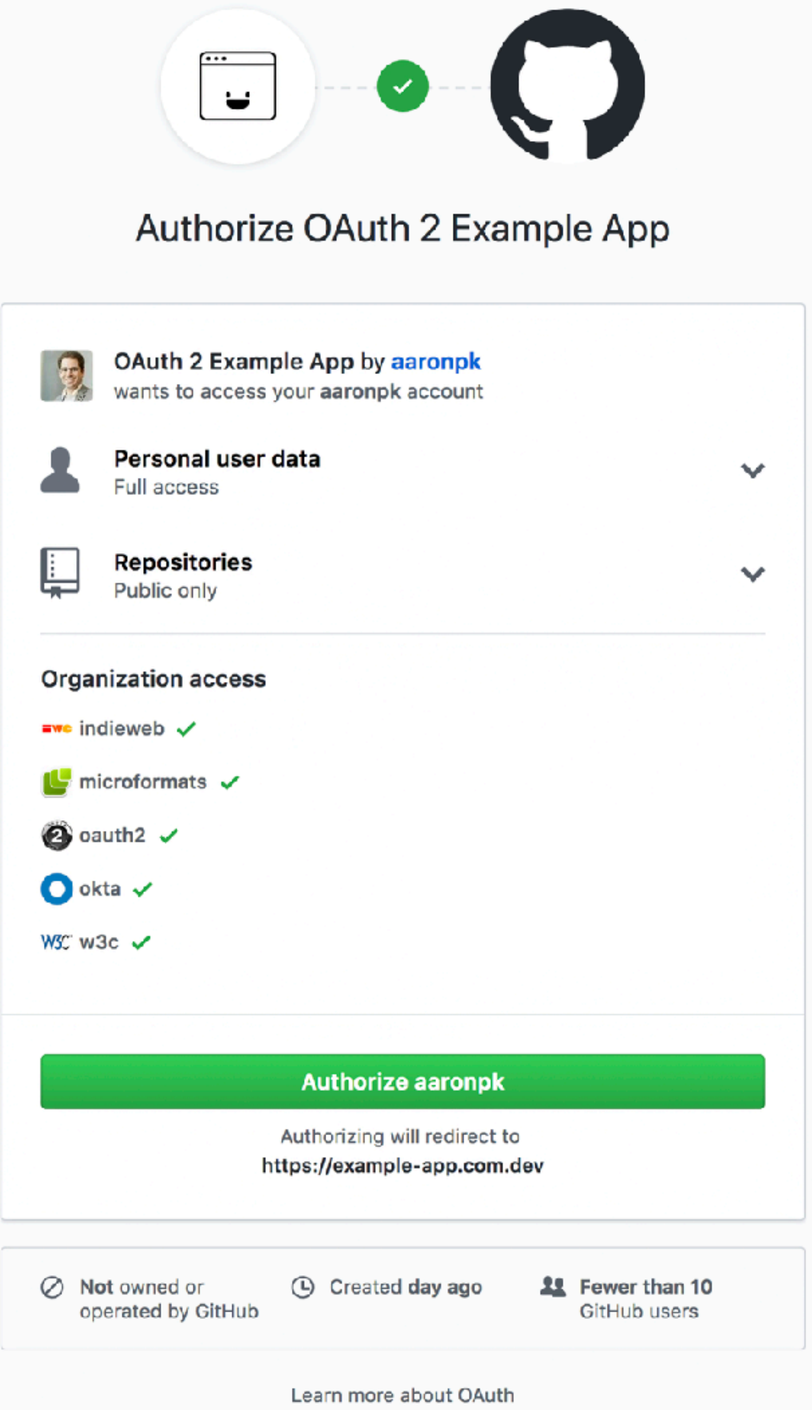
You land on **GitHub**, *not the app*.



GitHub shows you what the app wants access to.

You click **Authorize**.

You're back in the app & logged in.



OAUTH
OAUTH

Alena Darmel / Pexels

Before OAuth...

App A:

We have some cool stuff we can do with your data in App B. May we have access, pretty please?

You:

Sure. I trust you. Here's my user-name and password to App B.

Wow! This mash-up is so cool! I gotta tell all my friends!

...you gave them your password.

Full access
Not just the data they asked for, but everything

No expiry
Access lasts until you change your password

No revocation
Can’t take it back without changing your password

No audit trail
No way to know what they did

iMattSmart / Unsplash

Sanket Mishra / Pexels

The Twitter problem

Early Twitter (2007–2010)

Every third-party app asked for your **Twitter username and password**

TweetDeck, Twitterrific, Seesmic

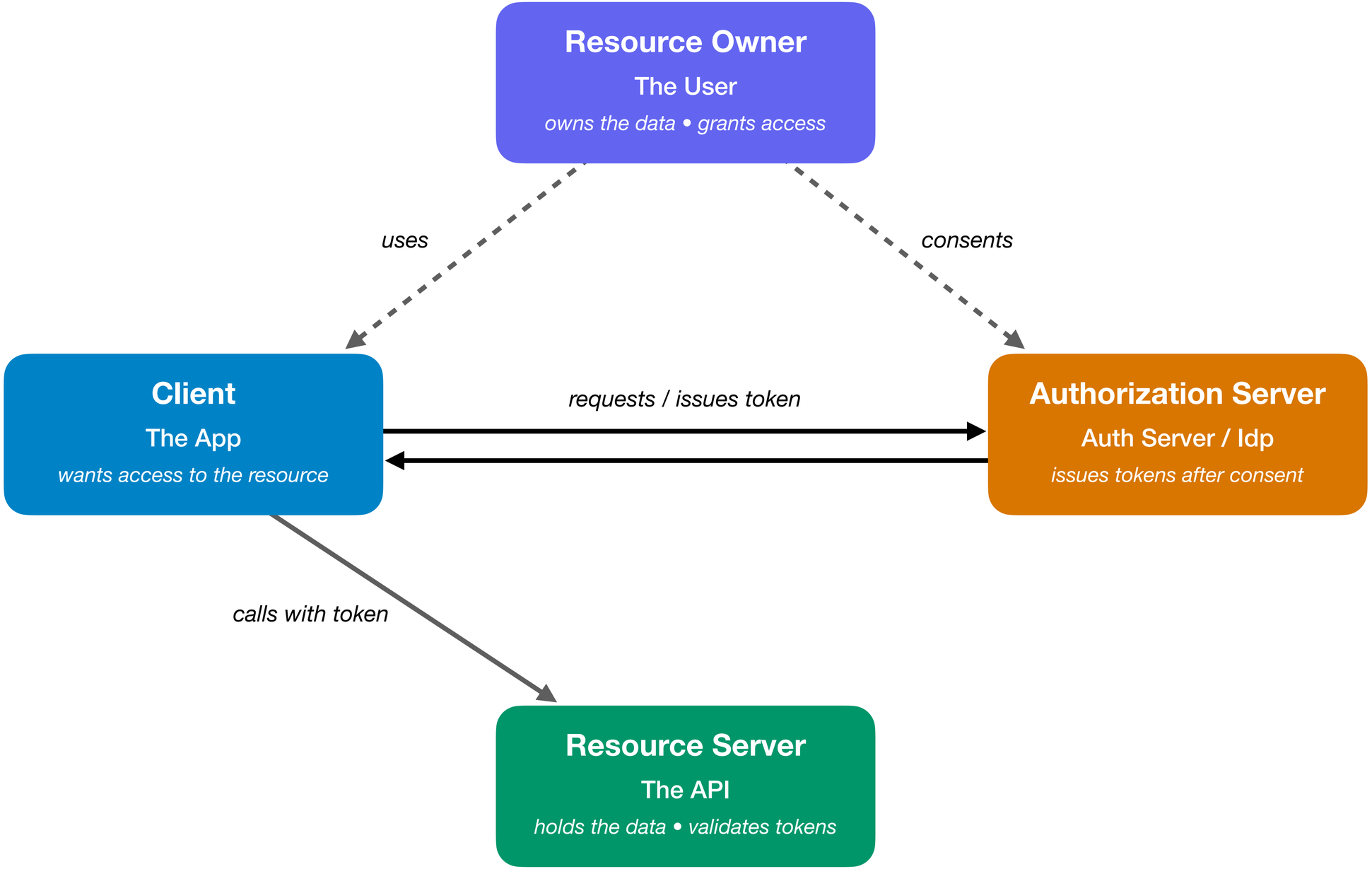
This was just…how it worked

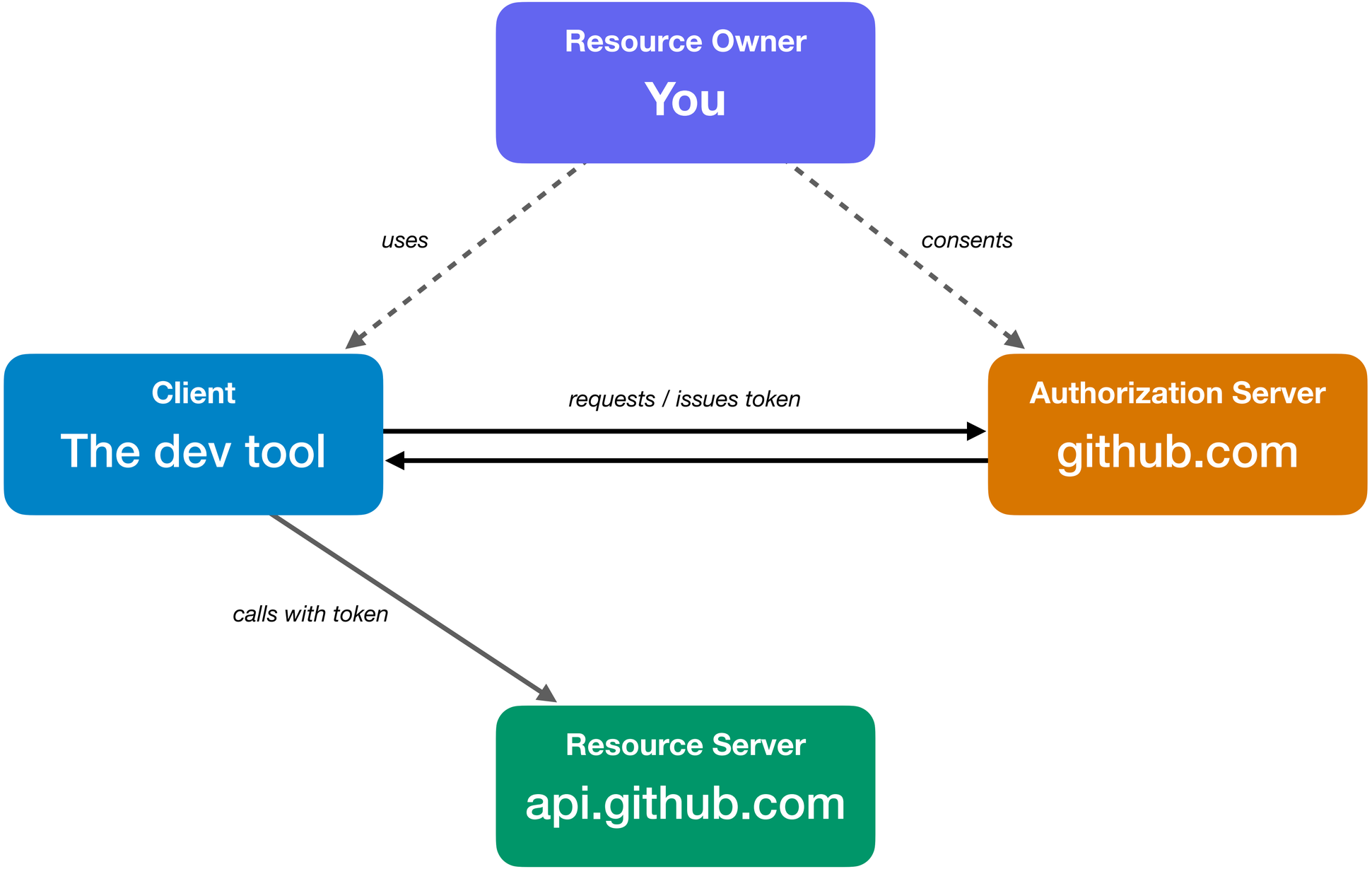
What if, instead of your password…

…you issued a **limited-use token**?

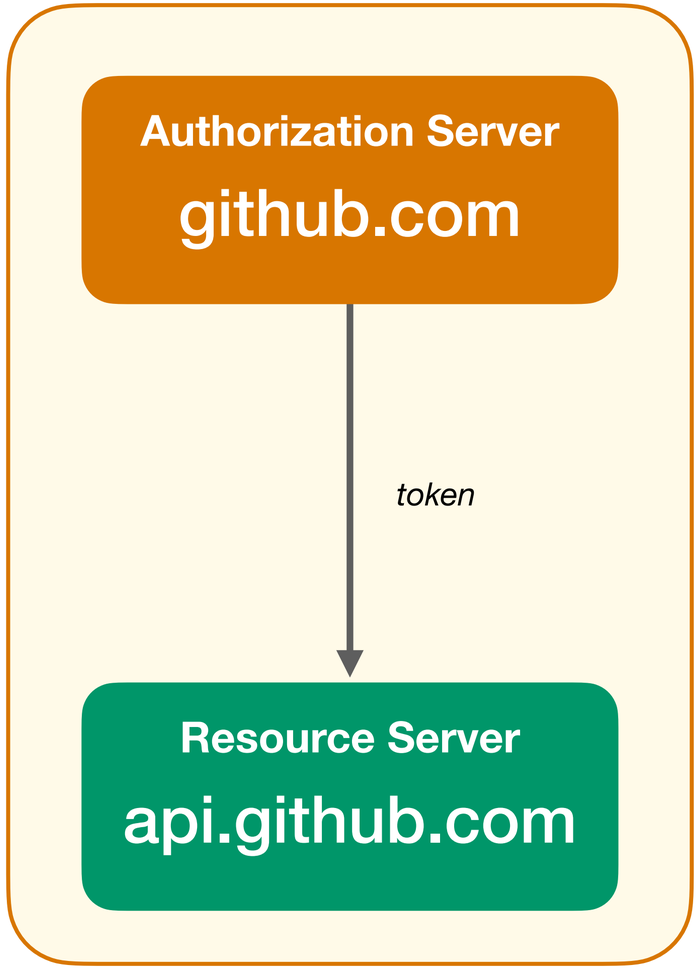
- **Scoped**
- **Time-restricted**
- **Revocable**

OAuth is about **delegated authorization**, granting a third party limited access to your resources, without sharing your password.

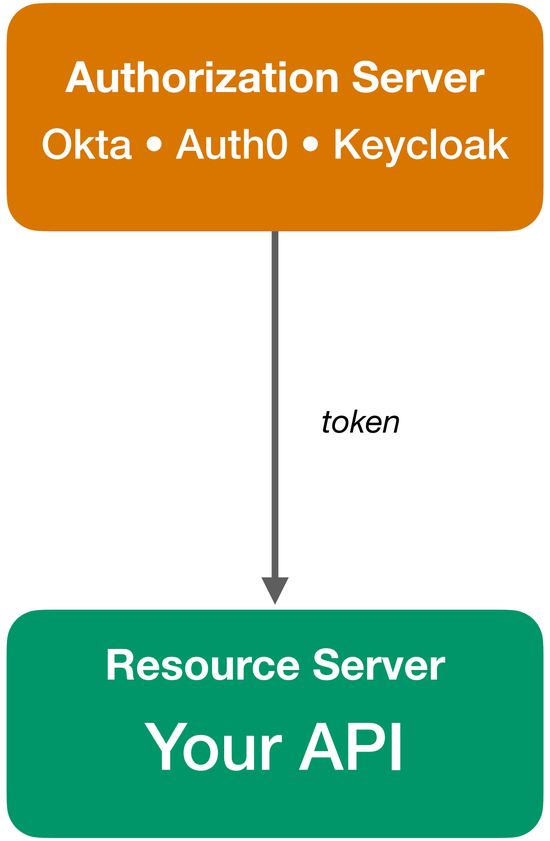

Resource Owner
The User
owns the data • grants access
uses
consents
Client
The App
wants access to the resource
requests / issues token
Authorization Server
Auth Server / Idp
issues tokens after consent
calls with token
Resource Server
The API
holds the data • validates tokens


Resource Owner
You
uses
consents
Client
The dev tool
requests / issues token
Authorization Server
github.com
calls with token
Resource Server
api.github.com

One Organization



Two Organizations



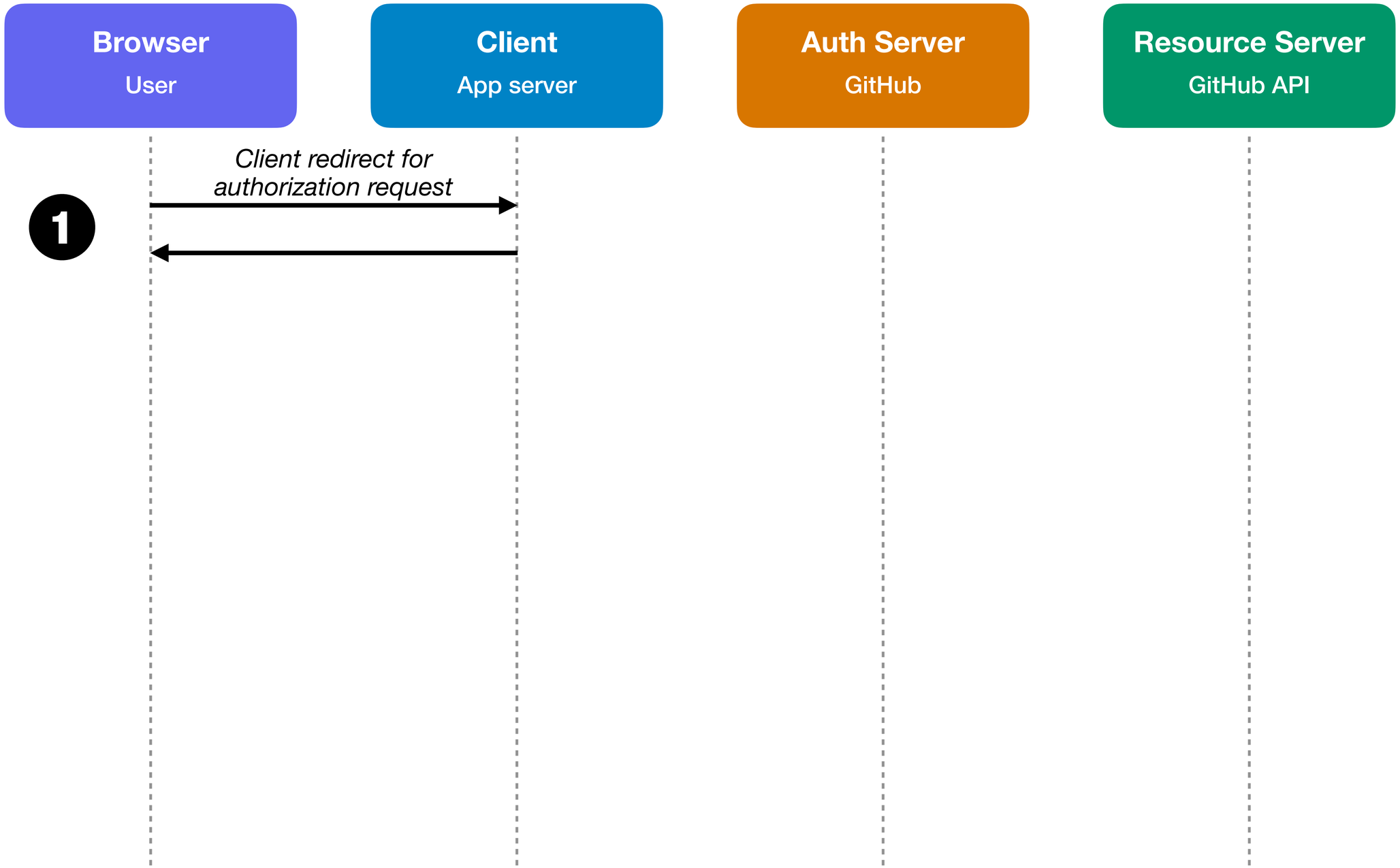
Authorization code flow

The most common, most secure OAuth flow.

Why this one first?

- Used by virtually every web and mobile app
- The access token **never passes through the browser**
- Once you understand this, every other flow is just a variation

BOOM Photography / Pexels


Browser
User
Client
App server
Auth Server
GitHub
Resource Server
GitHub API
1
Client redirect for
authorization request

1. Client redirect

```
https://github.com/login/oauth/authorize
  ?response_type=code
  &client_id=abc123
  &redirect_uri=https://example.com/callback
  &scope=read:user%20repo
  &state=xK9mQ2
```

1. Client redirect

```
https://github.com/login/oauth/authorize
  ?response_type=code
  &client_id=abc123
  &redirect_uri=https://example.com/callback
  &scope=read:user%20repo
  &state=xK9mQ2
```

1. Client redirect

```
https://github.com/login/oauth/authorize
  ?response_type=code
  &client_id=abc123
  &redirect_uri=https://example.com/callback
  &scope=read:user%20repo
  &state=xK9mQ2
```

1. Client redirect

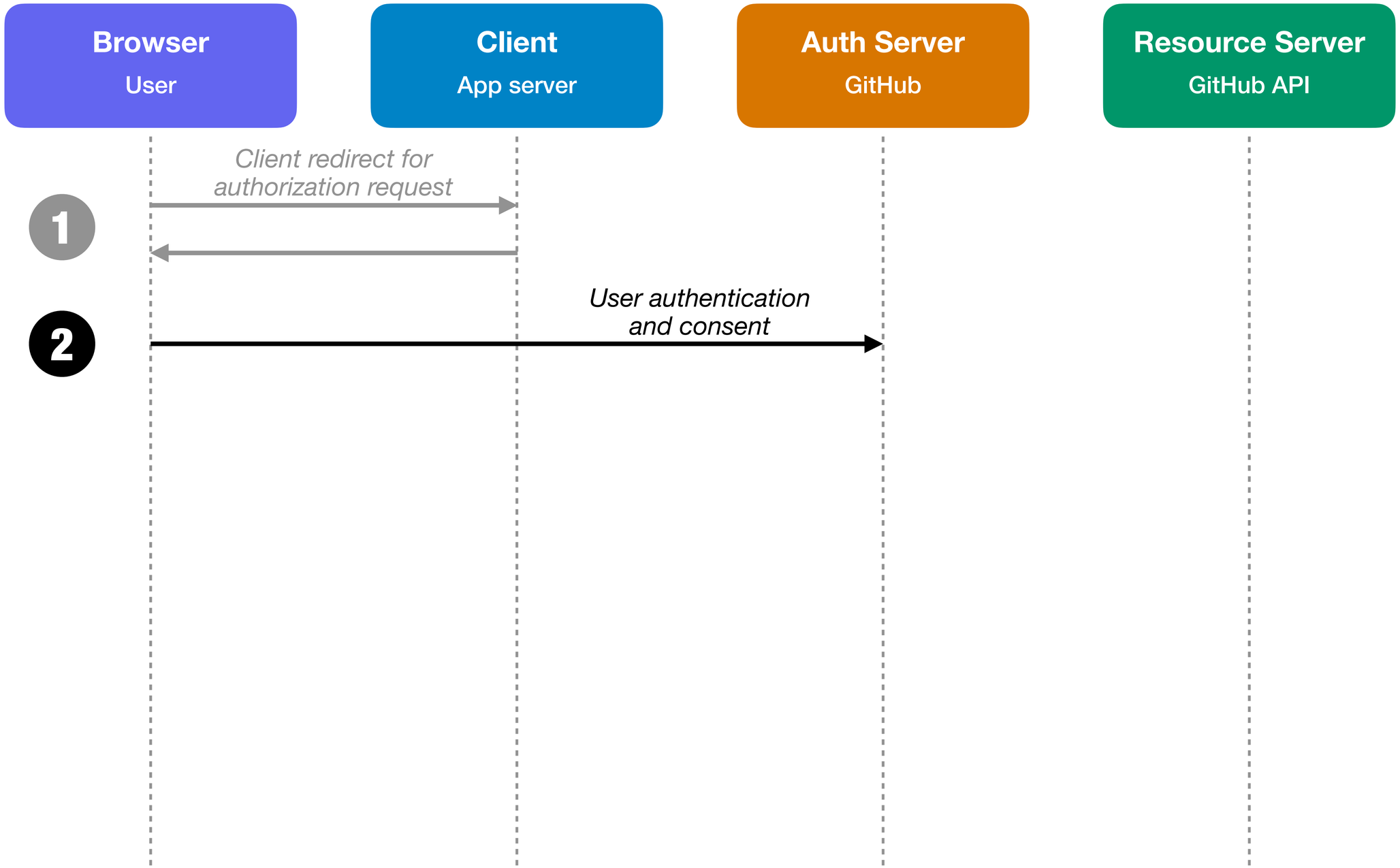
```
https://github.com/login/oauth/authorize
  ?response_type=code
  &client_id=abc123
  &redirect_uri=https://example.com/callback
  &scope=read:user%20repo
  &state=xK9mQ2
```

1. Client redirect

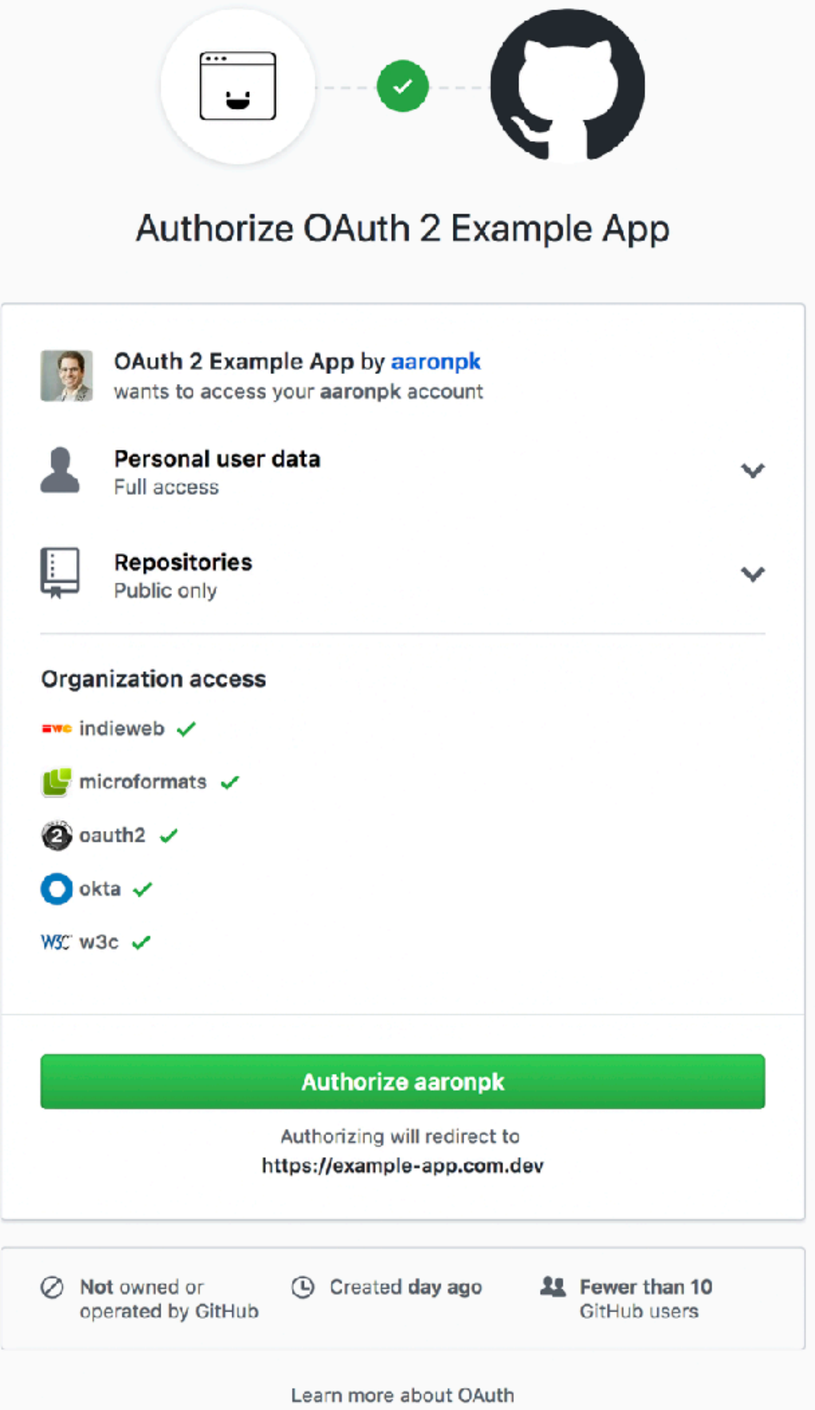
```
https://github.com/login/oauth/authorize
  ?response_type=code
  &client_id=abc123
  &redirect_uri=https://example.com/callback
  &scope=read:user%20repo
  &state=xK9mQ2
```

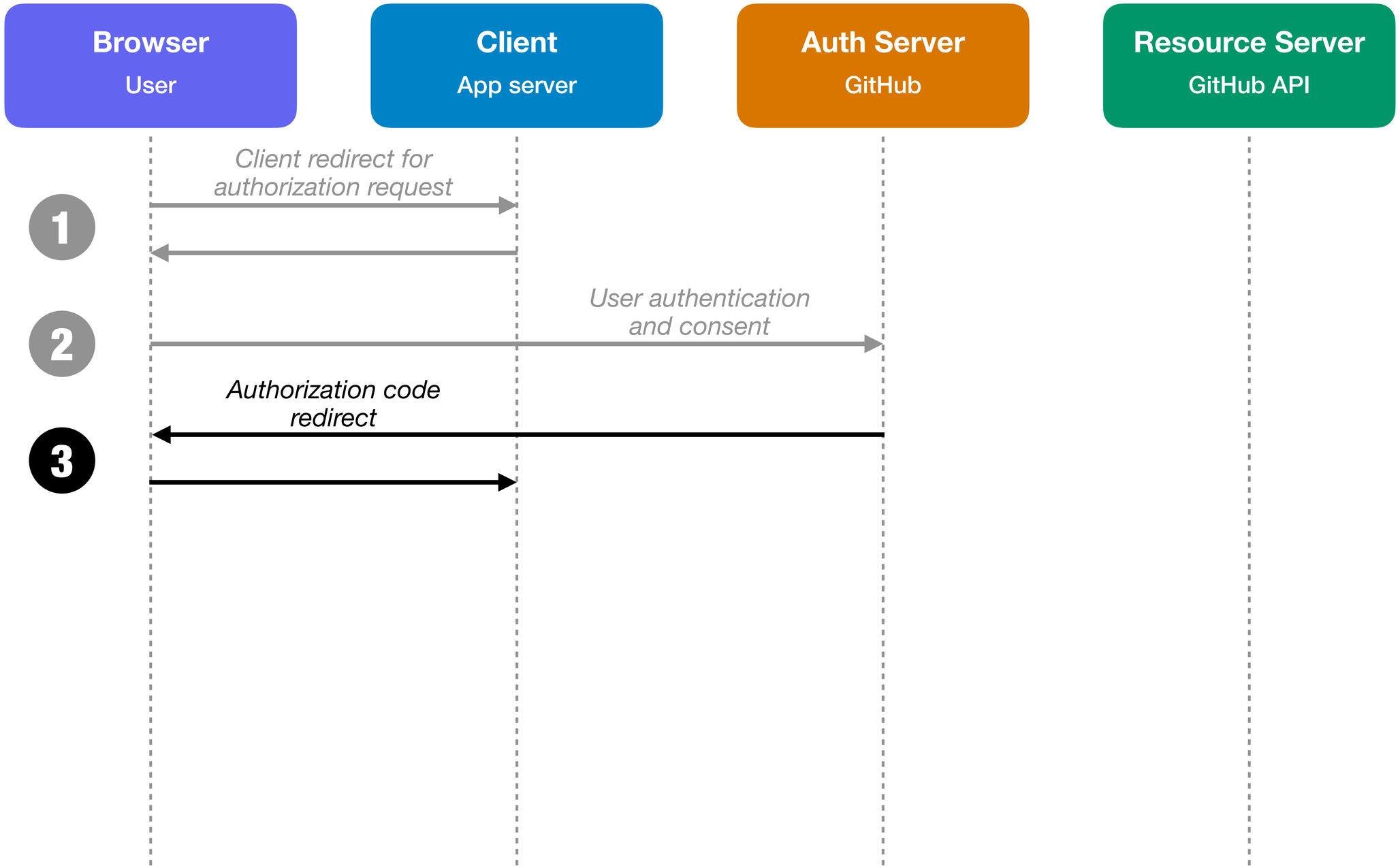
1. Client redirect

```
https://github.com/login/oauth/authorize
  ?response_type=code
  &client_id=abc123
  &redirect_uri=https://example.com/callback
  &scope=read:user%20repo
  &state=xK9mQ2
```


Browser
User
Client
App server
Auth Server
GitHub
Resource Server
GitHub API
Client redirect for authorization request
1
2
User authentication and consent

2. User authentication & consent




Browser
User
Client
App server
Auth Server
GitHub
Resource Server
GitHub API
Client redirect for authorization request
1
User authentication and consent
2
Authorization code redirect
3

3. Authorization code redirect

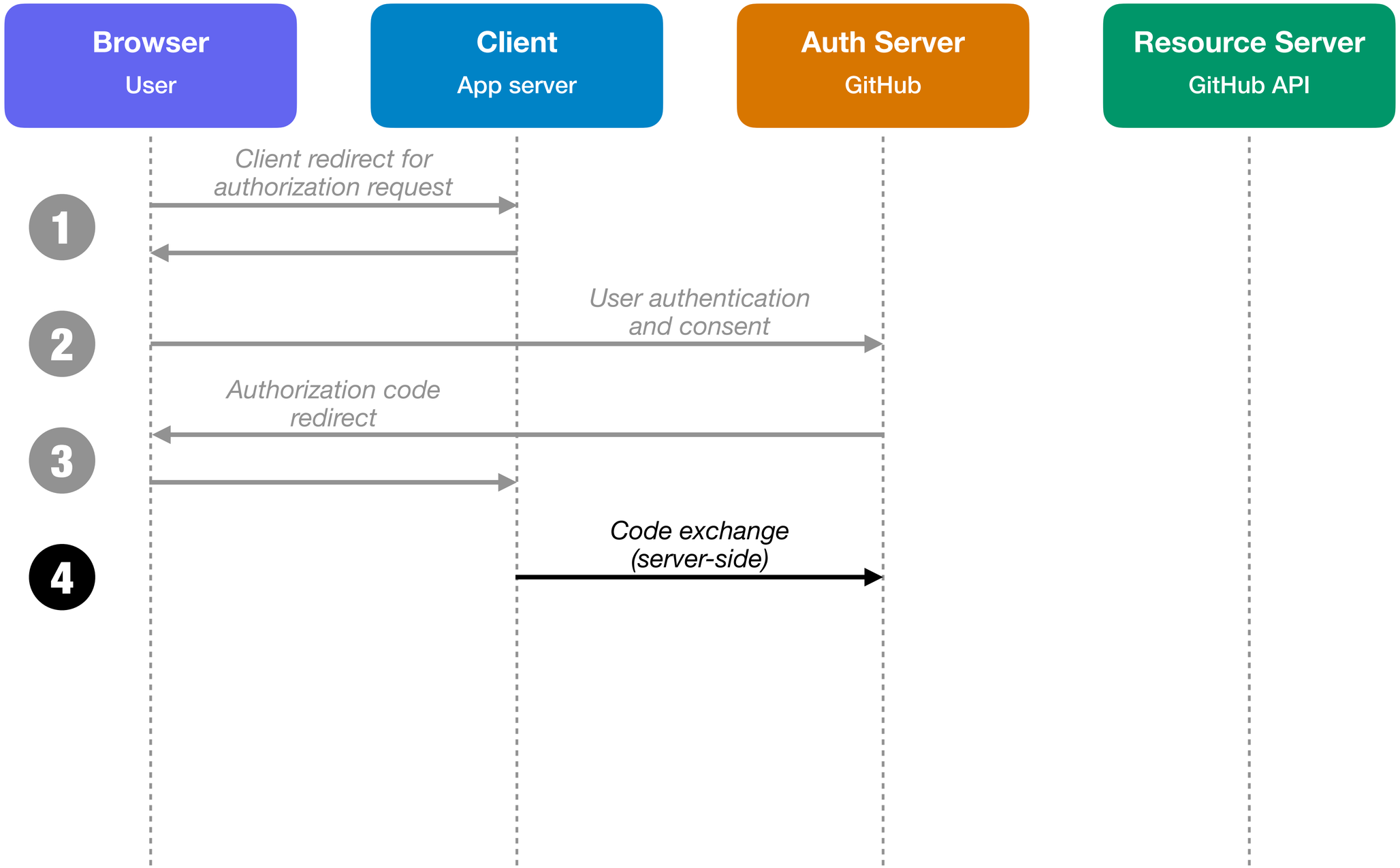
```
https://example.com/callback
  ?code=TEMP_CODE_HERE
  &state=xK9mQ2
```

3. Authorization code redirect

```
https://example.com/callback
  ?code=TEMP_CODE_HERE
  &state=xK9mQ2
```

3. Authorization code redirect

```
https://example.com/callback
  ?code=TEMP_CODE_HERE
  &state=xK9mQ2
```


Browser
User
Client
App server
Auth Server
GitHub
Resource Server
GitHub API
1
Client redirect for authorization request
2
User authentication and consent
3
Authorization code redirect
4
Code exchange (server-side)

4. Code exchange (server-side)

```
POST /login/oauth/access_token HTTP/1.1
host: github.com
content-type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=TEMP_CODE_HERE
&redirect_uri=https://example.com/callback
&client_id=abc123
&client_secret=super_secret
```

4. Code exchange (server-side)

```
POST /login/oauth/access_token HTTP/1.1
host: github.com
content-type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=TEMP_CODE_HERE
&redirect_uri=https://example.com/callback
&client_id=abc123
&client_secret=super_secret
```

4. Code exchange (server-side)

```
POST /login/oauth/access_token HTTP/1.1
host: github.com
content-type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=TEMP_CODE_HERE
&redirect_uri=https://example.com/callback
&client_id=abc123
&client_secret=super_secret
```

4. Code exchange (server-side)

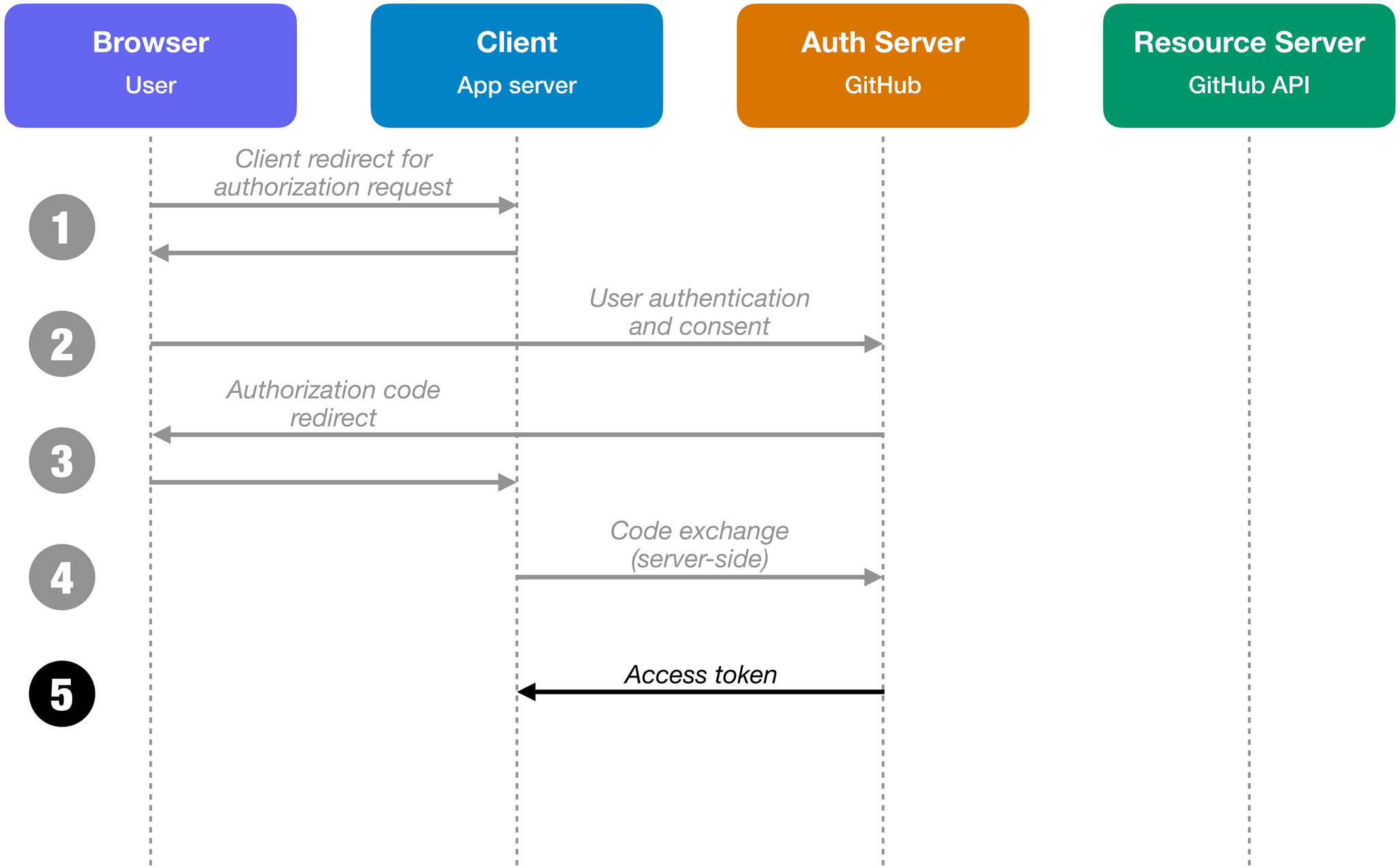
```
POST /login/oauth/access_token HTTP/1.1
host: github.com
content-type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=TEMP_CODE_HERE
&redirect_uri=https://example.com/callback
&client_id=abc123
&client_secret=super_secret
```

4. Code exchange (server-side)

```
POST /login/oauth/access_token HTTP/1.1
host: github.com
content-type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=TEMP_CODE_HERE
&redirect_uri=https://example.com/callback
&client_id=abc123
&client_secret=super_secret
```


Browser
User
Client
App server
Auth Server
GitHub
Resource Server
GitHub API
1
Client redirect for authorization request
2
User authentication and consent
3
Authorization code redirect
4
Code exchange (server-side)
5
Access token

5. Access token


```
{
  "access_token": "gho_16C7e42F292c6912E771…",
  "token_type": "Bearer",
  "scope": "read:user repo",
  "expires_in": 3600
}
```

5. Access token

```
{
  "access_token": "gho_16C7e42F292c6912E771…",
  "token_type": "Bearer",
  "scope": "read:user repo",
  "expires_in": 3600
}
```

5. Access token

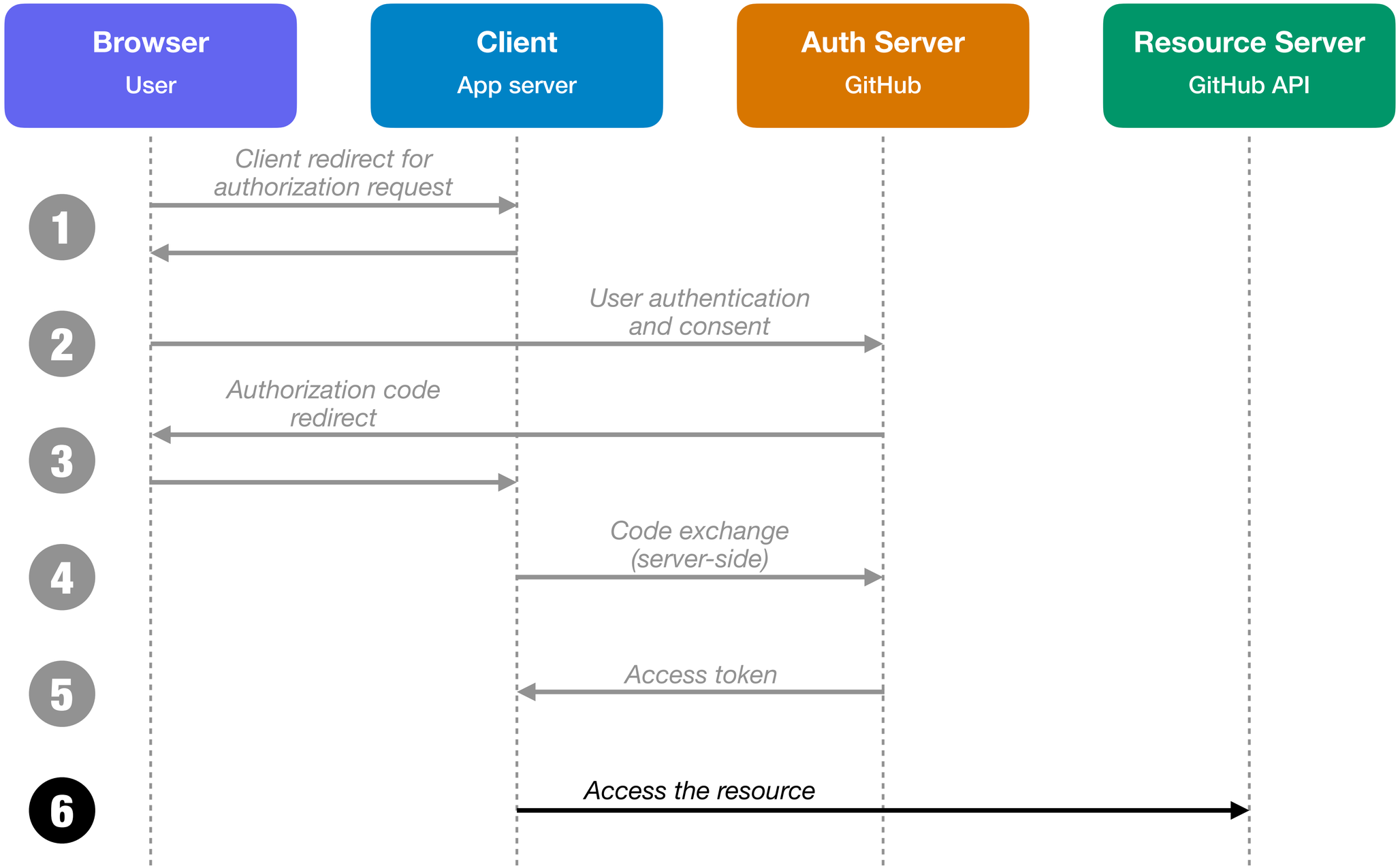

```
{
  "access_token": "gho_16C7e42F292c6912E771…",
  "token_type": "Bearer",
  "scope": "read:user repo",
  "expires_in": 3600
}
```

5. Access token


```
{
  "access_token": "gho_16C7e42F292c6912E771…",
  "token_type": "Bearer",
  "scope": "read:user repo",
  "expires_in": 3600
}
```

5. Access token


```
{
  "access_token": "gho_16C7e42F292c6912E771…",
  "token_type": "Bearer",
  "scope": "read:user repo",
  "expires_in": 3600
}
```


Browser
User
Client
App server
Auth Server
GitHub
Resource Server
GitHub API
1
Client redirect for authorization request
2
User authentication and consent
3
Authorization code redirect
4
Code exchange (server-side)
5
Access token
6
Access the resource

6. Accessing the resource

```
GET /user HTTP/1.1
host: api.github.com
authorization: Bearer gho_16C7e42F292c6912E771…
```

Filip Szalbot / Unsplash

Other grant types

grant type	use case
authorization code	web apps, mobile apps
client credentials	machine-to-machine; no user involved
device code	TVs, CLIs, devices with no browser
~~implicit~~	**deprecated**, not not use

PKCE

Authorization Code for Public Clients

Proof Key for Code Exchange (RFC 7636)

Problem: Some clients can’t keep a secret.

- Native mobile apps ship code to the user's device
- Single-page apps (SPAs) run in the browser
- Any embedded `client_secret` can be extracted

Solution: Replace the secret with a one-time cryptographic challenge.

Required in OAuth 2.1.

Modun Studio / Pexels

You have an **access token.**
Now what?

Pavel Danilyuk / Pexels

Access tokens

The credential your app uses to call an API.

Two popular formats:

Format	How it works
Opaque	A random string; only the Authorization Server knows what it means
JWT	Self-contained; the Resource Server can validate it locally

Both are **short-lived** by design.

Never store in local storage. Use an HttpOnly cookie or server-side session.

Refresh tokens

Long-lived credential used to obtain a new access token without sending the user through the consent flow again.

```
POST /login/oauth/access_token HTTP/1.1
host: github.com

grant_type=refresh_token
&refresh_token=YOUR_REFRESH_TOKEN
&client_id=abc123
&client_secret=super_secret
```

Treat like a password. Store securely, encrypted at rest, server-side only.

Jose Antonio Gallego Vázquez / Unsplash

Scopes

Scopes are the vocabulary of permission.

- Clients *request* scopes
- Users *approve* them
- Tokens *encode* what was granted

Use the **principle of least privilege**. Request only the scopes you need.

GitHub scopes examples

`read:user`	Read your public profile
`repo`	Full access to respositories
`email`	Read your primary email address
`openid`†	OpenID Connect identity

† GitHub doesn't actually have the `openid` scope; we'll talk more about this in a bit.

Lucas S / Unsplash

Token validation

On the resource server

1. Is it genuine?
2. Is it still valid?
3. Is it meant for me?
4. Does it grant the right scope?

And, now, some code...

Arif Riyanto / Unsplash

Introducing league/oauth2-client

It’s the de facto standard for OAuth 2.0 clients in PHP.

- Works with or without a framework
- Provider packages for GitHub, Google, Facebook, Slack, and dozens more
- Handles URL construction, state, token exchange, and API calls

```
composer require league/oauth2-client league/oauth2-github
```

Starting the flow

authorize.php — redirect user to GitHub

```php
use League\OAuth2\Client\Provider\Github;

session_start();

$provider = new Github([
    'clientId'     ⇒ $_ENV['GITHUB_CLIENT_ID'],
    'clientSecret' ⇒ $_ENV['GITHUB_CLIENT_SECRET'],
    'redirectUri'  ⇒ 'https://example.com/callback',
]);

$authUrl = $provider→getAuthorizationUrl([
    'scope' ⇒ ['read:user', 'repo'],
]);

$_SESSION['oauth2state'] = $provider→getState();

header('Location: ' . $authUrl);
```

```php
$provider = new Github([
    'clientId'     ⇒ $_ENV['GITHUB_CLIENT_ID'],
    'clientSecret' ⇒ $_ENV['GITHUB_CLIENT_SECRET'],
    'redirectUri'  ⇒ 'https://example.com/callback',
]);
```

```php
$authUrl = $provider->getAuthorizationUrl([
    'scope' => ['read:user', 'repo'],
]);
```

```php
$_SESSION['oauth2state'] = $provider→getState();

header('Location: ' . $authUrl);
```

Handling the Callback

callback.php – exchange the code, call the API

```php
// Use session_start() and instantiate GitHub provider (see authorize.php).

if (empty($_GET['state']) || $_GET['state'] !== $_SESSION['oauth2state']) {
    unset($_SESSION['oauth2state']);
    throw new RuntimeException('Invalid state; possible CSRF attack');
}

$token = $provider->getAccessToken('authorization_code', [
    'code' => $_GET['code'],
]);

$user = $provider->getResourceOwner($token);

echo 'Hello, ' . $user->getName() . "!\n";
echo 'GitHub: '  . $user->getUrl();
```

```php
// Validate state to prevent CSRF
if (
    empty($_GET['state'])
    || $_GET['state'] ≠ $_SESSION['oauth2state']
) {
    unset($_SESSION['oauth2state']);

    throw new RuntimeException(
        'Invalid state; possible CSRF attack',
    );
}
```

```php
// Exchange the authorization code for an access token
$token = $provider->getAccessToken(
    'authorization_code',
    [
        'code' => $_GET['code'],
    ],
);
```

```php
// Use the token to fetch the user's profile
$user = $provider->getResourceOwner($token);

echo 'Hello, ' . $user->getName() . "!\n";
echo 'GitHub: '  . $user->getUrl() . "\n";
```

What the library did

The **library** handled:

Task	Step
Building the authorization URL	1
Generating a state value	1
POSTing to the token endpoint	4
Parsing the token response	5
Sending the `Authorization` header	6

You handled:
config, verifying `state`, user data

ELEVATE / Pexels

OAuth 2.0 handles authorization.

But *who* authorized it?

OAuth 2.0 grants access to resources.

It says nothing about **who** you are.

Every provider does it differently.

Mikhail Nilov / Pexels

OpenID Connect (OIDC)

A thin identity layer on top of OAuth 2.0, standardized by OpenID Foundation

Four additions on top of the OAuth 2.0 Authorization Code flow:

Addition	What it is
`openid` scope	Signals to the server: “I want identity, not just access”
ID token	A JWT containing verified identity claims about the user
UserInfo endpoint	A standard endpoint to fetch additional claims
Discovery document	A standard URL where the server advertises its capabilities

accounts.google.com/.well-known/openid-configuration


```
{
 "issuer": "https://accounts.google.com",
 "authorization_endpoint": "https://accounts.google.com/o/oauth2/v2/auth",
 "device_authorization_endpoint": "https://oauth2.googleapis.com/device/code",
 "token_endpoint": "https://oauth2.googleapis.com/token",
 "userinfo_endpoint": "https://openidconnect.googleapis.com/v1/userinfo",
 "revocation_endpoint": "https://oauth2.googleapis.com/revoke",
 "jwks_uri": "https://www.googleapis.com/oauth2/v3/certs",
 "response_types_supported": [
  "code",
  "token",
  "id_token",
  "code token",
  "code id_token",
  "token id_token",
  "code token id_token",
  "none"
 ],
 "response_modes_supported": [
  "query",
  "fragment",
  "form_post"
 ],
 "subject_types_supported": [
  "public"
 ],
```

ID token

JWT issued with access token (decoded here)


```
{
  "iss": "https://accounts.google.com",
  "sub": "1234567890",
  "aud": "your-client-id",
  "exp": 1716345600,
  "iat": 1716342000,
  "email": "ben@example.com",
  "name": "Ben Ramsey",
  "picture": "https://lh3.googleusercontent.com/photo.jpg"
}
```

ID token

JWT issued with access token (decoded here)

```
{
  "iss": "https://accounts.google.com",
  "sub": "1234567890",
  "aud": "your-client-id",
  "exp": 1716345600,
  "iat": 1716342000,
  "email": "ben@example.com",
  "name": "Ben Ramsey",
  "picture": "https://lh3.googleusercontent.com/photo.jpg"
}
```

ID token

JWT issued with access token (decoded here)


```
{
  "iss": "https://accounts.google.com",
  "sub": "1234567890",
  "aud": "your-client-id",
  "exp": 1716345600,
  "iat": 1716342000,
  "email": "ben@example.com",
  "name": "Ben Ramsey",
  "picture": "https://lh3.googleusercontent.com/photo.jpg"
}
```

ID token

JWT issued with access token (decoded here)

```
{
  "iss": "https://accounts.google.com",
  "sub": "1234567890",
  "aud": "your-client-id",
  "exp": 1716345600,
  "iat": 1716342000,
  "email": "ben@example.com",
  "name": "Ben Ramsey",
  "picture": "https://lh3.googleusercontent.com/photo.jpg"
}
```

ID token

JWT issued with access token (decoded here)


```
{
  "iss": "https://accounts.google.com",
  "sub": "1234567890",
  "aud": "your-client-id",
  "exp": 1716345600,
  "iat": 1716342000,
  "email": "ben@example.com",
  "name": "Ben Ramsey",
  "picture": "https://lh3.googleusercontent.com/photo.jpg"
}
```

ID token

JWT issued with access token (decoded here)


```
{
  "iss": "https://accounts.google.com",
  "sub": "1234567890",
  "aud": "your-client-id",
  "exp": 1716345600,
  "iat": 1716342000,
  "email": "ben@example.com",
  "name": "Ben Ramsey",
  "picture": "https://lh3.googleusercontent.com/photo.jpg"
}
```

ID token

JWT issued with access token (decoded here)

```
{
  "iss": "https://accounts.google.com",
  "sub": "1234567890",
  "aud": "your-client-id",
  "exp": 1716345600,
  "iat": 1716342000,
  "email": "ben@example.com",
  "name": "Ben Ramsey",
  "picture": "https://lh3.googleusercontent.com/photo.jpg"
}
```

Eric Prouzet / Unsplash

ID token

Two important things

1. **Always** validate it before trusting it.
2. Use `sub`, not `email`, as your user identifier.

OAuth 2.0 authorization request

```
https://accounts.google.com/o/oauth2/auth
  ?response_type=code
  &client_id=your-client-id
  &redirect_uri=https://example.com/callback
  &scope=email%20profile
  &state=xK9mQ2
```

OAuth 2.0 + OpenID Connect

```
https://accounts.google.com/o/oauth2/auth
  ?response_type=code
  &client_id=your-client-id
  &redirect_uri=https://example.com/callback
  &scope=openid%20email%20profile
  &state=xK9mQ2
```

OAuth 2.0 + OpenID Connect

```
https://accounts.google.com/o/oauth2/auth
  ?response_type=code
  &client_id=your-client-id
  &redirect_uri=https://example.com/callback
  &scope=openid%20email%20profile
  &state=xK9mQ2
```

Sign in with...

Google. Apple. GitHub.

These are all OAuth 2.0 + OpenID Connect.

OIDC is the standard that makes them work the same way everywhere:

- Same `openid` scope trigger
- Same ID token format (JWT)
- Same claims (`sub`, `email`, `name`, `picture`, etc.)
- Same validation steps

Enterprise Single Sign-On (SSO) is built on OIDC.

Putting it all together

El Jundi / Pexels

OAuth 2.0

“I authorize this app to act on my behalf.”

OpenID Connect

“I authorize this app to act on my behalf, and here’s who I am.”

When to use which

Scenario	Use
Call a third-party API (GitHub, Stripe, Slack) on behalf of a user	OAuth 2.0 *(i.e., Authorization Code + PKCE)*
Build login or SSO	OpenID Connect *(which is OAuth + identity)*
Service-to-service API calls with no user	Client Credentials flow
SPA or mobile app authenticating a user	Authorization Code + PKCE

When in doubt: if a human is logging in, use **OIDC**.

Don’t reinvent the wheel

OAuth 2.0 client (PHP)	• league/oauth2-client • league/oauth2-github • league/oauth2-google, etc.
ID token validation (PHP)	• facile-it/php-openid-client • firebase/php-jwt • hybridauth/hybridauth • jumbojett/openid-connect-php
Running your own authorization server	• league/oauth2-server (PHP, self-hosted) • Keycloak (open source, self-hosted) • Auth0 / Okta (managed, enterprise-grade)

Common gotchas

Gotcha	What to do
Skipping `state` validation	Always verify `state` matches what you stored; it's your CSRF protection
Trusting the ID token without validating it	Always verify signature, `iss`, `aud`, and `exp` before reading claims
Storing `sub` but keying on `email`	Use `sub` (+ `iss`) as your user identifier; email addresses change
Putting tokens in localStorage	Use HttpOnly cookies or server-side sessions; localStorage is XSS-accessible
Using the Implicit flow	Don't. It's deprecated. Use Authorization Code + PKCE instead.
HTTP in development	Always use HTTPS, even locally if possible; tokens in transit must be protected

Resources

Start here:

- oauth.net — approachable guides, spec links, news
- jwt.io — paste any JWT and see it decoded instantly

The specs:

- RFC 6749 — OAuth 2.0 Authorization Framework
- OpenID Connect Core 1.0 - the OIDC spec
- RFC 7636 — PKCE

Thank you!

Keep in touch

ben.ramsey.dev

phpc.social/@ramsey

github.com/ramsey

www.linkedin.com/in/benramsey

ben@ramsey.dev

A Beginner's Guide to OAuth and OpenID Connect
Copyright © 2026 Ben Ramsey

This work is licensed under Creative Commons Attribution-ShareAlike 4.0 International.
For uses not covered under this license, please contact the author.

Ramsey, Ben. "A Beginner's Guide to OAuth and OpenID Connect." Longhorn PHP Conference, 16 Oct. 2026, Holiday Inn Austin Midtown, Austin, TX.