



IndieAuth

W3C Editor's Draft 05 January 2018

This version:

<https://indieauth.spec.indieweb.org/>

Latest published version:

<https://www.w3.org/TR/indieauth/>

Latest editor's draft:

<https://indieauth.spec.indieweb.org/>

Test suite:

<https://indieauth.rocks/>

Editor:

[Aaron Parecki](#)

Repository:

[Github](#)

[Issues](#)

[Commits](#)

Please check the **errata** for any errors or issues reported since publication.

Copyright © 2018 W3C[®] (MIT, ERCIM, Keio, Beihang). W3C liability, trademark and permissive document license rules apply.

Abstract

IndieAuth is an identity layer on top of OAuth 2.0 [[RFC6749](#)], primarily used to obtain an OAuth 2.0 Bearer Token [[RFC6750](#)] for use by [[Micropub](#)] clients. End-Users and Clients are all represented by URLs. IndieAuth enables Clients to verify the identity of an End-User, as well as to obtain an access token that can be used to access resources under the control of the End-User.

Author's Note

This section is non-normative.

This specification was contributed to the W3C from the [IndieWeb](#) community. More history and evolution of IndieAuth can be found on the [IndieWeb wiki](#).

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the [W3C technical reports index](https://www.w3.org/TR/) at <https://www.w3.org/TR/>.

This document was published by the [Social Web Working Group](#) as an Editor's Draft. Comments regarding this document are welcome. Please send them to public-socialweb@w3.org ([subscribe](#), [archives](#)).

Publication as an Editor's Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the [W3C Patent Policy](#). W3C maintains a [public list of any patent disclosures](#) made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains [Essential Claim\(s\)](#) must disclose the information in accordance with [section 6 of the W3C Patent Policy](#).

This document is governed by the [1 March 2017 W3C Process Document](#).

Table of Contents

1.	Introduction
1.1	Background
1.2	OAuth 2.0 Extension
2.	Conformance

- 2.1 Conformance Classes
 - 2.1.1 Authorization Endpoint
 - 2.1.2 Token Endpoint
 - 2.1.3 Micropub Client
 - 2.1.4 IndieAuth Client
- 3. Identifiers**
 - 3.1 User Profile URL
 - 3.2 Client Identifier
 - 3.3 URL Canonicalization
- 4. Discovery**
 - 4.1 Discovery by Clients
 - 4.1.1 Redirect Examples
 - 4.2 Client Information Discovery
 - 4.2.1 Application Information
 - 4.2.2 Redirect URL
- 5. Authentication**
 - 5.1 Discovery
 - 5.2 Authentication Request
 - 5.3 Authentication Response
 - 5.4 Authorization Code Verification
- 6. Authorization**
 - 6.1 Discovery
 - 6.2 Authorization Endpoint
 - 6.2.1 Authorization Request
 - 6.2.2 Authorization Response
 - 6.3 Token Endpoint
 - 6.3.1 Token Request
 - 6.3.2 Authorization Code Verification
 - 6.3.3 Access Token Response
 - 6.3.4 Access Token Verification
 - 6.3.5 Token Revocation

- 7. Security Considerations**
 - 7.1 Differing User Profile URLs
 - 7.2 Preventing Phishing and Redirect Attacks
- 8. IANA Considerations**
- A. Resources**
 - A.1 Articles
 - A.2 Implementations
- B. Acknowledgements**
- C. References**
 - C.1 Normative references
 - C.2 Informative references

1. Introduction

1.1 Background

This section is non-normative.

The IndieAuth spec began as a way to obtain an OAuth 2.0 access token for use by Micropub clients. It can be used to both obtain an access token, as well as authenticate users signing to any application. It is built on top of the OAuth 2.0 framework, and while this document should provide enough guidance for implementers, referring to the core OAuth 2.0 spec can help answer any remaining questions. More information can be found [on the IndieWeb wiki](#).

1.2 OAuth 2.0 Extension

IndieAuth builds upon the OAuth 2.0 [\[RFC6749\]](#) Framework as follows

- Specifies a format for user identifiers (a resolvable URL)
- Specifies a method of discovering the authorization and token endpoints given a profile URL

- Specifies a format for the Client ID (a resolvable URL)
- All clients are public clients (no `client_secret` is used)
- Client registration at the authorization endpoint is not necessary, since client IDs are resolvable URLs
- Redirect URL registration happens by verifying data fetched at the Client ID URL
- Specifies a mechanism for returning the user identifier for the user who authorized a request
- Specifies a mechanism for verifying authorization codes
- Specifies a mechanism for a token endpoint and authorization endpoint to communicate

Additionally, the parameters defined by OAuth 2.0 (in particular `state`, `code`, and `scope`) follow the same syntax requirements as defined by Appendix A of OAuth 2.0 [[RFC6749](#)].

2. Conformance

The key words "*MUST*", "*MUST NOT*", "*REQUIRED*", "*SHALL*", "*SHALL NOT*", "*SHOULD*", "*SHOULD NOT*", "*RECOMMENDED*", "*MAY*", and "*OPTIONAL*" in this document are to be interpreted as described in [[RFC2119](#)].

2.1 Conformance Classes

An IndieAuth implementation can implement one or more of the roles of an IndieAuth server or client. This section describes the conformance criteria for each role.

Listed below are known types of IndieAuth implementations.

2.1.1 Authorization Endpoint

An IndieAuth Authorization Endpoint is responsible for obtaining authorization or authentication consent from the end user and generating and verifying authorization codes.

2.1.2 Token Endpoint

An IndieAuth Token Endpoint is responsible for generating and verifying OAuth 2.0 Bearer Tokens.

2.1.3 Micropub Client

A Micropub client will attempt to obtain an OAuth 2.0 Bearer Token given an IndieAuth profile URL, and will use the token when making Micropub requests.

2.1.4 IndieAuth Client

An IndieAuth client is a client that is attempting to authenticate a user given their profile URL, but does not need an OAuth 2.0 Bearer Token.

3. Identifiers

3.1 User Profile URL

Users are identified by a [\[URL\]](#). Profile URLs *MUST* have either an **https** or **http** scheme, *MUST* contain a path component (**/** is a valid path), *MUST NOT* contain single-dot or double-dot path segments, *MAY* contain a query string component, *MUST NOT* contain a fragment component, *MUST NOT* contain a username or password component, and *MUST NOT* contain a port. Additionally, hostnames *MUST* be domain names and *MUST NOT* be ipv4 or ipv6 addresses.

Some examples of valid profile URLs are:

- **https://example.com/**
- **https://example.com/username**
- **https://example.com/users?id=100**

Some examples of invalid profile URLs are:

- **example.com** - missing scheme
- **mailto:user@example.com** - invalid scheme

- `https://example.com/foo/./bar` - contains a double-dot path segment
- `https://example.com/#me` - contains a fragment
- `https://user:pass@example.com/` - contains a username and password
- `https://example.com:8443/` - contains a port
- `https://172.28.92.51/` - host is an IP address

3.2 Client Identifier

Clients are identified by a [\[URL\]](#). Client identifier URLs *MUST* have either an `https` or `http` scheme, *MUST* contain a path component, *MUST NOT* contain single-dot or double-dot path segments, *MAY* contain a query string component, *MUST NOT* contain a fragment component, *MUST NOT* contain a username or password component, and *MAY* contain a port. Additionally, hostnames *MUST* be domain names or a loopback interface and *MUST NOT* be IPv4 or IPv6 addresses except for IPv4 `127.0.0.1` or IPv6 `:::1`.

3.3 URL Canonicalization

Since IndieAuth uses https/http URLs which fall under what [\[URL\]](#) calls "[Special URLs](#)", a string with no path component is not a valid [\[URL\]](#). As such, if a URL with no path component is ever encountered, it *MUST* be treated as if it had the path `/`. For example, if a user enters `https://example.com` as their profile URL, the client *MUST* transform it to `https://example.com/` when using it and comparing it.

Since domain names are case insensitive, the hostname component of the URL *MUST* be compared case insensitively. Implementations *SHOULD* convert the hostname to lowercase when storing and using URLs.

For ease of use, clients *MAY* allow users to enter just a hostname part of the URL, in which case the client *MUST* turn that into a valid URL before beginning the IndieAuth flow, by prepending a scheme and appending the path `/`. For example, if the user enters `example.com`, the client transforms it into `http://example.com/` before beginning discovery.

4. Discovery

This specification uses the link rel registry as defined by [\[HTML\]](#) for both HTML and HTTP link relations.

4.1 Discovery by Clients

Clients need to discover a few pieces of information when a user signs in. For the [Authentication](#) workflow, the client needs to find the user's `authorization_endpoint`. For the [Authorization](#) workflow, the client needs to find the user's `authorization_endpoint` and `token_endpoint`. When using the Authorization workflow to obtain an access token for use at a [\[Micropub\]](#) endpoint, the client will also discover the `micropub` endpoint.

Clients *MUST* start by making a GET or HEAD request to [\[Fetch\]](#) the user's profile URL to discover the necessary values. Clients *MUST* follow HTTP redirects (up to a self-imposed limit). If an HTTP permanent redirect (HTTP 301 or 308) is encountered, the client *MUST* use the resulting URL as the canonical profile URL. If an HTTP temporary redirect (HTTP 302 or 307) is encountered, the client *MUST* use the previous URL as the profile URL, but use the redirected-to page for discovery.

Clients *MUST* check for an HTTP `Link` header [\[RFC5988\]](#) with the appropriate `rel` value. If the content type of the document is HTML, then the client *MUST* check for an HTML `<link>` element with the appropriate `rel` value. If more than one of these is present, the first HTTP `Link` header takes precedence, followed by the first `<link>` element in document order.

The endpoints discovered *MAY* be relative URLs, in which case the client *MUST* resolve them relative to the profile URL according to [\[URL\]](#).

Clients *MAY* initially make an HTTP HEAD request [\[RFC7231\]](#) to follow redirects and check for the `Link` header before making a GET request.

4.1.1 Redirect Examples

4.1.1.1 http to https

In this example, the user enters `example.com` in the sign-in form, so the client initially transforms that to `http://example.com/` to perform discovery. The URL `http://example.com/` returns an HTTP 301 permanent redirect to `https://example.com/`, so the client updates the initial profile URL to `https://example.com/`, and looks at the contents of that page to find the authorization endpoint.

4.1.1.2 www to no-www

In this example, the user enters `www.example.com` in the sign-in form, so the client initially transforms that to `http://www.example.com/` to perform discovery. The URL `http://www.example.com/` returns an HTTP 301 permanent redirect to `https://example.com/`, so the client updates the initial profile URL to `https://example.com/`, and looks at the contents of that page to find the authorization endpoint.

4.1.1.3 Temporary Redirect

In this example, the user enters `example.com` in the sign-in form, so the client initially transforms that to `http://example.com/` to perform discovery. The URL `http://example.com/` returns an HTTP 301 permanent redirect to `https://example.com/`, and `https://example.com/` returns an HTTP 302 temporary redirect to `https://example.com/username`. The client stores the last 301 permanent redirect as the profile URL, `https://example.com/`, and uses the contents of `https://example.com/username` to find the authorization endpoint.

4.1.1.4 Permanent Redirect to a Different Domain

In this example, the user enters `username.example` in the sign-in form, so the client initially transforms that to `http://username.example/` to perform discovery. However, the user does not host any content there, and instead that page is a redirect to their profile elsewhere. The URL `http://username.example/` returns an HTTP 301 permanent redirect to `https://example.com/username`, so the client updates the initial profile URL to `https://example.com/username` when setting the `me` parameter in the initial authorization request. At the end of the flow, the authorization endpoint will return a `me` value of

<https://example.com/username>, which is not on the same domain as what the user entered, but the client can accept it because of the HTTP 301 redirect encountered during discovery.

4.1.1.5 Temporary Redirect to a Different Domain

In this example, the user enters username.example in the sign-in form, so the client initially transforms that to <http://username.example/> to perform discovery. However, the user does not host any content there, and instead that page is a temporary redirect to their profile elsewhere. The URL <http://username.example/> returns an HTTP 302 temporary redirect to <https://example.com/username>, so the client discovers the authorization endpoint at that URL. Since the redirect is temporary, the client still uses the user-entered <http://username.example/> when setting the `me` parameter in the initial authorization request. At the end of the flow, the authorization endpoint will return a `me` value of <https://username.example/>, which is not on the same domain as the authorization endpoint, but is the same domain as the user entered. This allows users to still use a profile URL under their control while delegating the authorization flow to an external account.

4.2 Client Information Discovery

When an authorization server presents its [authorization interface](#), it will often want to display some additional information about the client beyond just the `client_id` URL, in order to better inform the user about the request being made. Additionally, the authorization server needs to know the list of redirect URLs that the client is allowed to redirect to.

Since client identifiers are URLs, the authorization server *SHOULD* [\[Fetch\]](#) the URL to find more information about the client.

4.2.1 Application Information

Clients *SHOULD* have a web page at their `client_id` URL with basic information about the application, at least the application's name and icon. This page serves as a good landing page for human visitors, but can also serve as the place to include machine-readable information about the application. The HTML on the `client_id` URL *SHOULD* be marked up with [\[h-app\]](#)

[[Microformats](#)] to indicate the name and icon of the application. Authorization servers *SHOULD* support parsing the [[h-app](#)] Microformat from the `client_id`, and if there is an [[h-app](#)] with a `url` property matching the `client_id` URL, then it should use the name and icon and display them on the authorization prompt.

EXAMPLE 1

```
<div class="h-app">
  
  <a href="/" class="u-url p-name">Example App</a>
</div>
```

This can be parsed with a [Microformats2 parser](#), which will result in the following JSON structure.

EXAMPLE 2

```
{
  "type": [
    "h-app"
  ],
  "properties": {
    "name": ["Example App"],
    "logo": ["https://app.example.com/logo.png"],
    "url": ["https://app.example.com/"],
    "photo": ["http://app.example.com/logo.png"]
  }
}
```

4.2.2 Redirect URL

If a client wishes to use a redirect URL that is on a different domain than their `client_id`, or if the redirect URL uses a custom scheme (such as when the client is a native application), then the client will need to whitelist those redirect URLs so that authorization endpoints can be sure it is safe to

redirect users there. The client *SHOULD* publish one or more `<link>` tags or `Link` HTTP headers with a `rel` attribute of `redirect_uri` at the `client_id` URL.

Authorization endpoints verifying that a `redirect_uri` is allowed for use by a client *MUST* look for an exact match of the given `redirect_uri` in the request against the list of `redirect_uris` discovered after resolving any relative URLs.

EXAMPLE 3

```
GET / HTTP/1.1
Host: app.example.com

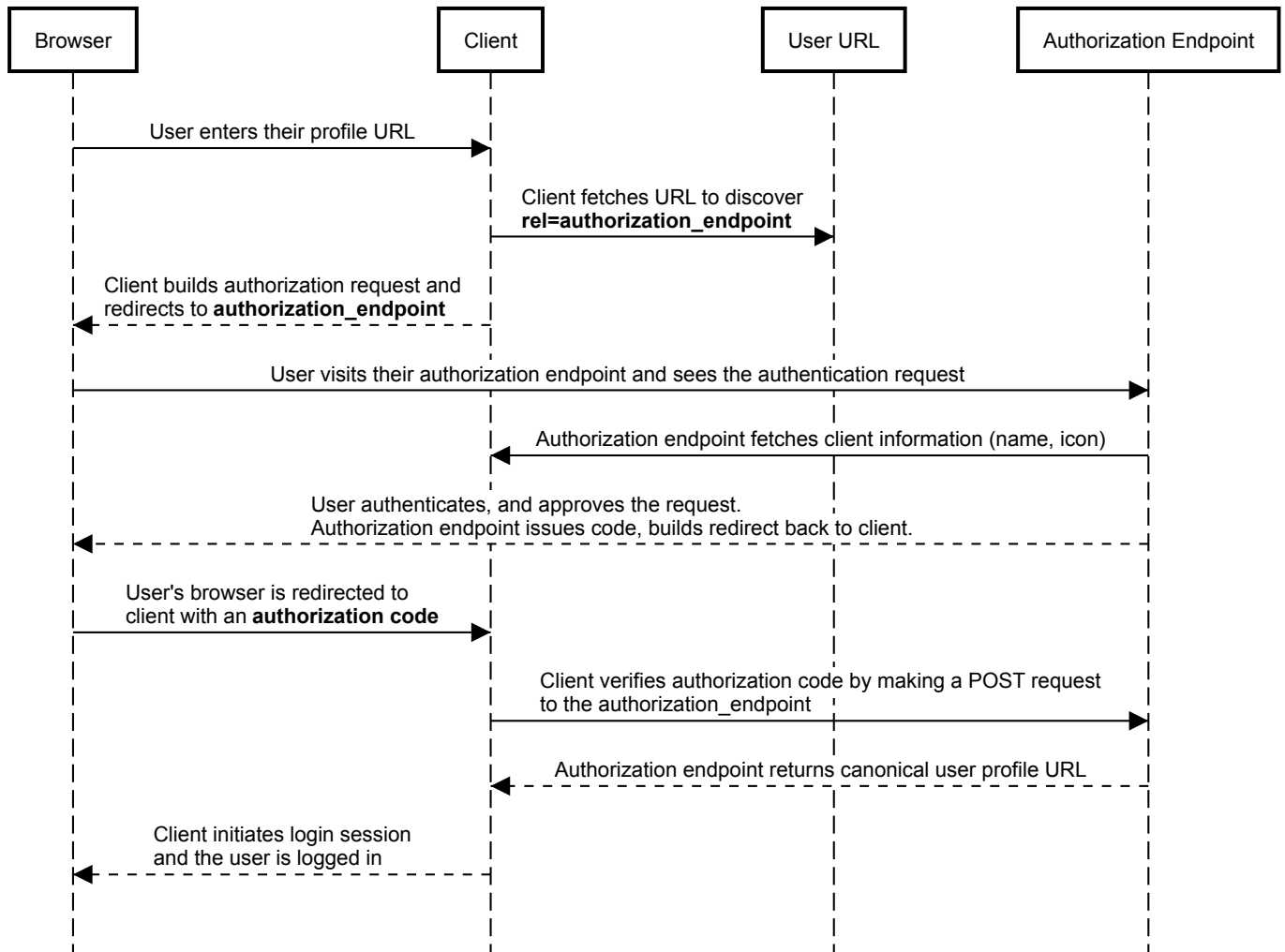
HTTP/1.1 200 Ok
Content-type: text/html; charset=utf-8
Link: <https://app.example.com/redirect>; rel="redirect_uri"

<!doctype html>
<html>
  <head>
    <link rel="redirect_uri" href="/redirect">
  </head>
  ...
</html>
```

5. Authentication

This section describes how to perform authentication using the Authorization Code Flow.

IndieAuth Authentication Flow Diagram



- The End-User enters their profile URL in the login form of the client and clicks "Sign in"
- The client discovers the End-User's authorization endpoint by fetching the End-User's profile URL and looking for the `rel=authorization_endpoint` value
- The client builds the authorization request including its client identifier, local state, and a redirect URI, and redirects the browser to the authorization endpoint
- The authorization endpoint fetches the client information from the client identifier URL in order to have an application name and icon to display to the user
- The authorization endpoint verifies the End-User, e.g. by logging in, and establishes whether the End-User grants or denies the client's authentication request
- The authorization endpoint generates an authorization code and redirects the browser back to the client, including an authorization code in the URL

- The client verifies the authorization code by making a POST request to the authorization endpoint. The authorization endpoint validates the authorization code, and responds with the End-User's canonical profile URL

5.1 Discovery

After obtaining the End-User's profile URL, the client fetches the URL and looks for the `authorization_endpoint` rel value in the HTTP `Link` headers and HTML `<link>` tags.

EXAMPLE 4

```
Link: <https://example.org/auth>; rel="authorization_endpoint"  
  
<link rel="authorization_endpoint" href="https://example.org/auth">
```

5.2 Authentication Request

The client builds the authentication request URL by starting with the discovered `authorization_endpoint` URL and adding the following parameters to the query component:

- `me` - The user's profile URL
- `client_id` - The client URL
- `redirect_uri` - The redirect URL indicating where the user should be redirected to after approving the request
- `state` - A parameter set by the client which will be included when the user is redirected back to the client. This is used to prevent CSRF attacks. The authorization server *MUST* return the unmodified state value back to the client.
- `response_type=id` - (optional) Indicates to the authorization server that this is an authentication request. If this parameter is missing, the authorization endpoint *MUST* default to `id`.

EXAMPLE 5

```
https://example.org/auth?me=https://user.example.net/&
                                redirect_uri=https://app.example.com/redirect&
                                client_id=https://app.example.com/&
                                state=1234567890&
                                response_type=id
```

The authorization endpoint *SHOULD* fetch the `client_id` URL to retrieve application information and the client's registered redirect URLs, see [Client Information Discovery](#) for more information.

If the URL scheme, host or port of the `redirect_uri` in the request do not match that of the `client_id`, then the authorization endpoint *SHOULD* verify that the requested `redirect_uri` matches one of the [redirect URLs](#) published by the client, and *SHOULD* block the request from proceeding if not.

It is up to the authorization endpoint how to authenticate the user. This step is out of scope of OAuth 2.0, and is highly dependent on the particular implementation. Some authorization servers use typical username/password authentication, and others use alternative forms of authentication such as [\[RelMeAuth\]](#), which uses [\[XFN11\]](#)'s simple `rel=me` markup.

Once the user is authenticated, the authorization endpoint presents the authentication prompt to the user. The prompt *MUST* indicate which application the user is signing in to, and *SHOULD* provide as much detail as possible about the request.

5.3 Authentication Response

If the user approves the request, the authorization endpoint generates an authorization code and builds the redirect back to the client.

The redirect is built by starting with the `redirect_uri` in the request, and adding the following parameters to the query component of the redirect URL:

- `code` - The authorization code generated by the authorization endpoint. The code *MUST* expire shortly after it is issued to mitigate the risk of leaks. A maximum lifetime of 10 minutes is

recommended. See [OAuth 2.0 Section 4.1.2](#) for additional requirements on the authorization code.

- **state** - The state parameter *MUST* be set to the exact value that the client set in the request.

EXAMPLE 6

HTTP/1.1 302 Found

Location: `https://app.example.com/redirect?code=xxxxxxx
state=1234567890`

Upon the redirect back to the client, the client *MUST* verify that the state parameter in the request is valid and matches the state parameter that it initially created, in order to prevent CSRF attacks. The state value can also store session information to enable development of clients that cannot store data themselves.

5.4 Authorization Code Verification

Request

After the state parameter is validated, the client makes a POST request to the authorization endpoint to verify the authorization code and retrieve the final user profile URL. The POST request contains the following parameters:

- **code** - The authorization code received from the authorization endpoint in the redirect
- **client_id** - The client's URL, which *MUST* match the `client_id` used in the authorization request.
- **redirect_uri** - The client's redirect URL, which *MUST* match the initial authorization request.

EXAMPLE 7

```
POST https://example.org/auth
Content-type: application/x-www-form-urlencoded

code=xxxxxxx
&client_id=https://app.example.com/
&redirect_uri=https://app.example.com/redirect
```

Response

The authorization endpoint verifies that the authorization code is valid, and that it was issued for the matching `client_id` and `redirect_uri`. If the request is valid, then the endpoint responds with a JSON [RFC7159] object containing one property, `me`, with the canonical user profile URL for the user who signed in.

EXAMPLE 8

```
HTTP/1.1 200 OK
Content-Type: application/json

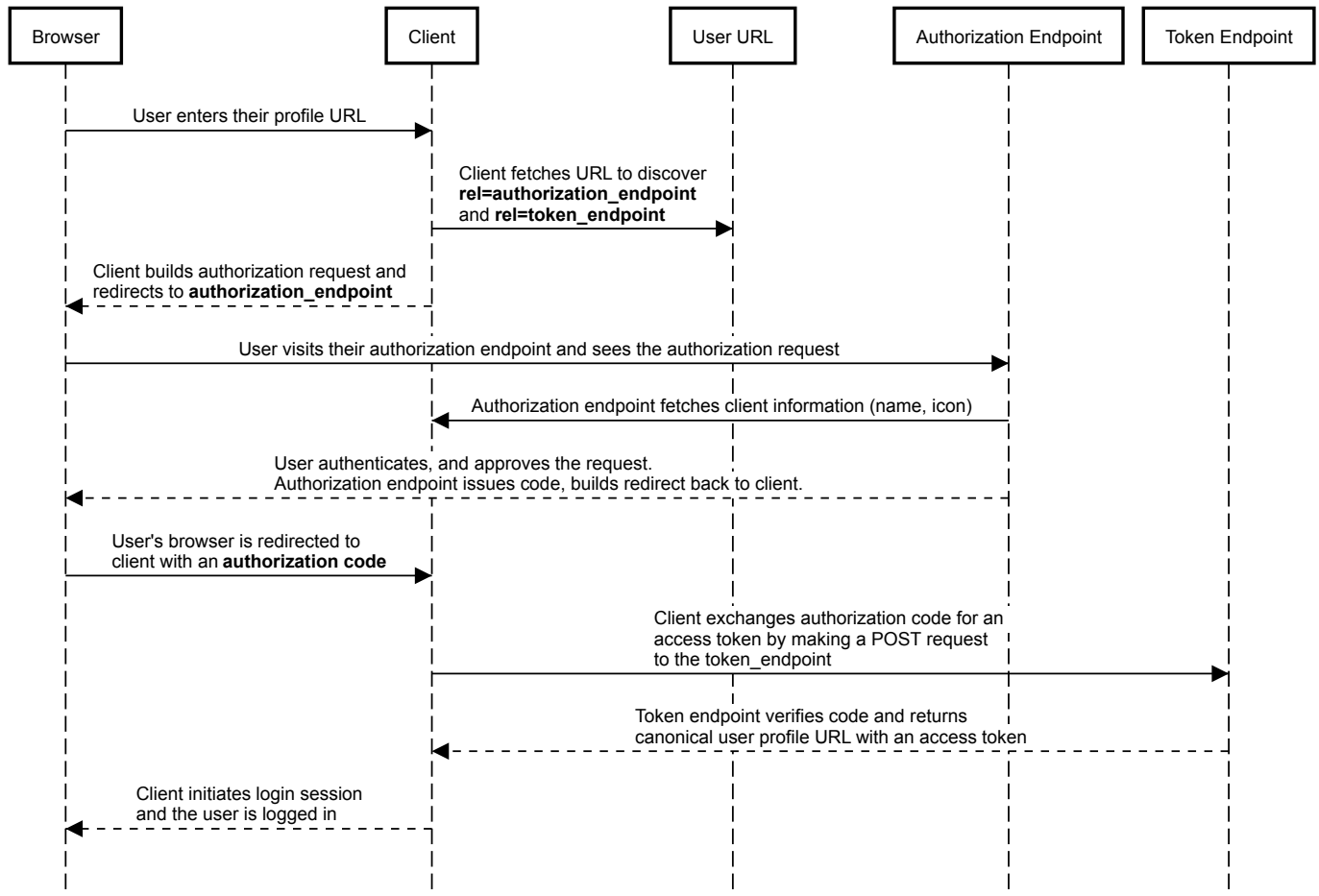
{
  "me": "https://user.example.net/"
}
```

The resulting profile URL *MAY* be different from what the user initially entered, but *MUST* be on the same domain. This gives the authorization endpoint an opportunity to canonicalize the user's URL, such as correcting `http` to `https`, or adding a path if required. See [Redirect Examples](#) for an example of how a service can allow a user to enter a URL on a domain different from their resulting `me` profile URL.

6. Authorization

This section describes how to obtain an access token using the Authorization Code Flow.

IndieAuth Authentication Flow Diagram



- The End-User enters their profile URL in the login form of the client and clicks "Sign in"
- The client discovers the End-User's authorization endpoint and token endpoint by fetching the profile URL and looking for the `rel=authorization_endpoint` and `rel=token_endpoint` values
- The client redirects the browser to the authorization endpoint, including its client identifier, requested scope, local state, and a redirect URL
- The authorization endpoint verifies the End-User, e.g. by logging in, and establishes whether the End-User grants or denies the client's request
- The authorization endpoint redirects the browser to the client's redirect URL, including an authorization code
- The client exchanges the authorization code for an access token by making a POST request to the token endpoint. The token endpoint validates the authorization code, and responds with the End-User's canonical profile URL and an access token

6.1 Discovery

After obtaining the End-User's profile URL, the client fetches the URL and looks for the `authorization_endpoint` and `token_endpoint` rel values in the HTTP `Link` headers and HTML `<link>` tags.

EXAMPLE 9

```
Link: <https://example.org/auth>; rel="authorization_endpoint"  
Link: <https://example.org/token>; rel="token_endpoint"  
  
<link rel="authorization_endpoint" href="https://example.org/auth">  
<link rel="token_endpoint" href="https://example.org/token">
```

6.2 Authorization Endpoint

6.2.1 Authorization Request

The client builds the authorization request URL by starting with the discovered `authorization_endpoint` URL and adding the following parameters to the query component:

- `response_type=code` - Indicates to the authorization server that this is an authorization request and an authorization code should be returned
- `me` - The profile URL that the user entered
- `client_id` - The client URL
- `redirect_uri` - The redirect URL indicating where the user should be redirected to after approving the request
- `state` - A parameter set by the client which will be included when the user is redirected back to the client. This is used to prevent CSRF attacks. The authorization server *MUST* return the unmodified state value back to the client.
- `scope` - (optional) A space-separated list of scopes the client is requesting, e.g. "create". If the client omits this value, the authorization server *MUST* use a default value.

EXAMPLE 10

```
https://example.org/auth?me=https://user.example.net/&
                                redirect_uri=https://app.example.com/redirect&
                                client_id=https://app.example.com/&
                                state=1234567890&
                                scope=create+update+delete&
                                response_type=code
```

The authorization endpoint *SHOULD* fetch the `client_id` URL to retrieve application information and the client's registered redirect URLs, see [Client Information Discovery](#) for more information.

If the URL scheme, host or port of the `redirect_uri` in the request do not match that of the `client_id`, then the authorization endpoint *SHOULD* verify that the requested `redirect_uri` matches one of the [redirect URLs](#) published by the client, and *SHOULD* block the request from proceeding if not.

It is up to the authorization endpoint how to authenticate the user. This step is out of scope of OAuth 2.0, and is highly dependent on the particular implementation. Some authorization servers use typical username/password authentication, and others use alternative forms of authentication such as [\[RelMeAuth\]](#).

Once the user is authenticated, the authorization endpoint presents the authorization prompt to the user. The prompt *MUST* indicate which application the user is signing in to, and *SHOULD* provide as much detail as possible about the request, such as information about the requested scopes.

6.2.2 Authorization Response

If the user approves the request, the authorization endpoint generates an authorization code and builds the redirect back to the client.

The redirect is built by starting with the `redirect_uri` in the request, and adding the following parameters to the query component of the redirect URL:

- `code` - The authorization code generated by the authorization endpoint. The code *MUST* expire shortly after it is issued to mitigate the risk of leaks. A maximum lifetime of 10 minutes is

recommended. See [OAuth 2.0 Section 4.1.2](#) for additional requirements on the authorization code.

- **state** - The state parameter *MUST* be set to the exact value that the client set in the request.

EXAMPLE 11

HTTP/1.1 302 Found

Location: `https://app.example.com/redirect?code=xxxxxxx
state=1234567890`

Upon the redirect back to the client, the client *MUST* verify that the state parameter in the request is valid and matches the state parameter that it initially created, in order to prevent CSRF attacks. The state value can also store session information to enable development of clients that cannot store data themselves.

6.3 Token Endpoint

6.3.1 Token Request

After the state parameter is validated, the client makes a POST request to the token endpoint to verify the authorization code and retrieve the final user profile URL. The POST request contains the following parameters:

- **grant_type=authorization_code**
- **code** - The authorization code received from the authorization endpoint in the redirect
- **client_id** - The client's URL, which *MUST* match the client_id used in the authorization request.
- **redirect_uri** - The client's redirect URL, which *MUST* match the initial authorization request.
- **me** - The user's profile URL as originally used in the authorization request

EXAMPLE 12

```
POST https://example.org/token
Content-type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=xxxxxxx
&client_id=https://app.example.com/
&redirect_uri=https://app.example.com/redirect
&me=https://user.example.net/
```

6.3.2 Authorization Code Verification

The token endpoint needs to verify that the authorization code is valid, and that it was issued for the matching `me`, `client_id` and `redirect_uri`, and contains at least one `scope`. If the authorization code was issued with no `scope`, the token endpoint *MUST NOT* issue an access token, as empty scopes are invalid per Section 3.3 of OAuth 2.0 [\[RFC6749\]](#).

If the authorization endpoint and token endpoint are tightly integrated, then they can use any mechanism to share the information about the authorization code, such as looking up the code in a database, or sharing a JWT secret. If the authorization endpoint and token endpoint are separate, then they *MUST* use the mechanism described below to communicate.

The token endpoint *MUST* make a POST request to the authorization endpoint to verify the authorization code if it is not able to verify it using other means. To find the authorization endpoint, the token endpoint uses the user's profile URL in the `me` parameter and performs [discovery](#) to find the user's `authorization_endpoint`. The token endpoint makes a POST request to the authorization endpoint with the following parameters:

- `code` - The authorization code received from the authorization endpoint in the redirect
- `client_id` - The client's URL, which *MUST* match the `client_id` used in the authorization request.
- `redirect_uri` - The client's redirect URL, which *MUST* match the initial authorization request.

EXAMPLE 13

```
POST https://example.org/auth
Content-type: application/x-www-form-urlencoded

code=xxxxxxx
&client_id=https://app.example.com/
&redirect_uri=https://app.example.com/redirect
```

Note that this is the same request that clients make to the authorization endpoint in the [authentication flow](#).

The authorization endpoint will validate that the code corresponds with the given `client_id` and `redirect_uri` and respond with a JSON response containing the `me` URL corresponding to this authorization code as well as the `scope` that was authorized, or an OAuth 2.0 error response. The error returned from the authorization endpoint is acceptable to pass through to the client.

EXAMPLE 14

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "me": "https://user.example.net/",
  "scope": "create update delete"
}
```

6.3.3 Access Token Response

If the request is valid, then the token endpoint can generate an access token and return the appropriate response. The token response is a JSON [\[RFC7159\]](#) object containing the OAuth 2.0 Bearer Token [\[RFC6750\]](#), as well as a property `me`, containing the canonical user profile URL for the user this access token corresponds to. For example:

EXAMPLE 15

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "access_token": "XXXXXX",
  "token_type": "Bearer",
  "scope": "create update delete",
  "me": "https://user.example.net/"
}
```

The resulting profile URL *MAY* be different from what the user initially entered, but *MUST* be on the same domain. This provides the opportunity to canonicalize the user's URL, such as correcting **http** to **https**, or adding a path if required. See [Redirect Examples](#) for an example of how a service can allow a user to enter a URL on a domain different from their resulting **me** profile URL.

6.3.4 Access Token Verification

Since in OAuth 2.0, access tokens are opaque to clients, clients do not need to know anything about the contents or structure of the token itself, if any. Additionally, endpoints that clients make requests to, such as [\[Micropub\]](#) endpoints, may not even understand how to interpret tokens if they were issued by a standalone token endpoint. If the token endpoint is not tightly integrated with the endpoint the client is interacting with, then the other endpoint needs a way to verify access tokens that it receives. If the token endpoint and Micropub endpoint are tightly coupled, then they can of course use an internal mechanism to verify access tokens.

Token endpoints that intend to interoperate with other endpoints *MUST* use the mechanism described below to allow other endpoints to verify access tokens.

If an external endpoint needs to verify that an access token is valid, it *MUST* make a GET request to the token endpoint containing an HTTP **Authorization** header with the Bearer Token according to [\[RFC6750\]](#). Note that the request to the endpoint will not contain any user-identifying information, so the external endpoint (e.g. Micropub endpoint) will need to know via out-of-band methods which token endpoint is in use.

EXAMPLE 16

```
GET https://example.org/token
Authorization: Bearer xxxxxxxx
```

The token endpoint verifies the access token using (how this verification is done is up to the implementation), and returns information about the token:

- **me** - The profile URL of the user corresponding to this token
- **client_id** - The client ID associated with this token
- **scope** - A space-separated list of scopes associated with this token

EXAMPLE 17

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "me": "https://user.example.net/",
  "client_id": "https://app.example.com/",
  "scope": "create update delete"
}
```

Specific implementations *MAY* include additional parameters as top-level JSON properties. Clients *SHOULD* ignore parameters they don't recognize.

6.3.5 Token Revocation

A client may wish to explicitly disable an access token that it has obtained, such as when the user signs out of the client. IndieAuth extends OAuth 2.0 Token Revocation [[RFC7009](https://tools.ietf.org/html/rfc7009)] by defining the following:

- The revocation endpoint is the same as the token endpoint.
- The revocation request includes an additional parameter, **action=revoke**.

An example revocation request is below.

EXAMPLE 18

```
POST https://example.org/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded

action=revoke
&token=xxxxxxx
```

7. Security Considerations

In addition to the security considerations in OAuth 2.0 Core [[RFC6749](#)] and OAuth 2.0 Threat Model and Security Considerations [[RFC6819](#)], the additional considerations apply.

7.1 Differing User Profile URLs

Clients will initially prompt the user for their profile URL in order to discover the necessary endpoints to perform authentication or authorization. However, there may be slight differences between the URL that the user initially enters vs what the system considers the user's canonical profile URL.

For example, a user might enter `user.example.net` in a login interface, and the client may assume a default scheme of `http`, providing an initial profile URL of `http://user.example.net`. Once the authentication or authorization flow is complete, the response in the `me` parameter might be the canonical `https://user.example.net/`. In some cases, user profile URLs have a full path component such as `https://example.net/username`, but users may enter just `example.net` in the login interface.

Clients *MUST* use the resulting `me` value from the [authorization code verification](#) or [access token response](#) rather than assume the initially-entered URL is correct, with the following condition:

- The resulting profile URL *MUST* have a matching domain of the initially-entered profile URL.

This ensures that an authorization endpoint is not able to issue valid responses for arbitrary profile URLs.

7.2 Preventing Phishing and Redirect Attacks

Authorization servers *SHOULD* fetch the `client_id` provided in the authorization request in order to provide users with additional information about the authorization request, such as the application name and logo. If the server does not fetch the client information, then it *SHOULD* take additional measures to ensure the user is provided with as much information as possible about the authorization request.

The authorization server *SHOULD* display the full `client_id` on the authorization interface, in addition to displaying the fetched application information if any. Displaying the `client_id` helps users know that they are authorizing the expected application.

Since all IndieAuth clients are public clients, and no strong client authentication is used, the only measure available to protect against some attacks described in [RFC6819] is strong verification of the client's `redirect_uri`. If the `redirect_uri` scheme, host or port differ from that of the `client_id`, then the authorization server *MUST* either verify the redirect URL as described in [Redirect URL](#), or display the redirect URL to the user so they can inspect it manually.

8. IANA Considerations

The link relation type below has been registered by IANA per Section 6.2.1 of [RFC5988]:

Relation Name:

authorization_endpoint

Description:

Used for discovery of the OAuth 2.0 authorization endpoint given an IndieAuth profile URL.

Reference:

[IndieAuth Specification \(https://indieauth.spec.indieweb.org/\)](https://indieauth.spec.indieweb.org/)

Relation Name:

token_endpoint

Description:

Used for discovery of the OAuth 2.0 token endpoint given an IndieAuth profile URL.

Reference:

[IndieAuth Specification \(https://indieauth.spec.indieweb.org/\)](https://indieauth.spec.indieweb.org/)

Relation Name:

redirect_uri

Description:

Used for discovery of the OAuth 2.0 redirect URI given an IndieAuth client ID.

Reference:

[IndieAuth Specification \(https://indieauth.spec.indieweb.org/\)](https://indieauth.spec.indieweb.org/)

A. Resources

This section is non-normative.

- [More IndieAuth resources](#)
- [Implementation guide for obtaining an access token using IndieAuth](#)
- [Implementation guide for authenticating users without obtaining an access token](#)

A.1 Articles

This section is non-normative.

You can find a list of [articles about IndieAuth](#) on the IndieWeb wiki.

A.2 Implementations

This section is non-normative.

You can find a list of [IndieAuth implementations](#) on indieauth.net

B. Acknowledgements

The editor wishes to thank the [IndieWeb](#) community and other implementers for their support, encouragement and enthusiasm, including but not limited to: Amy Guy, Barnaby Walters, Benjamin Roberts, Bret Comnes, Christian Weiske, François Kooman, Jeena Paradies, Martijn van der Ven, Sebastiaan Andeweg, Sven Knebel, and Tantek Çelik.

C. References

C.1 Normative references

[Fetch]

Fetch Standard. Anne van Kesteren. WHATWG. Living Standard. URL: <https://fetch.spec.whatwg.org/>

[h-app]

h-app. Aaron Parecki. microformats.org. Living Specification. URL: <http://microformats.org/wiki/h-app>

[HTML]

HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: <https://html.spec.whatwg.org/multipage/>

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: <https://tools.ietf.org/html/rfc2119>

[RFC5988]

Web Linking. M. Nottingham. IETF. October 2010. Proposed Standard. URL: <https://tools.ietf.org/html/rfc5988>

[RFC6749]

The OAuth 2.0 Authorization Framework. D. Hardt, Ed.. IETF. October 2012. Proposed Standard. URL: <https://tools.ietf.org/html/rfc6749>

[RFC6750]

The OAuth 2.0 Authorization Framework: Bearer Token Usage. M. Jones; D. Hardt. IETF. October 2012. Proposed Standard. URL: <https://tools.ietf.org/html/rfc6750>

[RFC6819]

OAuth 2.0 Threat Model and Security Considerations. T. Lodderstedt, Ed.; M. McGloin; P. Hunt. IETF. January 2013. Informational. URL: <https://tools.ietf.org/html/rfc6819>

[RFC7009]

[OAuth 2.0 Token Revocation](#). T. Lodderstedt, Ed.; S. Dronia; M. Scurtescu. IETF. August 2013. Proposed Standard. URL: <https://tools.ietf.org/html/rfc7009>

[RFC7159]

[The JavaScript Object Notation \(JSON\) Data Interchange Format](#). T. Bray, Ed.. IETF. March 2014. Proposed Standard. URL: <https://tools.ietf.org/html/rfc7159>

[RFC7231]

[Hypertext Transfer Protocol \(HTTP/1.1\): Semantics and Content](#). R. Fielding, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: <https://tools.ietf.org/html/rfc7231>

[URL]

[URL Standard](#). Anne van Kesteren. WHATWG. Living Standard. URL: <https://url.spec.whatwg.org/>

C.2 Informative references

[Microformats]

[Microformats](#). URL: <http://microformats.org>

[Micropub]

[Micropub](#). Aaron Parecki. W3C. 23 May 2017. W3C Recommendation. URL: <https://www.w3.org/TR/micropub/>

[RelMeAuth]

[RelMeAuth](#). Tantek Çelik. microformats.org. Living Specification. URL: <http://microformats.org/wiki/RelMeAuth>

[XFN11]

[XFN 1.1](#). Tantek Çelik; Matthew Mullenweg; Eric Meyer. Global Multimedia Protocols Group. Stable. URL: <https://gmpg.org/xfn/11>

