On this page

OAuth Client Implementation

This is a guide to implementing atproto OAuth clients "The Hard Way." Optimistically, most developers will have an SDK available for their programming language which supports OAuth, and they can simply refer to SDK documentation. This guide is intended for early adopters, SDK maintainers, or developers with more sophisticated OAuth needs. It is agnostic to whether developers are building clients to work the the `app.bsky` microblogging Lexicons, or implementing novel application Lexicons.

The atproto OAuth specification is the authoritative document on how to use OAuth with atproto. If there are discrepancies between this document and the specification, defer to the specification. This document skips over a few details and uses more atproto-specific terminology, but readers are still expected to be familiar with OAuth 2 concepts, terminology, and standards.

This guide is focused on apps which use OAuth to make authorized ("authz") requests to user PDS servers, for example to write records to atproto repositories, or make proxied API requests to other services. It is also possible to use the atproto identity system only for authentication ("authn"), similar to OpenID/OIDC.

Types of Clients

This guide covers three simplified types of OAuth client:

- **Web Services**: traditional web apps which involve a server/backend running code to make actual PDS requests and talk with a database. There is some form of auth between browsers and the web service, such as cookie sessions; this auth layer is distinct from OAuth. The server may return complete HTML pages, or there may be an API between code running in the browser and code running on the server. Integrations with and extensions of existing web services also fall under this category.

- **Browser Apps**: "single-page" applications which run in a web browser, implemented using web platform APIs and JavaScript or WASM runtimes. The server-side ("backend") component is minimal, or even just static file hosting.
- **Mobile and Desktop Apps**: what they sound like: "native" apps that run on mobile operating systems (smartphones, tablets, etc), or desktop applications

	Web Service	Browser App	Mobile or Desktop App
OAuth 2 Client Type	"Confidential"	"Public"	"Public"
`client_id`	▨ URL to metadata	▨ URL to metadata	▨ URL to metadata
`client_secret`	×	×	×
OAuth 2 Grant Types	`authorization_code`, `refresh_token`	`authorization_code`, `refresh_token`	`authorization_code`, `refresh_token`
Client Metadata	▨ Public Web	▨ Public Web	▨ Public Web
Client Metadata JWK	▨ Public Web	×	×
PKCE	▨	▨	▨
PAR	▨	▨	▨
DPoP	▨	▨	▨
Handle Resolution	DNS and HTTPS	DNS-over-HTTPS and HTTPS or via helper service	DNS and HTTPS or via helper service
DID Resolution	HTTPS	HTTPS	HTTPS
Recommended Client Secret	Environment Variable, Secrets	×	×

Key Storage	Manager, Hardware Enclave		
Recommended DPoP Key Storage	Secure Database	non-exportable CryptoKeyPair in IndexedDB	Secure File or Database, Hardware Enclave
Recommended Token Storage	Secure Database	IndexedDB or LocalStorage	Secure File or Database
SSRF + DoS Hardening	▨	▨	▨
Authorization UI	Browser Redirect	Browser Redirect	WebView/Browser
`redirect_uri`	App URL (HTTPS)	App URL (HTTPS)	App Link (Android), Universal Link (iOS), or Client-specific URI scheme

▨: Required

×: Forbidden

PKCE: Proof Key for Code Exchange (RFC 7636)

PAR: Pushed Authorization Requests (RFC 9126)

DPoP: Demonstrating Proof of Possession (RFC 9449)

Client Metadata: OAuth Client ID Metadata Document (`draft-parecki-oauth-client-id-metadata-document`)

Other architectures are possible. For example, a mobile app which uses a web service to mediate client authentication and refresh tokens, or a web service could act as a "Public" client. This guide focuses on the most common use-cases.

OAuth is not currently recommended as an auth solution for for "headless" clients, such as command-line tools or bots.

Components

OAuth 2 is a framework for designing authentication systems, not a single standard or API to implement. This section describes the standards used in the specific atproto "profile" of OAuth, and components that a typical client will need to implement.

Client and Server Metadata

The atproto network is decentralized: there are many independent PDS instances and many client apps. OAuth needs to facilitate any client app being authorized against any PDS instance, without prior registration or coordination between users, developers, or service operators. The atproto OAuth profile makes this possible by combining public client metadata and public authorization server metadata.

All atproto OAuth clients must publish a client metadata JSON document on the public web. The `client_id`, which globally identifies the client software instance, is the fully-qualified `https://` URL at which this JSON document can be accessed.

"Confidential" clients (Web Services) include public cryptographic keys in their client metadata which can be used during an authentication request to verify the client. It is important that such clients be able to remove the public keys from their client metadata in the event that the corresponding secret key is compromised or leaked.

Client metadata fields include:

- `client_id` (string, required): must exactly match the full URL used to fetch the client metadata JSON itself
- `application_type` (string, optional): must be one of `web` (default) or `native`
- `grant_types` (array of strings, required): usually `authorization_code` and `refresh_token`
- `scope` (string, sub-strings space-separated, required): any scope values which *might* be requested by this client are declared here. The `atproto` scope is required.
- `response_types` (array of strings, required): usually just `code`.
- `redirect_uris` (array of strings, required): the fully-qualified redirect/callback URL is declared here.

- `dpop_bound_access_tokens` (boolean, required): must be `true` (DPoP is mandatory)
- `token_endpoint_auth_method` (string, optional): confidential clients must set this to `private_key_jwt`.
- `token_endpoint_auth_signing_alg` (string, optional): confidential client set this to `ES256`
- `jwks` (object with array of JWKs, optional) or `jwks_uri` (string URL, optional): confidential clients must supply at least one public key in JWK format for use with JWT client authentication.

And some optional (but recommended) metadata fields:

- `client_name` (string, optional): human-readable name of the client
- `client_uri` (string, optional): not to be confused with `client_id`, this is a homepage URL for the client. If provided, the `client_uri` must have the same hostname as `client_id`.
- `logo_uri` (string, optional): HTTP URL to client logo
- `tos_uri` (string, optional): HTTP URL to human-readable terms of service ("ToS") for the client
- `policy_uri` (string, optional): HTTP URL to human-readable privacy policy for the client

Here is an example Browser App client metadata file, that would need to be hosted at https://app.example.com/oauth/client-metadata.json (served with Content-Type `application/json` and HTTP status 200, no redirects):

```
{
  "client_id": "https://app.example.com/oauth/client-metadata.json",
  "application_type": "web",
  "client_name": "Example Browser App",
  "client_uri": "https://app.example.com",
  "dpop_bound_access_tokens": true,
  "grant_types": ["authorization_code", "refresh_token"],
  "redirect_uris": ["https://app.example.com/oauth/callback"],
  "response_types": ["code"],
  "scope": "atproto transition:generic",
  "token_endpoint_auth_method": "none"
}
```

PDS instances (and any supporting servers) also publish public JSON documents containing authorization server metadata.

In OAuth terminology, the PDS is a "Resource Server" which authenticated requests are made to. The PDS publishes a "protected resource metadata" file at the well-known HTTPS path `/.well-known/oauth-protected-resource`. This contains a field `authorization_servers` with an array of URLs indicating the "Authorization Server" location (the origin or "issuer"). In OAuth terminology, the "Authorization Server" is responsible for authenticating the user and providing authorization tokens. The authorization server might be the PDS itself (same origin), or it might be separate. For example, an "entryway" service in large multi-PDS deployments, or an delegated authorization provider. The authorization server metadata endpoint is `/.well-known/oauth-authorization-server`. The response includes the following fields relevant to clients:

- `issuer` (string, required): the "origin" URL of the Authorization Server. Must be a valid URL, with `https` scheme, matching the origin of URL used to fetch this document. There must be no path segments.
- `pushed_authorization_request_endpoint` (string, required): URL for Pushed Authentication Requests (PAR)
- `authorization_endpoint` (string, required): URL for authorization interface
- `token_endpoint` (string, required): URL for token requests
- `scopes_supported` (space-separated string, required): must include `atproto`, to confirm that this server supports the atproto profile of OAuth. If supporting the transitional grants, they should be included here as well.

There is a longer list of fields that clients may want to confirm/validate in the atproto OAuth specification.

Fetches of any of these metadata documents should be made using a hardened HTTP client, as described below.

PKCE

All clients must implement PKCE. In practical terms, this means:

- creating a unique random value at the start of the session
- including a "challenge code" derived from this value during the Authentication Request
- verifying the value during the first token request

The "code challenge" method used is `S256`, which is the most popular PKCE challenge method. The transform involves a relatively simple SHA-256 hash and base64url string encoding. It can be implemented from scratch if needed, or

sometimes OAuth libraries provide a helpers. The code value is a set of 32 to 96 random bytes, encoded in base64url (resulting in 43 or more string-encoded characters).

For example, given a randomly generated "verifier" token, whose base64url representation is: `dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk`

The `S256` code challenge is: `E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM`

PAR

Pushed Authentication Requests (PAR) are required for all client types. This means that the client makes an HTTP POST request to the PDS/entryway PAR endpoint with all the authentication requests parameters as an HTTP form-encoded body, and receives a request token in response. The client then redirects the browser to the authorization endpoint with the token (and `client_id`) as a query parameter, instead of passing a long list of request fields as query parameters.

The PAR request is submitted to the `pushed_authorization_request_endpoint` (from server metadata), and must use `Content-Type: application/x-www-form-urlencoded`.

A successful response body will be a JSON object including the field `request_uri` (not to be confused with `redirect_uri`).

DPoP

Clients must use DPoP to bind auth tokens to a specific client device or server. DPoP nonces, provided by the auth server, must be used.

Clients generate a new DPoP cryptographic keypair *for each auth session*, and retain the keypair for the duration of the auth session. DPoP keypairs should never be exported or moved between devices, and should never be reused across users or between sessions for the same user. Client must start DPoP at the initial authorization request (PAR).

`ES256` (NIST "P-256") is the cryptographic algorithm/curve which must be supported by all clients and auth servers. Browser Apps should use the WebCrypto API to generate non-exportable keypairs, which can be stored in IndexedDB to persist across browser sessions (not to be confused with OAuth sessions). Other clients may find implementations of this cryptographic system in generic JWT libraries, or in generic cryptographic libraries for their language or environment. DPoP is

also increasingly required as part of OAuth profiles and will hopefully be supported by generic OAuth libraries.

DPoP involves setting a HTTP Header (`DPoP`) on every token request and every authorized request to the PDS. The header value is a self-signed JWT. There is a unique random field (`jti`) in the body, and JWTs are generated and signed uniquely for every request (DPoP proof JWTs can not be reused between requests).

The server returns the current DPoP nonce in the `DPoP-Nonce` HTTP header in every response. Nonce values may be shared across all users and sessions on the server, or may be scoped to individual users and sessions. Nonces may be shared between access token use (PDS requests) and authorization server requests (PDS or entryway), but they may be distinct servers, so clients should always track DPoP nonces separately for the two uses. Nonces change periodically, with a rotation period chosen by the server. Clients should persist the DPoP nonce for each session, and update the persisted value when a response is received with a different value.

If the nonce is missing (because it isn't known yet), or has become outdated, the server will return an HTTP 401 ("Unauthorized") response, indicating the error type as `use_dpop_nonce` and including the current nonce value in the `DPoP-Nonce` header. The Authorization Server (entryway or PDS, when doing PAR or token requests) indicates the error type with a JSON object body with the `error` field set to `use_dpop_nonce`. The Resource Server (PDS, when making authorized requests) indicates the error type using the `WWW-Authenticate` header with an `error` value set to `use_dpop_nonce`. For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP error="use_dpop_nonce", error_description="Resource serve
DPoP-Nonce: eyJ7S_zG.eyJH0-Z.HX4w-7v
```

For other server type, the client can retry the request with a new DPoP proof JWT including the nonce value. The client discovers the initial nonce for each server by doing this request/error/retry cycle at least once. Servers will usually accept stale/old nonce values for a short time window to reduce errors-and-retries caused by clients making multiple concurrent authorized requests. Ideally the request/error/retry cycle does not need to happen again, though clients should be ready for it at any time (eg, if the nonce has rotated multiple times between requests).

When making DPoP requests to token endpoint:

- JWT header fields must be:
  - `typ`: `dpop+jwt`
  - `alg`: `ES256`
  - `jwk`: DPoP public key in JSON Web Key (JWK) string format
- JWT fields should include:
  - `jti`: random token string (unique per request)
  - `htm`: HTTP method (eg, "POST" or "GET")
  - `htu`: HTTP request URL
  - `iat`: current UNIX time (integer seconds)
  - `exp`: optional, expiration UNIX time (integer seconds) in the near future
  - `nonce`: server-provided nonce string. If nonce isn't known yet, don't include this field, then receive the nonce via header in the error response
- JWT string in the `DPoP` HTTP header

When making DPoP requests to PDS endpoints:

- same JWT header fields as above
- same JWT body fields as above, plus:
  - `ath`: hash of the access token, using the same mechanism as the `S256` PKCE challenge hash
- JWT string in the `DPoP` HTTP header
- access token in the `Authorization` HTTP header, with type `DPoP` (so header looks like `Authorization: DPoP <token>`)

> **NOTE**
>
> An earlier version of this guide indicated that `iss` should be included in the JWT body for DPoP requests to PDS endpoints, with the value being the authorization server issuer. Currently clients SHOULD NOT include this value, or any value, in the `iss` field for these JWTs.

Confidential Client Authentication

Confidential clients declare their authentication keypair by including the public key in their client metadata (in the `jwks` or `jwks_uri` fields), and then authenticate by including a JWT bearer assertion in requests to the authorization server. They are required for the authorization request (PAR) and all token requests (both the initial token request and any subsequent refresh requests).

The client assertion type is `urn:ietf:params:oauth:client-assertion-type:jwt-bearer`.

When constructing an assertion JWT:

- assertion JWTs must not be reused: they include a random token, and must be generated and signed on every relevant request
- the `ES256` cryptographic system must be supported by all clients and auth servers
- JWT header fields include:
  - `alg`: `ES256`
  - `kid`: string indicating which of the declared keys (in JWKS) is being used
- JWT body fields include:
  - `iss`: the `client_id`
  - `sub`: the `client_id`
  - `aud`: the authorization server issuer URL (origin)
  - `jti`: randomly generated token string
  - `iat`: current UNIX time (integer seconds)

Token Management

Long-lived clients will need to manage access token lifetimes and periodic refresh token requests. This is functionality that is sometimes implemented in generic OAuth libraries. Care must be taken to ensure that concurrent resource requests don't result in concurrent token refresh requests, which could result in errors and loss of the overall auth session (requiring re-authorization).

Access and refresh tokens should never be copied or shared across end devices. They should not be stored in session cookies.

Hardened HTTP Client (SSRF)

Clients need to make a several HTTP network requests using URLs provided by unknown parties. These raise a number of security concerns, including network requests to local or private IP addresses (SSRF), and trivial Denial of Service attacks (large response bodies, slow responses, etc).

A good way to mitigate these issues is to use or implement a hardened HTTP client. It should set appropriate timeouts and resource limits, validate URLs,

and check that resolve domain names don't point to protected or local IP addresses.

Authorization Flow

Account or Server Identifier

Clients can start an auth flow in one of two ways:

- starting with an atproto account identifier: handle or DID
- starting with a server URL or hostname (PDS or entryway)

One use case for starting with a server URL instead of an account identifier is when the user does not remember their full account handle or only knows their account email. Another is for authentication when a user's handle is broken. The user still needs to know their hosting provider in these situations.

When starting with an account identifier, the client must resolve the atproto identity to a DID document. If starting with a handle, it is critical (mandatory) to bidirectionally verify the handle by checking that the DID document claims the handle (see atproto Handle specification). All handle resolution techniques and all atproto-blessed DID methods must be supported to ensure interoperability with all accounts.

In some client environments, it may be difficult to resolve all identity types. For example, handle resolution may involve DNS TXT queries, which are not directly supported from browser apps. Client implementations might use alternative techniques (such as DNS-over-HTTPS) or could make use of a supporting web service to resolve identities.

If the auth flow instead starts with a server (hostname or URL), the client will first attempt to fetch Resource Server metadata (and resolve to Authorization Server if found) and then attempt to fetch Authorization Server metadata. If either is successful, the client will end up with an identified Authorization Server. The Authorization Request and flow will proceed without a `login_hint` or account identifier being bound to the session, but the Authorization Server `issuer` will be bound to the session.

Either way, by the end of the authorization flow it will be important to resolve the DID of the authorized account and verify that it is consistent with the

authorization server being talked to, and that the server granted access tokens for the expected account.

Authorization Request

The client next makes a Pushed Authorization Request via HTTP POST request to the `pushed_authorization_request_endpoint`. Notable details include:

- a randomly generated `state` token is required, and will be used to reference this authorization request with the subsequent response callback
- PKCE is required, so a secret value is generated and stored, and a derived challenge is included in the request
- `scopes` value is included here, and must include `atproto`
- for confidential clients, a `client_assertion` is included, with type `jwt-bearer`, signed using the secret client authentication key
- the client generates a new DPoP key for the user/session and uses it starting with the PAR request
- if the auth flow started with an account identifier, the client should pass that starting identifier via the `login_hint` field

The initial response will be a DPoP error, with the server nonce included in an HTTP header. The client includes this nonce in a new DPoP JWT and retries the request.

The Authorization Server will receive the PAR request and use the `client_id` URL to resolve the client metadata document. If all goes well, the server returns a `request_uri` token to the client.

The client persists information about the session to some form of storage. This might be a database (for a web service backend) or web platform storage like IndexedDB (for a browser app).

Then the client redirects the user via browser to the Authorization Server's auth endpoint, including the `request_uri` as a URL parameter.

The user will authenticate with the server and approve the authorization request, using the "authorization interface" on the PDS/entryway.

Callback and Access Token Request

The server redirects the user back to the `redirect_uri` (from the authorization interface), with some query parameters included:

- `state`: matching `state` included in the authorization request
- `iss`: the URL (origin) of the authorization server
- `code`: the authorization code which can be used for an initial token request

The client can now make an initial token request to the authorization server token endpoint. It includes the `code` and PKCE code verification. Confidential clients must also include a client assertion (JWT signed with the client keypair).

This request uses DPoP, using the authorization server nonce saved after the earlier authorization request.

The server will return a JSON object with a set of tokens (`access_token` and `refresh_token`). It will also include a `sub` field containing the atproto account DID, and authorized `scope`.

It is **critical** for the client to verify that the `sub` DID matches the account expected. If the entire auth flow started with an account identifier (handle or DID), this value is compared against the original DID. If the auth flow started with a PDS/entryway URL, the client should now resolve the DID document, and verify that the declared PDS instances is consistent with the authorization server.

Authentication-only clients can end the flow here.

PDS Requests and Token Refresh

Using the access token, clients are now able to make authorized requests to the PDS. They must use DPoP for all such requests, with a separate server-provided nonce, along with the access token.

Tokens (both access and refresh) will need to be periodically "refreshed" by subsequent request to the Authorization Server token endpoint.

« Action Intent Links

API Hosts and Auth »