Mail Maintenance
Internet-Draft
Intended status: Informational
Expires: 20 September 2026

N.M. Jenkins, Ed.
Fastmail
B. Bucksch
Beonex

19 March 2026

OAuth Profile for Open Public Clients
draft-ietf-mailmaint-oauth-public-03

Abstract

This document specifies a profile of the OAuth authorization protocol to allow for interoperability between native clients and servers using open protocols, such as JMAP, IMAP, SMTP, POP, CalDAV, and CardDAV.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 20 September 2026.

Copyright Notice

Table of Contents

1. Introduction

This document pulls together several existing standards and uses them to specify a specific OAuth profile, allowing interoperable modern authentication for native clients of open protocols, such as JMAP, IMAP, SMTP, POP, CalDAV, and CardDAV. For these protocols, there are many servers and many clients with no pre-existing relationship, that need to be able to connect. At the moment, the only interoperable way to do so is with a basic username and password, which has many deficiencies from a security standpoint.

1.1. Presumptions

This OAuth flow presumes you have a set of one or more resource server endpoints (e.g., for JMAP/IMAP/SMTP/POP/CardDAV/CalDAV), and a common authorization server issuer identifier for all of them. For example, you might have a JMAP session endpoint https://api.example.com/jmap/session, an IMAP endpoint imaps://imap.example.com:993, and an issuer identifier https://auth.example.com.

1.2. Getting the Authorization Server Issuer Identifier

The endpoints may be discovered via an autoconfiguration mechanism or manual user input. Autoconfiguration may also include the issuer identifier. To allow OAuth configuration for manual input or when the issuer identifier is not included in the autoconfig, HTTP endpoints that support this profile MUST support Protected Resource Metadata [RFC9728], accessible via the oauth-protected-resource well-known path, as specified in Section 3 of [RFC9728]. The authorization_servers property MUST be present in the resource metadata, with the value containing the authorization server issuer identifier. HTTP requests made without valid authentication to the resource endpoints MUST indicate the protected resource metadata URL in the WWW-Authenticate HTTP response header field, as described in Section 5 of [RFC9728].

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. The Open Public Client OAuth Profile

3.1. Overview

OAuth 2 [RFC6749] can be used in many different ways. This document specifies one particular set of options to ensure interoperability and security. Servers may implement more options, but MUST support the flow as described in this document for interoperability with clients. Similarly, clients may choose to support additional flows but there is no guarantee that this will be interoperable.

The general flow works like this:

1. The OAuth 2.0 Authorization Server Metadata [RFC8414] is fetched.

2. The client instance registers with the authorization server to get a client id using the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591].
3. The client instance authorizes using the Authorization Code Grant flow ([RFC6749], Section 4.1) with PKCE [RFC7636], Issuer Identification [RFC9207] and Resource Indicators [RFC8707].
4. The client instance gets an access token and refresh token, as per Section 5 of [RFC6749].

The access token can now be used as a Bearer token to authenticate requests to the application servers as per [RFC6750] for HTTP requests, or [RFC7628] for SASL authentication. When it expires, a new one can be requested using the refresh token as per Section 6 of [RFC6749].

The rest of this document describes in detail each of the above steps.

3.2. Fetching the Authorization Server Metadata

The authorization server issuer identifier MUST be an HTTPS URL with no path, query parameters, or fragment part, i.e. it MUST be just "https://" followed by a hostname. Otherwise, abort the flow. The authorization server metadata may be fetched by a GET request to the /.well-known/oauth-authorization-server path at this origin.

When fetched, a successful response is indicated by a 200 OK HTTP status code. The response MUST have an "application/json" content type. Anything else MUST be treated as an error, and the flow MUST be aborted.

The authorization server metadata is a JSON document with properties as specified in [RFC8414]. It MUST include the following properties:

issuer
: The authorization server’s issuer identifier. This MUST be identical to the issuer identifier used to fetch this document. For example, if this document was fetched from https://auth.example.com/.well-known/oauth-authorization-server, the issuer MUST be https://auth.example.com. If not, the flow MUST be aborted.

  This MUST be verified again during the authorization flow (see Section 3.4, or [RFC9207] for more details), to prevent against mix-up attacks.

registration_endpoint

The URL the client instance will use to register, to get a client id it needs for authorization (see Section 3.3).

authorization_endpoint
The URL the client instance will use to start the authorization process (see Section 3.4) once it has registered.

token_endpoint
The URL the client instance will use to get refresh and access tokens after successful authorization (see Section 3.5).

scopes_supported
An array of supported scopes on the server (see Section 3.7).

response_types_supported
A list of response types supported in the OAuth authorization flow. This is an array of strings that MUST include "code".

grant_types_supported
A list of the OAuth 2.0 grant type values that this authorization server supports. This is an array of strings that MUST include "authorization_code" and "refresh_token".

token_endpoint_auth_methods_supported
This is an array of strings that MUST include "none".

code_challenge_methods_supported
An array of strings listing Proof Key for Code Exchange (PKCE) [RFC7636] code challenge methods supported by this authorization server. This MUST include "S256".

authorization_response_iss_parameter_supported
This MUST have the boolean value true.

The metadata MAY include other properties, or other values for multi-valued properties, however clients are not required to understand or use any of them for interoperability.

It is RECOMMENDED servers support DPoP ([RFC9449]) to allow sender-constrained refresh and access tokens for HTTP-based protocols. Servers that support DPoP MUST include the following property:

dpop_signing_alg_values_supported
A JSON array containing a list of the JWS alg values (from the [IANA.JOSE.ALGS] registry) supported by the authorization server for DPoP proof JWTs, as defined in [RFC9449], Section 5.1.

Also of possible interest to client/server implementors following this document:

revocation_endpoint
: The URL the client instance can use to revoke their tokens, as per [RFC7009].

revocation_endpoint_auth_methods_supported
: If a revocation_endpoint is included, this property MUST be included, and is an array of strings that MUST include "none".

Clients MUST verify the required properties are present and conform to the requirements of this document. If not, the server is not using OAuth in conformance with this document and no compatibility may be presumed. It is RECOMMENDED clients abort the flow in such a case.

3.3. Dynamic Client Registration

To register, the client instance sends an HTTP POST to the client registration endpoint (as found in the metadata) with a content type of "application/json", and a body consisting of a JSON document with the following properties:

redirect_uris
: An array of URIs the client instance may use to receive back information at the end of the authorization flow. Each URI MUST satisfy all of these conditions:

  * The URI MUST start with one of the following:
    - http://127.0.0.1/
    - http://[::1]/
    - A private-use scheme in reverse domain notation, e.g., com.example:/. Such a scheme MUST have at least one dot in it.
  * The URI MUST NOT include two consecutive dots (e.g., /../).
  * The URI MUST NOT include a fragment part (#).

  The URI may include a path and query parameters.

token_endpoint_auth_method
: This MUST be "none".

grant_types
: This is an array of strings that MUST include "authorization_code" and "refresh_token".

response_types
: This is an array of strings that MUST include "code".

scope
: A string containing a space-separated list of scope values the client may request access for. (Note! This is not a JSON array.)

  For compatibility with servers implementing OpenID Connect [openid-connect], if the server advertised the offline_access scope in the "scopes_supported" property of the authorization server metadata, the client instance MUST also register for this scope.

client_name
: The name of the client software to be presented to the end-user during authorization.

client_uri
: A URL for a web page providing information about the client software. This MUST use HTTPS.

logo_uri
: A URL for a logo to display for this client software. This SHOULD be square, and in a PNG or SVG image format. This MUST use HTTPS.

tos_uri
: A URL that points to a human-readable terms of service or license document for the client software. This MUST use HTTPS.

policy_uri
: A URL that points to a human-readable privacy policy document for the client software. This MUST use HTTPS.

software_id
: A unique identifier string (e.g., a Universally Unique Identifier (UUID)) assigned by the client developer or software publisher, used by registration endpoints to identify the client software doing the dynamic registration. Unlike "client_id", which is issued by the authorization server and may vary between instances, the "software_id" SHOULD remain the same for all instances of the client software. The "software_id" SHOULD remain the same across multiple updates or versions of the same piece of software. The value of this field is not intended to be human readable and is usually opaque to the client and authorization server.

software_version
: A version identifier string for the client software identified by "software_id". The value of the "software_version" SHOULD change

on any update to the client software identified by the same "software_id". The value of this field is intended to be compared using string equality matching and no other comparison semantics are defined by this specification.

If the server indicated in its metadata that it supports DPoP [RFC9449] and the client instance is intending to authenticate all requests using DPoP, the client instance SHOULD also include the following property:

dpop_bound_access_tokens
true

(If set, the server MUST then require all token requests associated with the issued client id use DPoP. Note, DPoP is only currently defined for HTTP protocols, so this precludes usage for non-HTTP protocols as of time of writing.)

The server will check that all required properties are present and have valid values. Any unknown properties supplied by the client instance MUST just be ignored. The authorization server MAY replace any of the client's requested metadata values submitted during the registration and substitute them with suitable values.

If there is an exact match for all properties except for software_version, an existing registration may be returned. Otherwise, servers SHOULD create a new registration and client id.

There is no way to verify the authenticity of the information supplied by the client, however the general case of accurate information is still useful to the server, for example to be able to contact client authors to help debug issues if aberrant behaviour is observed. Servers MAY choose to ignore all of the information instead and just return a static client id to all requests.

The redirect URI restrictions MUST be enforced. These ensure the OAuth flow can only be completed by native clients — not web clients. Since a malicious native client could present the user with a custom browser to phish credentials anyway, the lack of verification of client registration details does not provide additional danger beyond existing threats. Allowing seamless dynamic registration for web-based clients, however, unfortunately makes it much easier for a phishing site to gain access to an account, by sending the user through the OAuth flow.

If successful, the server responds with an HTTP 201 Created status code and a body of type "application/json", with the content being a JSON object containing all the properties submitted during registration (with their values as set by the server, if overwritten), plus the following property:

client_id
: The OAuth 2.0 client identifier string, used in the authorization flow (see Section 3.4).

If the registration fails, the server will respond with an HTTP 400 status code and a JSON body as described in [RFC7591], section 3.2.2.

3.4. Authorization

Clients construct an authorization request URL by taking the authorization_endpoint from the authorization server metadata and adding the following additional query parameters:

client_id
: The client id as returned in the registration.

redirect_uri
: One of the redirect uris registered by the client instance. This MUST be identical to the registered URI. The one exception to this is if a URI with the prefix "http://127.0.0.1/" or "http://[::1]/" was registered, in which case the matching URI MUST also include an arbitrary port.

  For example, if http://127.0.0.1/redirect was registered, then the client instance could send http://127.0.0.1:49152/redirect as the redirect_uri for authorization.

response_type
: This MUST be "code".

scope
: A space delimited set of scopes the client instance would like access to. This MUST be a subset of the scopes registered for this client id.

  For compatibility with servers implementing OpenID Connect [openid-connect], if the server advertised the offline_access scope in the "scopes_supported" property of the authorization server metadata, the client instance MUST also request this scope.

code_challenge

A PKCE code challenge as per [RFC7636], using SHA 256. To generate a challenge, first generate a code_verifier: a high-entropy cryptographic random string using the unreserved characters [A-Z] / [a-z] / [0-9] / "-" / "." / "_" / "~" from Section 2.3 of [RFC3986], with a minimum length of 43 characters and a maximum length of 128 characters.

The code_challenge is then BASE64URL-ENCODE(SHA256(ASCII(code_verifier))).

code_challenge_method
This MUST be "S256".

resource
The URL for the initial resource endpoint (as returned in the autodiscover) you wish to access after successful authorization. For example, https://api.example.com/jmap/session. If you wish to use multiple protocols, the client instance MUST include multiple "resource" query parameters: one for each endpoint. The server MUST verify that all requested endpoints are permitted locations to send access tokens to (i.e., they are the real resource endpoints associated with this authorization server), in order to prevent mix up attacks.

state
An opaque value used by the client instance to verify that an authorization response is due to a request that the client instance initiated. The authorization server will include this value when redirecting the user-agent back to the client instance. Client instances MUST generate a state with a unique, unguessable random string when initiating an authorization request.

login_hint (optional)
The username the user originally asked to log in with. The server can prefill this in a login form. Note, the user may choose to log in with a different username.

After constructing the authorization request URL, the app uses platform-specific APIs to open it in an external user-agent. Typically, the external user-agent used is the default browser, that is, the application configured for handling "http" and "https" scheme URIs on the system. See Section 6 of [RFC8252] for a discussion of best practices and alternatives. The user-agent MUST support session cookies, JavaScript, and the Web Authentication API [webauthn].

The authorization server MUST verify all required parameters, and that they conform to the restrictions in this document. Other URL parameters MAY be supplied but will be ignored.

If verified, the authorization server will authenticate the user and ask them if they wish to grant authorization to the client.

If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server SHOULD inform the user of the error and MUST NOT automatically redirect the user-agent to the invalid redirection URI. If the authorization request fails for any other reason, the client instance will receive an error response via the redirect_uri. This MUST include an error query parameter with an appropriate error code, as defined in Section 4.1.2.1 of [RFC6749].

If authorization is successful, the client instance will receive a response via the redirect_uri, which will include the following query parameters:

code
: The authorization code. This may be exchanged for the refresh token. This code MUST expire, and MUST remain valid for at least 10 minutes from authorization. It MUST NOT be used again once the client instance has successfully exchanged it for a refresh token. Doing so may cause the server to detect it as stolen and revoke all associated tokens.

state
: The value of the state parameter that was passed in with the initial request.

iss
: The issuer identifier of the authorization server.

The client instance MUST verify all of the following:

* The "iss" returned is identical to the "issuer" property in the authorization server metadata. This is critical for preventing against various mix-up attacks should a malicious OAuth metadata object be fetched, as discussed in the security considerations.
* The state returned matches exactly the state it sent, to verify that this request was indeed initiated by the client and not an attacker.

If any verification fails, the client instance MUST abort the flow and not send the authorization code anywhere. It MUST return a page to the user-agent with a description of what went wrong and what the user can do next to solve the problem.

3.5. Obtaining a Refresh Token

Following authorization, the client instance will obtain initial refresh and access tokens by making a POST request to the token_endpoint URL. The following parameters MUST be present, using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body:

client_id
: The client id as returned in the registration.

redirect_uri
: The redirect_uri parameter sent with the authorization request from which the code was obtained.

grant_type
: This MUST be "authorization_code".

code
: The code returned via the redirect back from authorization.

code_verifier
: The code_verifier generated for the authorization (the random string generated in the authorization step, as per [RFC7636]).

Other parameters MAY be supplied but will be ignored. If using DPoP, the client instance MUST also set a DPoP header in accordance with Section 5 of [RFC9449].

The server will verify the parameters and if successful, return a 200 OK response with a content type of application/json. The body will be a JSON object with the following properties:

access_token
: A bearer token used to authenticate API requests. This will be valid for a fixed, limited time.

token_type
: The type of the access token. This MUST be "Bearer", or "DPoP" if the client is using DPoP ([RFC9449]).

expires_in
: The lifetime in seconds of the access token. For example, the value 3600 denotes that the access token will expire in one hour from the time the response was generated.

scope
: The space delimited set of scopes that this access token may use.

Note, this MAY be different to the set of scopes requested. Servers MAY allow users to choose to authorize only a subset of the requested scopes. Clients MUST check the set of scopes granted is sufficient for its needs. The server MAY also return scopes beyond those requested, for example a server that adds JMAP support when previously it only supported IMAP may choose to give scopes that enable JMAP access to the same data, even if only the IMAP scope was requested, so the client instance can seamlessly upgrade protocols.

refresh_token
The refresh token to use next time the client instance needs to get a new access token.

If the request fails, the server MUST return a 400 Bad Request response with a content type of application/json. The body will be a JSON object with properties as described in Section 5.2 of [RFC6749].

3.6. Using the Access Token

The client instance is now authenticated. It can connect to the servers given in the discovered autoconfiguration with the Bearer scheme [RFC6750]. For HTTP-based protocols, this means setting an Authorization header with the value Bearer {access_token} (where {access_token} is replaced with the value of the access_token). If using DPoP, the client instance must also set a DPoP header in accordance with Section 7 of [RFC9449].

For protocols that use SASL authentication, such as IMAP, POP, and SMTP, the access token is used in accordance with the OAUTHBEARER mechanism defined in [RFC7628]. Servers MUST NOT require the username associated with the resource (the "authzid") be included in the GS2 header.

3.7. Scopes

To work interoperably, clients and server must use a standard set of scopes for access. This document creates a new IANA registry for such scopes (see Section 5), and registers scopes to cover IMAP, SMTP, POP, CardDAV, CalDAV, and JMAP.

3.8. Getting a New Access Token

Clients SHOULD keep using an access token they have been issued until it expires, which will result in getting a 401 error back.

When the access token expires, the client instance MUST get a new one by making another POST request to the authorization server token endpoint. The following parameters MUST be present, using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body:

client_id
: The client id as returned in the registration.

grant_type
: This MUST be "refresh_token".

refresh_token
: The refresh token returned last time the client instance obtained a new access token.

The success and failures responses are identical to those documented in "Obtaining a refresh token" (Section 3.5).

A new refresh token SHOULD be returned in the response and the client instance MUST replace their previous refresh token with this if given. The client instance MUST NOT try to use an old refresh token again; this SHOULD result in the authorization being revoked as a protection against leaked refresh tokens.

If a user has multiple devices with the same client software installed, each instance of the client MUST obtain separate authorization. You cannot share a refresh token between devices.

3.9. Token Expiry Times

To ensure a good user experience, access tokens MUST have a lifetime of at least 1 hour. Refresh tokens SHOULD NOT expire, unless the user explicitly revokes the token or it has been unused for a considerable period of time (at least 30 days).

For stateful connection-based protocols such as SMTP and IMAP the access token is only presented on first connection. Expiry of the access token SHOULD NOT affect current IMAP/SMTP etc. sessions that have already authenticated. However, should the server choose to force reauthorization it MUST do so by unilaterally closing the connection, as there is no way to reauthenticate an existing session in these protocols and all clients have to be able to handle reconnection in the case of dropped connections.

If the refresh token is revoked, all existing stateful connections MUST immediately be closed.

3.10. Client Id Validity

The client id MUST remain valid for as long as there is a valid refresh token associated with that client id.

To ensure a good user experience, the client id assigned upon dynamic registration SHOULD be valid for use in the authorization flow (Section 3.4) for at least one hour after issuance.

Clients MUST associate a client id with each refresh token it has. If the refresh token is invalidated and the client instance has to reauthenticate, it MUST submit the dynamic client registration (Section 3.3) again and use the client id returned, which MAY be different to the previous time.

Servers that wish to store client registration information MUST be careful of resource exhaustion attacks. A RECOMMENDED approach is to:

* Normalize and securely hash the submitted registration data.

* Check if this hash already exists in the database, and if so return the existing client id issued for this registration.

* Otherwise, keep the registration in temporary storage and only insert it into the database upon successful authorization. Creating a new client id in the temporary storage can be rate limited, for example by IP.

* Remove any client registrations from the database whenever there are no valid refresh tokens associated with that client id.

4. Security Considerations

This profile mandates best practices for OAuth with native clients, as defined in [RFC8252]. A thorough discussion of the security considerations generally applicable to OAuth is out of scope for this document, but can be found in [RFC6819], as well as the security considerations (Section 10) of [RFC6749].

Implementors are encouraged to read all of the above documents for a more thorough consideration of the specific threats and mitigations with OAuth.

The choices made for this profile are intended to mitigate as far as possible the inherent risks that come from allowing arbitrary clients to talk to arbitrary servers.

The key restriction of this profile is that the redirect_uri MUST be something only a native client can access. If the user has downloaded and run a malicious native app, it could already undetectably spoof the user's browser to phish them, or in unsandboxed environments install malware, so supporting the OAuth flow from an unknown client is not increasing risks. Indeed, it is more secure than the current alternative, which is legitimate clients storing the user's password.

The dynamic registration part of this document is not a security component, as there is no way to verify any of the data. The data in the registration may be shown to the user as part of the authorization flow, which may help with phishing, but as noted above the redirect_uri can only be used by a native app, which could already phish the user. However, the registration gives the server more information to detect suspicious behaviour, which can help it to detect compromised users and devices more easily.

The issuer is expected to be autodetected from the user's email address. A threat scenario that must be considered is the user making a small typo in the domain (especially for a common email service), and an attacker controlling this domain. In this scenario the client instance will fetch the OAuth metadata from the attacker's server, and has no way to know it is not the real server the user wishes to connect to. This leads to a number of threats:

1. The attacker defines a malicious authorization_endpoint under their control. They attempt to phish the user's credentials with this. This is not something that can be specifically mitigated by the requirements of this document, however requiring origin-bound authentication such as passkeys for authentication will mitigate this, and most browsers have a block list of known phishing sites that can also help mitigate this.

2. The attacker defines the real authorization_endpoint and token_endpoint, but their own resource servers. This is protected against by the use of Resource Indicators ([RFC8707]) — the client instance must send the list of all resource endpoints it intends to connect to with the authorization request. If an unknown resource server is present, the server can reject the request.

   It is also protected against via the issuer identifier. The authorization response will include an "iss" parameter which will be the legitimate issuer identifier. However, the issuer for the metadata will not match, as this must be at the attacker's domain (if it were not, the client instance will have aborted the flow after fetching the metadata, as it would not match the domain it

was fetched from). Therefore a client instance following this specification will abort the flow and not send the authorization code to the token endpoint.

3. The attacker defines the real authorization_endpoint and resource servers, but their own token_endpoint. The issuer identifier check above will also protect against this.

5. IANA Considerations

5.1. Interoperable OAuth Scopes Registry

IANA shall add a new registry called "Interoperable OAuth Scopes" to the "OAuth Parameters" registry group. The registry records interoperable OAuth scope values. Each scope provides specific access rights to protected resources.

5.1.1. Registration Procedure

The "Specification Required" policy, as defined in Section 4.6 of [RFC8126], shall govern registrations in this registry. The Designated Expert shall verify that:

1. The proposed scope name follows established naming conventions.

2. The description clearly defines the access rights granted by the scope, with reference to publicly available and stable documentation sufficient for interoperability.

3. The change controller is clearly identified.

5.1.2. Registration Template

Scope Name: The name of the OAuth scope value.
Description: A clear description of what access rights this scope grants to the client, with reference to publicly available and stable documentation sufficient for interoperability.
Change Controller: The name of the person, organization, or entity controlling changes to the scope definition.

5.1.3. Initial Registry Contents

Scope Name: urn:ietf:params:oauth:scope:mail
Description: Requests access for a client to manage the user’s email. This MUST include full access via all of the following protocols that the server supports:
* IMAP [RFC9051]

* POP [RFC1939]

* SMTP submission [RFC6409]

* JMAP [RFC8620] - this scope grants the right to use all JMAP capabilities that the server supports for which urn:ietf:params:oauth:scope:mail is included in the Interoperable OAuth Scopes column of the JMAP capabilities registry.

Change Controller: IETF

Scope Name: urn:ietf:params:oauth:scope:contacts
Description: Requests access for a client to manage the user's contacts. This MUST include full access via the CardDAV protocol [RFC6352] if the server supports it. If the server supports JMAP [RFC8620], this scope grants the right to use all JMAP capabilities that the server supports for which urn:ietf:params:oauth:scope:contacts is included in the Interoperable OAuth Scopes column of the JMAP capabilities registry.
Change Controller: IETF

Scope Name: urn:ietf:params:oauth:scope:calendars
Description: Requests access for a client to manage the user's calendars. This MUST include full access via the CalDAV protocol [RFC4791] if the server supports it. If the server supports JMAP [RFC8620], this scope grants the right to use all JMAP capabilities that the server supports for which urn:ietf:params:oauth:scope:calendars is included in the Interoperable OAuth Scopes column of the JMAP capabilities registry.
Change Controller: IETF

Scope Name: offline_access
Description: May be advertised or requested for compatibility with OpenID connect, as described in this document.
Change Controller: IETF

5.2. Update to the JMAP Capabilties Registry

IANA shall update the "JMAP Capabilities" registry established in [RFC8620] to add a new column called "Interoperable OAuth Scopes". For each capability, this column shall contain a list of zero or more scopes registered in the Interoperable OAuth Scopes Registry. Authorization for any scope in the list will grant access to use that JMAP capability, if the capability is supported by the server.

Future registrations in the "JMAP Capabilities" registry SHOULD specify a value for the Interoperable OAuth Scopes column in addition to the fields established in the registration template by Section 9.4.5 of [RFC8620]. If omitted, registrations will be added with no value for this column (i.e., no interoperable scopes are available to grant access to this capability via OAuth).

5.2.1. Initial Interoperable OAuth Scopes Values for the JMAP Capabilities Registry

IANA shall update the existing JMAP Capabilities registrations with the following values for the Interoperable OAuth Scopes column.

Capability Name: urn:ietf:params:jmap:core
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail,
urn:ietf:params:oauth:scope:contacts,
urn:ietf:params:oauth:scope:calendars

Capability Name: urn:ietf:params:jmap:mail
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail

Capability Name: urn:ietf:params:jmap:mdn
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail

Capability Name: urn:ietf:params:jmap:smimeverify
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail

Capability Name: urn:ietf:params:jmap:submission
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail

Capability Name: urn:ietf:params:jmap:vacationresponse
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail

Capability Name: urn:ietf:params:jmap:blob
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail,
urn:ietf:params:oauth:scope:contacts,
urn:ietf:params:oauth:scope:calendars

Capability Name: urn:ietf:params:jmap:quota
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail,
urn:ietf:params:oauth:scope:contacts,
urn:ietf:params:oauth:scope:calendars

Capability Name: urn:ietf:params:jmap:sieve
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail

Capability Name: urn:ietf:params:jmap:principals
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail,

urn:ietf:params:oauth:scope:contacts,
urn:ietf:params:oauth:scope:calendars

Capability Name: urn:ietf:params:jmap:principals:owner
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail,
urn:ietf:params:oauth:scope:contacts,
urn:ietf:params:oauth:scope:calendars

Capability Name: urn:ietf:params:jmap:contacts
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:contacts

Capability Name: urn:ietf:params:jmap:calendars
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:calendars

Capability Name: urn:ietf:params:jmap:principals:availability
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:calendars

Capability Name: urn:ietf:params:jmap:webpush-vapid
Interoperable OAuth Scopes: urn:ietf:params:oauth:scope:mail,
urn:ietf:params:oauth:scope:contacts,
urn:ietf:params:oauth:scope:calendars

6. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <https://www.rfc-editor.org/info/rfc6819>.

[RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <https://www.rfc-editor.org/info/rfc7009>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[RFC7628] Mills, W., Showalter, T., and H. Tschofenig, "A Set of Simple Authentication and Security Layer (SASL) Mechanisms for OAuth", RFC 7628, DOI 10.17487/RFC7628, August 2015, <https://www.rfc-editor.org/info/rfc7628>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <https://www.rfc-editor.org/info/rfc8252>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, <https://www.rfc-editor.org/info/rfc8707>.

[RFC9207] Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identification", RFC 9207, DOI 10.17487/RFC9207, March 2022, <https://www.rfc-editor.org/info/rfc9207>.

[RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", RFC 9449, DOI 10.17487/RFC9449, September 2023, <https://www.rfc-editor.org/info/rfc9449>.

[RFC9728] Jones, M.B., Hunt, P., and A. Parecki, "OAuth 2.0 Protected Resource Metadata", RFC 9728, DOI 10.17487/RFC9728, April 2025, <https://www.rfc-editor.org/info/rfc9728>.

[webauthn] W3C, "Web Authentication: An API for accessing Public Key Credentials, Level 2", <https://www.w3.org/TR/webauthn/>.

7. Informative References

[RFC1939] Myers, J. and M. Rose, "Post Office Protocol - Version 3", STD 53, RFC 1939, DOI 10.17487/RFC1939, May 1996, <https://www.rfc-editor.org/info/rfc1939>.

[RFC4791] Daboo, C., Desruisseaux, B., and L. Dusseault, "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, DOI 10.17487/RFC4791, March 2007, <https://www.rfc-editor.org/info/rfc4791>.

[RFC6352] Daboo, C., "CardDAV: vCard Extensions to Web Distributed Authoring and Versioning (WebDAV)", RFC 6352, DOI 10.17487/RFC6352, August 2011, <https://www.rfc-editor.org/info/rfc6352>.

[RFC6409] Gellens, R. and J. Klensin, "Message Submission for Mail", STD 72, RFC 6409, DOI 10.17487/RFC6409, November 2011, <https://www.rfc-editor.org/info/rfc6409>.

[RFC8620] Jenkins, N. and C. Newman, "The JSON Meta Application Protocol (JMAP)", RFC 8620, DOI 10.17487/RFC8620, July 2019, <https://www.rfc-editor.org/info/rfc8620>.

[RFC9051] Melnikov, A., Ed. and B. Leiba, Ed., "Internet Message Access Protocol (IMAP) - Version 4rev2", RFC 9051, DOI 10.17487/RFC9051, August 2021, <https://www.rfc-editor.org/info/rfc9051>.

[openid-connect] OpenID Foundation, "OpenID Connect", <https://openid.net/developers/specs/>.

Authors' Addresses

Neil Jenkins (editor)
Fastmail
PO Box 234, Collins St West
Melbourne VIC 8007
Australia
Email: neilj@fastmailteam.com
URI: https://www.fastmail.com

Ben Bucksch
Beonex
Email: ben.bucksch@beonex.com