

Web Authorization Protocol
Internet-Draft
Intended status: Standards Track
Expires: May 5, 2020

T. Lodderstedt
yes.com
B. Campbell
Ping Identity
N. Sakimura
Nomura Research Institute
D. Tonge
Moneyhub Financial Technology
F. Skokan
Auth0
November 2, 2019

OAuth 2.0 Pushed Authorization Requests
draft-lodderstedt-oauth-par-01

Abstract

This document defines the pushed authorization request endpoint, which allows clients to push the payload of an OAuth 2.0 authorization request to the authorization server via a direct request and provides them with a request URI that is used as reference to the data in a subsequent authorization request.

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 May 5, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

(<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

| | |
|--|----|
| 1. Introduction | 2 |
| 1.1. Conventions and Terminology | 4 |
| 2. Pushed Authorization Request Endpoint | 5 |
| 2.1. Request | 5 |
| 2.2. Successful Response | 7 |
| 2.3. Error Response | 8 |
| 3. "request" Parameter | 9 |
| 3.1. Error responses for Request Object | 10 |
| 3.1.1. Authentication Required | 10 |
| 4. Authorization Request | 10 |
| 5. Authorization Server Metadata | 10 |
| 6. Security Considerations | 11 |
| 6.1. Request URI Guessing | 11 |
| 6.2. Open Redirection | 11 |
| 6.3. Request Object Replay | 11 |
| 6.4. Client Policy Change | 11 |
| 7. Acknowledgements | 11 |
| 8. IANA Considerations | 11 |
| 9. References | 12 |
| 9.1. Normative References | 12 |
| 9.2. Informative References | 12 |
| 9.3. URIs | 13 |
| Appendix A. Document History | 13 |
| Authors' Addresses | 13 |

1. Introduction

In OAuth [RFC6749] authorization request parameters are typically sent as URI query parameters via redirection in the user-agent. This is simple but also yields challenges:

- o There is no cryptographic integrity and authenticity protection, i.e. the request can be modified on its way through the user-agent and attackers can impersonate legitimate clients.
- o There is no mechanism to ensure confidentiality of the request parameters.

- o Authorization request URLs can become quite large, especially in scenarios requiring fine-grained authorization data.

JWT Secured Authorization Request (JAR) [I-D.ietf-oauth-jwsreq] provides solutions for those challenges by allowing OAuth clients to wrap authorization request parameters in a signed, and optionally encrypted, JSON Web Token (JWT), the so-called "Request Object".

In order to cope with the size restrictions, JAR introduces the "request_uri" parameter that allows clients to send a reference to a request object instead of the request object itself.

This document complements JAR by providing an interoperable way to push the payload of a request object directly to the AS in exchange for a "request_uri".

It also allows for clients to push the form encoded authorization request parameters to the AS in order to exchange them for a request URI that the client can use in a subsequent authorization request.

For example, the following authorization request,

```
GET /authorize?response_type=code
  &client_id=s6BhdRkqt3&state=af0ifjsldkj
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb HTTP/1.1
Host: as.example.com
```

could be pushed directly to the AS by the client as follows:

```
POST /as/par HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxS3REUmJuZlZkbUl3

response_type=code
&client_id=s6BhdRkqt3&state=af0ifjsldkj
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
```

The AS responds with a request URI,

```
HTTP/1.1 201 Created
Cache-Control: no-cache, no-store
Content-Type: application/json

{

  "request_uri": "urn:example:bwc4JK-ESC0w8acc191e-Y1LTC2",
  "expires_in": 90
}
```

which is used by the client in the subsequent authorization request as follows,

```
GET /authorize?request_uri=
urn%3Aexample%3Abwc4JK-ESC0w8acc191e-Y1LTC2 HTTP/1.1
```

The pushed authorization request endpoint fosters OAuth security by providing all clients a simple means for an integrity protected authorization request, but it also allows clients requiring an even higher security level, especially cryptographically confirmed non-repudiation, to explicitly adopt JWT-based request objects.

As a further benefit, the pushed authorization request allows the AS to authenticate the clients before any user interaction happens, i.e., the AS may refuse unauthorized requests much earlier in the process and has much higher confidence in the client's identity in the authorization process than before.

This is directly utilized by this draft to allow confidential clients to set the redirect URI for every authorization request, which gives them more flexibility in building redirect URI. And if the client IDs and credentials are managed by some external authority (e.g. a certification authority), explicit client registration with the particular AS could practically be skipped.

1.1. Conventions and Terminology

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.

This specification uses the terms "access token", "refresh token", "authorization server", "resource server", "authorization endpoint", "authorization request", "authorization response", "token endpoint", "grant type", "access token request", "access token response", and "client" defined by The OAuth 2.0 Authorization Framework [RFC6749].

2. Pushed Authorization Request Endpoint

The pushed authorization request endpoint is an HTTP API at the authorization server that accepts POST requests with parameters in the HTTP request entity-body using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 as described in Appendix B of [RFC6749].

The endpoint accepts the parameters defined in [RFC6749] for the authorization endpoint as well as all applicable extensions defined for the authorization endpoint. Some examples of such extensions include PKCE [RFC7636], Resource Indicators [I-D.ietf-oauth-resource-indicators], and OpenID Connect [OIDC].

The rules for client authentication as defined in [RFC6749] for token endpoint requests, including the applicable authentication methods, apply for the pushed authorization request endpoint as well. If applicable, the "token_endpoint_auth_method" client metadata parameter indicates the registered authentication method for the client to use when making direct requests to the authorization server, including requests to the pushed authorization request endpoint.

Note that there's some potential ambiguity around the appropriate audience value to use when JWT client assertion based authentication is employed. To address that ambiguity the issuer identifier URL of the AS according to [RFC8414] SHOULD be used as the value of the audience. In order to facilitate interoperability the AS MUST accept its issuer identifier, token endpoint URL, or pushed authorization request endpoint URL as values that identify it as an intended audience.

2.1. Request

A client can send all the parameters that usually comprise an authorization request to the pushed authorization request endpoint. A basic parameter set will typically include:

- o "client_id"
- o "response_type"
- o "redirect_uri"
- o "scope"
- o "state"

- o "code_challenge"
- o "code_challenge_method"

Depending on client type and authentication method, the request might also include other parameters for client authentication such as the "client_secret" parameter, the "client_assertion" parameter and the "client_assertion_type" parameter. The "request_uri" authorization request parameter MUST NOT be provided in this case (see Section 3).

The client adds the parameters in "x-www-form-urlencoded" format with a character encoding of UTF-8 as described in Appendix B of [RFC6749] to the body of an HTTP POST request. If applicable, the client also adds client credentials to the request header or the request body using the same rules as for token endpoint requests.

This is illustrated by the following example:

```
POST /as/par HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxS3REUmJuZlZkbUl3

response_type=code&
state=af0ifjsldkj&
client_id=s6BhdRkqt3&
redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb&
code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
code_challenge_method=S256&
scope=ais
```

The AS MUST process the request as follows:

1. The AS MUST authenticate the client in the same way as at the token endpoint.
2. The AS MUST reject the request if the "request_uri" authorization request parameter is provided.
3. The AS MUST validate the request in the same way as at the authorization endpoint. For example, the authorization server checks whether the redirect URI matches one of the redirect URIs configured for the client. It MUST also check whether the client is authorized for the "scope" for which it is requesting access. This validation allows the authorization server to refuse unauthorized or fraudulent requests early.

The AS MAY allow confidential clients to establish per-authorization request redirect URIs with every pushed authorization request. This is possible since, in contrast to [RFC6749], this specification gives the AS the ability to authenticate and authorize clients before the actual authorization request is performed.

This feature gives clients more flexibility in building redirect URIs and, if the client IDs and credentials are managed by some authority (CA or other type), the explicit client registration with the particular AS (manually or via dynamic client registration [RFC7591]) could practically be skipped. This makes this mechanism especially useful for clients interacting with a federation of ASs (or OpenID Connect OPs), for example in Open Banking, where the certificate is provided as part of a federated PKI.

2.2. Successful Response

If the verification is successful, the server MUST generate a request URI and return a JSON response that contains "request_uri" and "expires_in" members at the top level with "201 Created" HTTP response code.

- o "request_uri" : The request URI corresponding to the authorization request posted. This URI is used as reference to the respective request data in the subsequent authorization request only. The way the authorization process obtains the authorization request data is at the discretion of the authorization server and out of scope of this specification. There is no need to make the authorization request data available to other parties via this URI.
- o "expires_in" : A JSON number that represents the lifetime of the request URI in seconds. The request URI lifetime is at the discretion of the AS.

The "request_uri" value MUST be generated using a cryptographically strong pseudorandom algorithm such that it is computationally infeasible to predict or guess a valid value.

The "request_uri" MUST be bound to the client that posted the authorization request.

Since the request URI can be replayed, its lifetime SHOULD be short and preferably limited to one-time use.

The following is an example of such a response:

```
HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "request_uri": "urn:example:bwc4JK-ESC0w8acc191e-Y1LTC2",
  "expires_in": 3600
}
```

2.3. Error Response

For an error the authorization server sets an appropriate HTTP status code and MAY include additional error parameters in the entity-body of the HTTP response using the format specified for the token endpoint in Section 5.2 of [RFC6749].

If the authorization server sets an error code, it SHOULD be one of the defined codes for the token endpoint in Section 5.2 or for the authorization endpoint in Sections 4.1.2.1 and 4.2.2.1 of [RFC6749], or by an OAuth extension if one is involved in the initial processing of authorization request that was pushed. Since initial processing of the pushed authorisation request doesn't involve resource owner interaction, error codes related to user interaction, such as "consent_required" defined by [OIDC], are not returned.

In addition to the error codes above, the pushed authorization request endpoint specifies use of the following HTTP status codes:

- o 405: If the request did not use POST, the authorization server responds with an HTTP 405 (Method Not Allowed) status code.
- o 413: If the request size was beyond the upper bound that the authorization server allows, the authorization server responds with an HTTP 413 (Payload Too Large) status code.
- o 429: If the request from the client for a time period goes beyond the number the authorization server allows, the authorization server responds with an HTTP 429 (Too Many Requests) status code.

The following is an example of an error response from the pushed authorization request endpoint:

1. If applicable, the AS decrypts the request object as specified in JAR [I-D.ietf-oauth-jwsreq], section 6.1.
2. The AS validates the request object signature as specified in JAR [I-D.ietf-oauth-jwsreq], section 6.2.
3. If the client is a confidential client, the authorization server MUST check whether the authenticated "client_id" matches the "client_id" claim in the request object. If they do not match, the authorization server MUST refuse to process the request. It is at the authorization server's discretion to require the "iss" claim to match the "client_id" as well.

3.1. Error responses for Request Object

This section gives the error responses that go beyond the basic Section 2.3.

3.1.1. Authentication Required

If the signature validation fails, the authorization server returns a "401 Unauthorized" HTTP error response. The same applies if the "client_id" or, if applicable, the "iss" claim in the request object do not match the authenticated "client_id".

4. Authorization Request

The client uses the "request_uri" value returned by the authorization server as the authorization request parameter "request_uri" as defined in JAR.

```
GET /authorize?request_uri=
  urn%3Aexample%3Abwc4JK-ESC0w8acc191e-Y1LTC2 HTTP/1.1
```

Clients are encouraged to use the request URI as the only parameter in order to use the integrity and authenticity provided by the pushed authorization request.

5. Authorization Server Metadata

If the authorization server has a pushed authorization request endpoint, it SHOULD include the following OAuth/OpenID Provider Metadata parameter in discovery responses:

"pushed_authorization_request_endpoint" : The URL of the pushed authorization request endpoint at which the client can post an authorization request and get a request URI in exchange.

6. Security Considerations

6.1. Request URI Guessing

An attacker could attempt to guess and replay a valid request URI value and try to impersonate the respective client. The AS MUST consider the considerations given in JAR [I-D.ietf-oauth-jwsreq], section 10.2, clause d on request URI entropy.

6.2. Open Redirection

An attacker could try register a redirect URI pointing to a site under his control in order to obtain authorization codes or launch other attacks towards the user. The AS MUST only accept new redirect URIs in the PAR request from confidential clients after successful authentication and authorization.

6.3. Request Object Replay

An attacker could replay a request URI captured from a legitimate authorization request. In order to cope with such attacks, the AS SHOULD make the request URIs one-time use.

6.4. Client Policy Change

The client policy might change between the lodging of the request object and the authorization request using a particular request object. It is therefore recommended that the AS check the request parameter against the client policy when processing the authorization request.

7. Acknowledgements

This specification is based on the work towards Pushed Request Object [1] conducted at the Financial-grade API working group at the OpenID Foundation. We would like to thank the members of the WG for their valuable contributions.

We would like to thank Vladimir Dzhuvinov, Aaron Parecki, Joseph Heenan, and Takahiko Kawasaki for their valuable feedback on this draft.

8. IANA Considerations

...

9. References

9.1. Normative References

- [I-D.ietf-oauth-jwsreq] Sakimura, N. and J. Bradley, "The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request (JAR)", draft-ietf-oauth-jwsreq-20 (work in progress), October 2019.
- [OIDC] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", Nov 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.
- [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>>.
- [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>>.
- [RFC7617] Reschke, J., "The 'Basic' HTTP Authentication Scheme", RFC 7617, DOI 10.17487/RFC7617, September 2015, <<https://www.rfc-editor.org/info/rfc7617>>.
- [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>>.
- [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>>.

9.2. Informative References

- [I-D.ietf-oauth-resource-indicators] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-08 (work in progress), September 2019.
- [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>>.

[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>>.

9.3. URIs

[1] https://bitbucket.org/openid/fapi/src/master/Financial_API_Pushed_Request_Object.md

Appendix A. Document History

[[To be removed from the final specification]]

-01

- o List "client_id" as one of the basic parameters
- o Explicitly forbid "request_uri" in the processing rules
- o Clarification regarding client authentication and that public clients are allowed
- o Added option to let clients register per-authorization request redirect URIs
- o General clean up and wording improvements

-00

- o first draft

Authors' Addresses

Torsten Lodderstedt
yes.com

Email: torsten@lodderstedt.net

Brian Campbell
Ping Identity

Email: bcampbell@pingidentity.com

Nat Sakimura
Nomura Research Institute

Email: nat@sakimura.org

Dave Tonge
Moneyhub Financial Technology

Email: dave@tonge.org

Filip Skokan
Auth0

Email: panva.ip@gmail.com

Web Authorization Protocol
Internet-Draft
Intended status: Standards Track
Expires: May 6, 2020

T. Lodderstedt
yes.com
J. Richer
Bespoke Engineering
B. Campbell
Ping Identity
November 3, 2019

OAuth 2.0 Rich Authorization Requests
draft-lodderstedt-oauth-rar-03

Abstract

This document specifies a new parameter "authorization_details" that is used to carry fine grained authorization data in the OAuth authorization request.

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 May 6, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

| | |
|--|----|
| 1. Introduction | 2 |
| 1.1. Conventions and Terminology | 4 |
| 2. Request parameter "authorization_details" | 4 |
| 2.1. Authorization data elements types | 5 |
| 2.2. Relationship to "scope" parameter | 8 |
| 2.2.1. Scope value "openid" and "claims" parameter | 8 |
| 2.3. Relationship to "resource" parameter | 9 |
| 3. Using "authorization_details" | 11 |
| 3.1. Authorization Request | 11 |
| 3.2. Authorization Request Processing | 14 |
| 3.3. Token Request | 15 |
| 3.4. Token Response | 15 |
| 3.4.1. Token Content | 16 |
| 3.5. Token Introspection Request | 18 |
| 3.6. Token Introspection Response | 18 |
| 4. Metadata | 19 |
| 5. Implementation Considerations | 20 |
| 6. Security Considerations | 20 |
| 7. Privacy Considerations | 20 |
| 8. Acknowledgements | 21 |
| 9. IANA Considerations | 21 |
| 10. References | 21 |
| 10.1. Normative References | 21 |
| 10.2. Informative References | 22 |
| Appendix A. Additional Examples | 23 |
| A.1. OpenID Connect | 23 |
| A.2. Remote Electronic Signing | 25 |
| A.3. Access to Tax Data | 26 |
| A.4. eHealth | 27 |
| Appendix B. Document History | 29 |
| Authors' Addresses | 30 |

1. Introduction

The OAuth 2.0 authorization framework [RFC6749] defines the parameter "scope" that allows OAuth clients to specify the requested scope, i.e., the permission, of an access token. This mechanism is sufficient to implement static scenarios and coarse-grained authorization requests, such as "give me read access to the resource owner's profile" but it is not sufficient to specify fine-grained authorization requirements, such as "please let me make a payment with the amount of 45 Euros" or "please give me read access to folder A and write access to file X".

This draft introduces a new parameter "authorization_details" that allows clients to specify their fine-grained authorization requirements using the expressiveness of JSON data structures.

For example, a request for payment authorization can be represented using a JSON object like this:

```
{
  "type": "payment_initiation",
  "locations": [
    "https://example.com/payments"
  ],
  "instructedAmount": {
    "currency": "EUR",
    "amount": "123.50"
  },
  "creditorName": "Merchant123",
  "creditorAccount": {
    "iban": "DE02100100109307118603"
  },
  "remittanceInformationUnstructured": "Ref Number Merchant"
}
```

This object contains detailed information about the intended payment, such as amount, currency, and creditor, that are required to inform the user and obtain her consent. The AS and the respective RS (providing the payment initiation API) will together enforce this consent.

For a comprehensive discussion of the challenges arising from new use cases in the open banking and electronic signing spaces see [transaction-authorization].

In addition to facilitating custom authorization requests, this draft also introduces a set of common data type fields for use across different APIs.

Most notably, the field "locations" allows a client to specify where it intends to use a certain authorization, i.e., it is now possible to unambiguously assign permissions to resource servers. In situations with multiple resource servers, this prevents unintended client authorizations (e.g. a "read" scope value potentially applicable for an email as well as a cloud service). In combination with the "resource" token request parameter as specified in [I-D.ietf-oauth-resource-indicators] it enables the AS to mint RS-specific structured access tokens that only contain the permissions applicable to the respective RS.

1.1. Conventions and Terminology

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.

This specification uses the terms "access token", "refresh token", "authorization server", "resource server", "authorization endpoint", "authorization request", "authorization response", "token endpoint", "grant type", "access token request", "access token response", and "client" defined by The OAuth 2.0 Authorization Framework [RFC6749].

2. Request parameter "authorization_details"

The request parameter "authorization_details" contains, in JSON notation, an array of objects. Each JSON object contains the data to specify the authorization requirements for a certain type of resource. The type of resource or access requirement is determined by the "type" field.

This example shows the specification of authorization details using the payment authorization object shown above:

```
[
  {
    "type": "payment_initiation",
    "actions": [
      "initiate",
      "status",
      "cancel"
    ],
    "locations": [
      "https://example.com/payments"
    ],
    "instructedAmount": {
      "currency": "EUR",
      "amount": "123.50"
    },
    "creditorName": "Merchant123",
    "creditorAccount": {
      "iban": "DE02100100109307118603"
    },
    "remittanceInformationUnstructured": "Ref Number Merchant"
  }
]
```

This example shows a combined request asking for access to account information and permission to initiate a payment:

```
[
  {
    "type": "account_information",
    "actions": [
      "list_accounts",
      "read_balances",
      "read_transactions"
    ],
    "locations": [
      "https://example.com/accounts"
    ]
  },
  {
    "type": "payment_initiation",
    "actions": [
      "initiate",
      "status",
      "cancel"
    ],
    "locations": [
      "https://example.com/payments"
    ],
    "instructedAmount": {
      "currency": "EUR",
      "amount": "123.50"
    },
    "creditorName": "Merchant123",
    "creditorAccount": {
      "iban": "DE02100100109307118603"
    },
    "remittanceInformationUnstructured": "Ref Number Merchant"
  }
]
```

The JSON objects with "type" fields of "account_information" and "payment_initiation" represent the different authorization data to be used by the AS to ask for consent and MUST subsequently also be made available to the respective resource servers. The array MAY contain several elements of the same "type".

2.1. Authorization data elements types

This draft defines a set of common data elements that are designed to be usable across different types of APIs. These data elements MAY be

combined in different ways depending on the needs of the API. Unless otherwise noted, all data elements are OPTIONAL.

type:

The type of resource request as a string. This field MAY define which other elements are allowed in the request. This element is REQUIRED.

locations:

An array of strings representing the location of the resource or resource server. This is typically composed of URIs.

actions:

An array of strings representing the kinds of actions to be taken at the resource. The values of the strings are determined by the API being protected.

datatypes:

An array of strings representing the kinds of data being requested from the resource.

identifier:

A string identifier indicating a specific resource available at the API.

When different element types are used in combination, the permissions the client requests is the cartesian product of the values. In the following example

```
[
  {
    "type": "customer_information",
    "locations": [
      "https://example.com/customers",
    ],
    "actions": [
      "read",
      "write"
    ],
    "datatypes": [
      "contacts",
      "photos"
    ]
  }
]
```

the client is requesting read and write access to both the contacts and photos belonging to customers in a customer information API. If the client wishes to have finer control over its access, it can send multiple objects. For example:

```
[
  {
    "type": "customer_information",
    "locations": [
      "https://example.com/customers"
    ],
    "actions": [
      "read"
    ],
    "datatypes": [
      "contacts"
    ]
  },
  {
    "type": "customer_information",
    "locations": [
      "https://example.com/customers"
    ],
    "actions": [
      "write"
    ],
    "datatypes": [
      "photos"
    ]
  }
]
```

The client is asking for read access to the contacts and write access to the photos in the same API endpoint.

An API MAY define its own extensions, subject to the "type" of the respective authorization object. It is assumed that the full structure of each of the authorization objects is tailored to the needs of a certain application, API, or resource type. The example structures shown above are based on certain kinds of APIs that can be found in the Open Banking space.

Note: Applications MUST ensure that their authorization data types do not collide. This is either achieved by using a namespace under the control of the entity defining the type name or by registering the type with the new "OAuth Authorization Data Type Registry" (see Section 9).

The following example shows how an implementation could utilize the namespace "https://scheme.example.org/" to ensure collision resistant element names.

```
{
  "type": "https://scheme.example.org/files",
  "locations": [
    "https://example.com/files"
  ],
  "permissions": [
    {
      "path": "/myfiles/A",
      "access": [
        "read"
      ]
    },
    {
      "path": "/myfiles/A/X",
      "access": [
        "read",
        "write"
      ]
    }
  ]
}
```

2.2. Relationship to "scope" parameter

"authorization_details" and "scope" can be used in the same authorization request for carrying independent authorization requirements.

The AS MUST consider both sets of requirements in combination with each other for the given authorization request. The details of how the AS combines these parameters are specific to the APIs being protected and outside the scope of this specification.

It is RECOMMENDED that a given API use only one form of requirement specification.

When gathering user consent, the AS MUST present the merged set of requirements represented by the authorization request.

2.2.1. Scope value "openid" and "claims" parameter

OpenID Connect [OIDC] specifies the JSON-based "claims" request parameter that can be used to specify the claims a client (acting as OpenID Connect Relying Party) wishes to receive in a fine-grained and

privacy preserving way as well as assign those claims to a certain delivery mechanisms, i.e. ID Token or userinfo response.

The combination of the scope value "openid" and the additional parameter "claims" can be used beside "authorization_details" in the same way as every non-OIDC scope value.

Alternatively, there could be an authorization data type for OpenID Connect. Appendix A.1 gives an example of how such an authorization data type could look like.

2.3. Relationship to "resource" parameter

The request parameter "resource" as defined in [I-D.ietf-oauth-resource-indicators] indicates to the AS the resource(s) where the client intends to use the access tokens issued based on a certain grant. This mechanism is a way to audience-restrict access tokens and to allow the AS to create resource server specific access tokens.

If a client uses "authorization_details" with "locations" elements and the "resource" parameter in the same authorization request, the "locations" data take precedence over the data conveyed in the "resource" parameter for that particular authorization details object.

If such a client uses the "resource" parameter in a subsequent token requests, the AS MUST utilize the data provided in the "locations" elements to filter the authorization data objects applicable to the respective resource server. The AS will select all authorization details object where the "resource" string matches as prefix of one of the URLs provided in the respective "locations" element.

This shall be illustrated using an example.

The client has sent an authorization request using the following example authorization details.

```
[
  {
    "type": "account_information",
    "actions": [
      "list_accounts",
      "read_balances",
      "read_transactions"
    ],
    "locations": [
      "https://example.com/accounts"
    ]
  },
  {
    "type": "payment_initiation",
    "actions": [
      "initiate",
      "status",
      "cancel"
    ],
    "locations": [
      "https://example.com/payments"
    ],
    "instructedAmount": {
      "currency": "EUR",
      "amount": "123.50"
    },
    "creditorName": "Merchant123",
    "creditorAccount": {
      "iban": "DE02100100109307118603"
    },
    "remittanceInformationUnstructured": "Ref Number Merchant"
  }
]
```

If this client then sends the following token request to the AS,

```
POST /token HTTP/1.1
Host: as.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=Sp1xl0BeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&resource=https%3A%2F%2Fexample%2Ecom%2Fpayments
```

that contains a resource parameter with the value of "https://example.com/payments", this value will be matched against the locations elements ("https://example.com/accounts" and

"https://example.com/payments") and will select the element of type "payment_initiation" for inclusion in the access token as illustrated by the following example JWT content.

```
{
  "iss": "https://as.example.com",
  "sub": "24400320",
  "aud": "a7AfcPcsl2",
  "exp": 1311281970,
  ...
  "authorization_details": [
    {
      "type": "https://www.someorg.com/payment_initiation",
      "actions": [
        "initiate",
        "status",
        "cancel"
      ],
      "locations": [
        "https://example.com/payments"
      ],
      "instructedAmount": {
        "currency": "EUR",
        "amount": "123.50"
      },
      "creditorName": "Merchant123",
      "creditorAccount": {
        "iban": "DE02100100109307118603"
      },
      "remittanceInformationUnstructured": "Ref Number Merchant"
    }
  ],
  ...
}
```

3. Using "authorization_details"

3.1. Authorization Request

The request parameter can be used to specify authorization requirements in all places where the "scope" parameter is used for the same purpose, examples include:

- o Authorization requests as specified in [RFC6749],
- o Access token requests as specified in [RFC6749], if also used as authorization requests, e.g. in the case of assertion grant types [RFC7521],

- o Request objects as specified in [I-D.ietf-oauth-jwsreq],
- o Device Authorization Request as specified in [RFC8628],
- o Backchannel Authentication Requests as defined in [OpenID.CIBA].

Parameter encoding is determined by the respective context.

In the context of an authorization request according to [RFC6749], the parameter is encoded using the "application/x-www-form-urlencoded" format of the serialized JSON as shown in the following example:

```
GET /authorize?response_type=code
&client_id=s6BhdRkqt3
&state=af0ifjsldkj
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&code_challenge_method=S256
&code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U
&authorization_details=%5B%7B%22type%22%3A%22account%5Finformati
on%22%2C%22actions%22%3A%5B%22list%5Faccounts%22%2C%22read%5Fbal
ances%22%2C%22read%5Ftransactions%22%5D%2C%22locations%22%3A%5B%
22https%3A%2F%2Fexample%2Ecom%2Faccounts%22%5D%7D%5D HTTP/1.1
Host: server.example.com
```

Implementors MUST ensure to protect personal identifiable information in transit. One way is to utilize encrypted request objects as defined in [I-D.ietf-oauth-jwsreq]. In the context of a request object, "authorization_details" is added as another top level JSON element.

```
{
  "iss": "s6BhdRkqt3",
  "aud": "https://server.example.com",
  "response_type": "code",
  "client_id": "s6BhdRkqt3",
  "redirect_uri": "https://client.example.com/cb",
  "state": "af0ifjsldkj",
  "code_challenge_method": "S256",
  "code_challenge": "K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U",
  "authorization_details": [
    {
      "type": "account_information",
      "actions": [
        "list_accounts",
        "read_balances",
        "read_transactions"
      ],
      "locations": [
        "https://example.com/accounts"
      ]
    },
    {
      "type": "payment_initiation",
      "actions": [
        "initiate",
        "status",
        "cancel"
      ],
      "locations": [
        "https://example.com/payments"
      ],
      "instructedAmount": {
        "currency": "EUR",
        "amount": "123.50"
      },
      "creditorName": "Merchant123",
      "creditorAccount": {
        "iban": "DE02100100109307118603"
      },
      "remittanceInformationUnstructured": "Ref Number Merchant"
    }
  ]
}
```

Authorization request URIs containing authorization details in a request parameter or a request object can become very long. Implementers SHOULD therefore consider using the "request_uri" parameter as defined in [I-D.ietf-oauth-jwsreq] in combination with

the pushed request object mechanism as defined in [I-D.lodderstedt-oauth-par] to pass authorization details in a reliable and secure manner. Here is an example of such a pushed authorization request that sends the authorization request data directly to the AS via a HTTPS-protected connection:

```
POST /as/par HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxS3REUmJuZlZkbUl3

response_type=code&
client_id=s6BhdRkqt3
&state=af0ifjsldkj
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&code_challenge_method=S256
&code_challenge=K2-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U
&authorization_details=%7B%22iss%22%3A%22s6BhdRkqt3%22%2C%22aud%22%
3A%22https%3A%2F%2Fserver%2Eexample%2Ecom%22%2C%22response%5Ftype%2
2%3A%22code%22%2C%22client%5Fid%22%3A%22s6BhdRkqt3%22%2C%22redirect
%5Furi%22%3A%22https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb%22%2C%22st
ate%22%3A%22af0ifjsldkj%22%2C%22code%5Fchallenge%5Fmethod%22%3A%22S
256%22%2C%22code%5Fchallenge%22%3A%22K2%2Dltc83acc4h0c9w6ESC%5FrEMT
J3bww%2DuCHaoeK1t8U%22%2C%22authorization%5Fdetails%22%3A%5B%7B%22t
ype%22%3A%22account%5Finformation%22%2C%22actions%22%3A%5B%22list%5
Faccounts%22%2C%22read%5Fbalances%22%2C%22read%5Ftransactions%22%5D
%2C%22locations%22%3A%5B%22https%3A%2F%2Fexample%2Ecom%2Faccounts%2
2%5D%7D%2C%7B%22type%22%3A%22payment%5Finitiation%22%2C%22actions%2
2%3A%5B%22initiate%22%2C%22status%22%2C%22cancel%22%5D%2C%22locatio
ns%22%3A%5B%22https%3A%2F%2Fexample%2Ecom%2Fpayments%22%5D%2C%22ins
tructedAmount%22%3A%7B%22currency%22%3A%22EUR%22%2C%22amount%22%3A%
22123%2E50%22%7D%2C%22creditorName%22%3A%22Merchant123%22%2C%22cred
itorAccount%22%3A%7B%22iban%22%3A%22DE02100100109307118603%22%7D%2C
%22remittanceInformationUnstructured%22%3A%22Ref%20Number%20Merchan
t%22%7D%5D%7D
```

3.2. Authorization Request Processing

Based on the data provided in the "authorization_details" parameter the AS will ask the user for consent to the requested access permissions.

The AS MUST refuse to process any unknown authorization data type. If the "authorization_details" contain any unknown authorization data type, the AS MUST abort processing and respond with an error "invalid_authorization_details" to the client.

Note: If the authorization request also contained the "scope" parameter, the AS MUST present the merged set of requirements represented by the authorization request in the user consent.

If the resource owner grants the client the requested access, the AS will issue tokens to the client that are associated with the respective "authorization_details" (and scope values, if applicable).

Note: The AS MUST make the "authorization_details" available to the respective resource servers. The AS MAY add the "authorization_details" element to access tokens in JWT format and to Token Introspection responses (see below).

3.3. Token Request

Clients utilizing authorization details are RECOMMENDED to use the "resource" token request parameter to allow the AS to issue audience restricted access tokens as recommended in [I-D.ietf-oauth-security-topics].

For example the following token request selects authorization details applicable for the resource server represented by the URI "https://example.com/payments".

```
POST /token HTTP/1.1
Host: as.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=Sp1xl0BeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&resource=https%3A%2F%2Fexample%2Ecom%2Fpayments
```

3.4. Token Response

In addition to the token response parameters as defined in [RFC6749], the authorization server MUST also return the authorization details as granted by the resource owner and assigned to the respective access token.

This is shown in the following example:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "access_token": "2YotnFZFEjrlzCsicMWpAA",
  "token_type": "example",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA",
  "authorization_details": [
    {
      "type": "https://www.someorg.com/payment_initiation",
      "actions": [
        "initiate",
        "status",
        "cancel"
      ],
      "locations": [
        "https://example.com/payments"
      ],
      "instructedAmount": {
        "currency": "EUR",
        "amount": "123.50"
      },
      "creditorName": "Merchant123",
      "creditorAccount": {
        "iban": "DE02100100109307118603"
      },
      "remittanceInformationUnstructured": "Ref Number Merchant"
    }
  ]
}
```

3.4.1. Token Content

In order to enable the RS to enforce the authorization details as approved in the authorization process, the AS MUST make this data available to the RS.

If the access token is a JWT [RFC7519], the AS is RECOMMENDED to add the "authorization_details" object, filtered to the specific audience, as top-level claim.

The AS will typically also add further claims to the JWT the RS requires for request processing, e.g., user id, roles, and transaction specific data. What claims the particular RS requires is defined by the RS-specific policy with the AS.

The following shows the contents of an example JWT for the payment initiation example above:

```
{
  "iss": "https://as.example.com",
  "sub": "24400320",
  "aud": "a7AfcPcsl2",
  "exp": 1311281970,
  "acr": "psd2_sca",
  "txn": "8b4729cc-32e4-4370-8cf0-5796154d1296",
  "authorization_details": [
    {
      "type": "https://www.someorg.com/payment_initiation",
      "actions": [
        "initiate",
        "status",
        "cancel"
      ],
      "locations": [
        "https://example.com/payments"
      ],
      "instructedAmount": {
        "currency": "EUR",
        "amount": "123.50"
      },
      "creditorName": "Merchant123",
      "creditorAccount": {
        "iban": "DE02100100109307118603"
      },
      "remittanceInformationUnstructured": "Ref Number Merchant"
    }
  ],
  "debtorAccount": {
    "iban": "DE40100100103307118608",
    "user_role": "owner"
  }
}
```

In this case, the AS added the following example claims:

- o "sub": conveys the user on which behalf the client is asking for payment initiation
- o "txn": transaction id used to trace the transaction across the services of provider "example.com"
- o "debtorAccount": API-specific element containing the debtor account. In the example, this account was not passed in the

authorization details but selected by the user during the authorization process. The field "user_role" conveys the role the user has with respect to this particular account. In this case, she is the owner. This data is used for access control at the payment API (the RS).

3.5. Token Introspection Request

In case of opaque access tokens, the data provided to a certain RS is determined using the RS's identifier with the AS (see [I-D.ietf-oauth-jwt-introspection-response], section 3).

3.6. Token Introspection Response

The token endpoint response provides the RS with the authorization details applicable to it as a top-level JSON element along with the claims the RS requires for request processing.

Here is an example for the payment initiation example RS:

```
{
  "active": true,
  "sub": "24400320",
  "aud": "s6BhdRkqt3",
  "exp": 1311281970,
  "acr": "psd2_sca",
  "txn": "8b4729cc-32e4-4370-8cf0-5796154d1296",
  "authorization_details": [
    {
      "type": "https://www.someorg.com/payment_initiation",
      "actions": [
        "initiate",
        "status",
        "cancel"
      ],
      "locations": [
        "https://example.com/payments"
      ],
      "instructedAmount": {
        "currency": "EUR",
        "amount": "123.50"
      },
      "creditorName": "Merchant123",
      "creditorAccount": {
        "iban": "DE02100100109307118603"
      },
      "remittanceInformationUnstructured": "Ref Number Merchant"
    }
  ],
  "debtorAccount": {
    "iban": "DE40100100103307118608",
    "user_role": "owner"
  }
}
```

4. Metadata

The AS advertises support for "authorization_details" using the metadata parameter "authorization_details_supported" of type boolean.

The authorization data types supported can be determined using the metadata parameter "authorization_data_types_supported", which is an JSON array.

Clients announce the authorization data types they use in the new dynamic client registration parameter "authorization_data_types".

The registration of new authorization data types with the AS is out of scope of this draft.

5. Implementation Considerations

The scheme and processing will vary significantly among different authorization data types. Any implementation of this draft is therefore supposed to allow the customization of the user consent and the handling of access token data.

One option would be to have a mechanism allowing the registration of extension modules, each of them responsible for rendering the respective user consent and any transformation needed to provide the data needed to the resource server by way of structured access tokens or token introspection responses.

6. Security Considerations

Authorization details are sent through the user agent in case of an OAuth authorization request, which makes them vulnerable to modifications by the user. In order to ensure their integrity, the client SHOULD send authorization details in a signed request object as defined in [I-D.ietf-oauth-jwsreq] or use the "request_uri" authorization request parameter as defined in [I-D.ietf-oauth-jwsreq] to pass the URI of the request object to the authorization server.

All strings MUST be compared using the exact byte representation of the characters as defined by [RFC8259]. This is especially true for the "type" field, which dictates which other fields and functions are allowed in the request. The server MUST NOT perform any form of collation, transformation, or equivalence on the string values.

7. Privacy Considerations

Implementers MUST design and use authorization details in a privacy preserving manner.

Any sensitive personal data included in authorization details MUST be prevented from leaking, e.g., through referrer headers. Implementation options include encrypted request objects as defined in [I-D.ietf-oauth-jwsreq] or transmission of authorization details via end-to-end encrypted connections between client and authorization server by utilizing the "request_uri" authorization request parameter as defined in [I-D.ietf-oauth-jwsreq].

Even if the request data are encrypted, an attacker could use the authorization server to learn the user data by injecting the encrypted request data into an authorization request on a device

under his control and use the authorization server's user consent screens to show the (decrypted) user data in the clear. Implementations MUST consider this attacker vector and implement appropriate counter measures, e.g. by only showing portions of the data or, if possible, determining whether the assumed user context is still the same (after user authentication).

The AS MUST take into consideration the privacy implications when sharing authorization details with the resource servers. The AS SHOULD share this data with the resource servers on a "need to know" basis.

8. Acknowledgements

We would would like to thank Daniel Fett, Sebastian Ebling, Dave Tonge, Mike Jones, Nat Sakimura, and Rob Otto for their valuable feedback during the preparation of this draft.

We would also like to thank Daniel Fett, Dave Tonge, Travis Spencer, Joergen Binningsboe, Aamund Bremer, Steinar Noem, and Aaron Parecki for their valuable feedback to this draft.

9. IANA Considerations

TBD

- o "authorization_details" as JWT claim
- o "authorization_details_supported" and "authorization_data_types_supported" as metadata parameters
- o "authorization_data_types" as dynamic client registration parameter
- o establish authorization data type registry
- o register type "openid_claims"

10. References

10.1. 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>>.

- [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>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<https://www.rfc-editor.org/info/rfc7519>>.
- [RFC7521] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <<https://www.rfc-editor.org/info/rfc7521>>.
- [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>>.
- [RFC8628] Denniss, W., Bradley, J., Jones, M., and H. Tschofenig, "OAuth 2.0 Device Authorization Grant", RFC 8628, DOI 10.17487/RFC8628, August 2019, <<https://www.rfc-editor.org/info/rfc8628>>.

10.2. Informative References

- [CSC] Consortium, C. S., "Architectures and protocols for remote signature applications", Jun 2019, <https://cloudsignatureconsortium.org/wp-content/uploads/2019/07/CSC_API_V1_1.0.4.0.pdf>.
- [ETSI] ETSI, "ETSI TS 119 432, Electronic Signatures and Infrastructures (ESI); Protocols for remote digital signature creation", Mar 2019, <https://www.etsi.org/deliver/etsi_ts/119400_119499/119432/01.01.01_60/ts_119432v010101p.pdf>.
- [I-D.ietf-oauth-jwsreq] Sakimura, N. and J. Bradley, "The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request (JAR)", draft-ietf-oauth-jwsreq-20 (work in progress), October 2019.
- [I-D.ietf-oauth-jwt-introspection-response] Lodderstedt, T. and V. Dzhuvinov, "JWT Response for OAuth Token Introspection", draft-ietf-oauth-jwt-introspection-response-08 (work in progress), September 2019.

- [I-D.ietf-oauth-resource-indicators]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-08 (work in progress), September 2019.
- [I-D.ietf-oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", draft-ietf-oauth-security-topics-13 (work in progress), July 2019.
- [I-D.lodderstedt-oauth-par]
Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D., and F. Skokan, "OAuth 2.0 Pushed Authorization Requests", draft-lodderstedt-oauth-par-00 (work in progress), September 2019.
- [OIDC]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", Nov 2014,
<http://openid.net/specs/openid-connect-core-1_0.html>.
- [OpenID.CIBA]
Fernandez, G., Walter, F., Nennker, A., Tonge, D., and B. Campbell, "OpenID Connect Client Initiated Backchannel Authentication Flow - Core 1.0", January 2019,
<https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html>.
- [RFC8259]
Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017,
<<https://www.rfc-editor.org/info/rfc8259>>.
- [transaction-authorization]
Lodderstedt, T., "Transaction Authorization or why we need to re-think OAuth scopes", Apr 2019, <<https://medium.com/oauth-2/transaction-authorization-or-why-we-need-to-re-think-oauth-scopes-2326e2038948>>.

Appendix A. Additional Examples

A.1. OpenID Connect

These hypothetical examples try to encapsulate all details specific to the OpenID Connect part of an authorization process into an authorization JSON object.

The top-level elements are based on the definitions given in [OIDC]:

- o "claim_sets": names of predefined claim sets, replacement for respective scope values, such as "profile"
- o "max_age": Maximum Authentication Age
- o "acr_values": array of ACR values
- o "claims": the "claims" JSON structure as defined in [OIDC]

This is a simple request for some claim sets.

```
[
  {
    "type": "openid",
    "locations": [
      "https://op.example.com/userinfo"
    ],
    "claim_sets": [
      "email",
      "profile"
    ]
  }
]
```

Note: "locations" specifies the location of the userinfo endpoint since this is the only place where an access token is used by a client (RP) in OpenID Connect to obtain claims.

A more sophisticated example is shown in the following

```
[
  {
    "type": "openid",
    "locations": [
      "https://op.example.com/userinfo"
    ],
    "max_age": 86400,
    "acr_values": "urn:mace:incommon:iap:silver",
    "claims": {
      "userinfo": {
        "given_name": {
          "essential": true
        },
        "nickname": null,
        "email": {
          "essential": true
        },
        "email_verified": {
          "essential": true
        },
        "picture": null,
        "http://example.info/claims/groups": null
      },
      "id_token": {
        "auth_time": {
          "essential": true
        }
      }
    }
  }
]
```

A.2. Remote Electronic Signing

The following example is based on the concept layed out for remote electronic signing in ETSI TS 119 432 [ETSI] and the CSC API for remote signature creation [CSC].

```
[
  {
    "type": "sign",
    "locations": [
      "https://signing.example.com/signdoc"
    ],
    "credentialID": "60916d31-932e-4820-ba82-1fcead1c9ea3",
    "documentDigests": [
      {
        "hash": "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
        "label": "Credit Contract"
      },
      {
        "hash": "HZQzZmMAIWekfGH0/ZKWlnsdt0xg3H6bZYztgsMTLw0=",
        "label": "Contract Payment Protection Insurance"
      }
    ],
    "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1"
  }
]
```

The top-level elements have the following meaning:

- o "credentialID": identifier of the certificate to be used for signing
- o "documentDigests": array containing the hash of every document to be signed ("hash" elements). Additionally, the corresponding "label" element identifies the respective document to the user, e.g. to be used in user consent.
- o "hashAlgorithm": algorithm that was used to calculate the hash values.

The AS is supposed to ask the user for consent for the creation of signatures for the documents listed in the structure. The client uses the access token issued as result of the process to call the sign doc endpoint at the respective signing service to actually create the signature. This access token is bound to the client, the user id and the hashes (and signature algorithm) as consented by the user.

A.3. Access to Tax Data

This example is inspired by an API allowing third parties to access citizen's tax declarations and income statements, for example to determine their credit worthiness.

```
[
  {
    "type": "tax_data",
    "locations": [
      "https://taxservice.govehub.no"
    ],
    "actions": "read_tax_declaration",
    "periods": ["2018"],
    "duration_of_access": 30,
    "tax_payer_id": "23674185438934"
  }
]
```

The top-level elements have the following meaning:

- o "periods": determines the periods the client wants to access
- o "duration_of_access": how long does the client intend to access the data in days
- o "tax_payer_id": identifier of the tax payer (if known to the client)

A.4. eHealth

This example is inspired by an API used in the Norwegian eHealth system.

In this use case the physical therapist sits in front of her computer using a local Electronic Health Records (EHR) system. She wants to look at the electronic patient records of a certain patient and she also wants to fetch the patients journal entries in another system, perhaps at another institution or a national service. Access to this data is provided by an API.

The information necessary to authorize the request at the API is only known by the EHR system, and must be presented to the API.

Here is an example authorization details object:

```
[
  {
    "type": "patient_record",
    "location": "https://fhir.example.com/patient",
    "actions": [
      "read"
    ],
    "patient_identifier": [
```

```

        {
            "system": "urn:oid:2.16.578.1.12.4.1.4.1",
            "value": "12345678901"
        }
    ],
    "reason_for_request": "Clinical treatment",
    "requesting_entity": {
        "type": "Practitioner",
        "practitioner_identifier": [
            {
                "system": "urn:oid:2.16.578.1.12.4.1.4.4",
                "value": "1234567"
            }
        ],
        "practitioner_role": {
            "organization": {
                "organization_identifier": [
                    {
                        "system": "urn:oid:2.16.578.1.12.4.1.2.101",
                        "value": "<organizational number>"
                    }
                ],
                "organization_type": {
                    "coding": [
                        {
                            "system":
                                "http://hl7.org/fhir/organization-type",
                            "code": "dept",
                            "display": "Hospital Department"
                        }
                    ]
                },
                "name": "Akuttmottak"
            },
            "role": {
                "coding": [
                    {
                        "system": "http://snomed.info/sct",
                        "code": "36682004",
                        "display": "Physical therapist"
                    }
                ]
            }
        }
    }
}
]

```

Description of the elements:

- o "patient_identifier": the identifier of the patient composed of a system identifier in OID format (namespace) and the actual value within this namespace.
- o "reason_for_request": the reason why the user wants to access a certain API
- o "requesting_entity": specification of the requester by means of identity, role and organizational context. This data is provided to facilitate authorization and for auditing purposes.

In this use case, the AS authenticates the requester, who is not the patient, and approves access based on policies.

Appendix B. Document History

[[To be removed from the final specification]]

-03

- o Reworked examples to illustrate privacy preserving use of "authorization_details"
- o Added text on audience restriction
- o Added description of relationship between "scope" and "authorization_details"
- o Added text on token request & response and "authorization_details"
- o Added text on how authorization details are conveyed to RSs by JWTs or token endpoint response
- o Added description of relationship between "claims" and "authorization_details"
- o Added more example from different sectors
- o Clarified string comparison to be byte-exact without collation

-02

- o Added Security Considerations
- o Added Privacy Considerations

- o Added notes on URI size and authorization details
- o Added requirement to return the effective authorization details granted by the resource owner in the token response
- o changed "authorization_details" structure from object to array
- o added Justin Richer & Brian Campbell as Co-Authors

-00 / -01

- o first draft

Authors' Addresses

Torsten Lodderstedt
yes.com

Email: torsten@lodderstedt.net

Justin Richer
Bespoke Engineering

Email: ietf@justin.richer.org

Brian Campbell
Ping Identity

Email: bcampbell@pingidentity.com

GNAP
Internet-Draft
Intended status: Standards Track
Expires: 12 April 2021

J. Richer, Ed.
Bespoke Engineering
9 October 2020

Grant Negotiation and Authorization Protocol
draft-richer-transactional-authz-14

Abstract

This document defines a mechanism for delegating authorization to a piece of software, and conveying that delegation to the software. This delegation can include access to a set of APIs as well as information passed directly to the software.

This document has been prepared by the GNAP working group design team of Kathleen Moriarty, Fabien Imbault, Dick Hardt, Mike Jones, and Justin Richer. This document is intended as a starting point for the working group and includes decision points for discussion and agreement. Many of the features in this proposed protocol can be accomplished in a number of ways. Where possible, the editor has included notes and discussion from the design team regarding the options as understood.

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.

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 12 April 2021.

| | |
|---|-----|
| 10.3. Registering a Resource Handle | 97 |
| 10.4. Requesting a Resources With Insufficient Access | 98 |
| 11. Acknowledgements | 98 |
| 12. IANA Considerations | 99 |
| 13. Security Considerations | 99 |
| 14. Privacy Considerations | 99 |
| 15. Normative References | 99 |
| Appendix A. Document History | 101 |
| Appendix B. Component Data Models | 105 |
| Appendix C. Example Protocol Flows | 105 |
| C.1. Redirect-Based User Interaction | 106 |
| C.2. Secondary Device Interaction | 110 |
| Appendix D. No User Involvement | 113 |
| D.1. Asynchronous Authorization | 114 |
| D.2. Applying OAuth 2 Scopes and Client IDs | 117 |
| Appendix E. JSON Structures and Polymorphism | 118 |
| Author's Address | 119 |

1. Protocol

This protocol allows a piece of software, the resource client, to request delegated authorization to resource servers and direct information. This delegation is facilitated by an authorization server usually on behalf of a resource owner. The requesting party operating the software may interact with the authorization server to authenticate, provide consent, and authorize the request.

The process by which the delegation happens is known as a grant, and the GNAP protocol allows for the negotiation of the grant process over time by multiple parties acting in distinct roles.

This protocol solves many of the same use cases as OAuth 2.0 [RFC6749], OpenID Connect [OIDC], and the family of protocols that have grown up around that ecosystem. However, GNAP is not an extension of OAuth 2.0 and is not intended to be directly compatible with OAuth 2.0. GNAP seeks to provide functionality and solve use cases that OAuth 2.0 cannot easily or cleanly address. Even so, GNAP and OAuth 2.0 will exist in parallel for many deployments, and considerations have been taken to facilitate the mapping and transition from legacy systems to GNAP. Some examples of these can be found in Appendix D.2.

1.1. Roles

The parties in the GNAP protocol perform actions under different roles. Roles are defined by the actions taken and the expectations leveraged on the role by the overall protocol.

Authorization Server (AS) Manages the requested delegations for the RO. The AS issues tokens and directly delegated information to the RC. The AS is defined by its grant endpoint, a single URL that accepts a POST request with a JSON payload. The AS could also have other endpoints, including interaction endpoints and user code endpoints, and these are introduced to the RC as needed during the delegation process.

Resource Client (RC, aka "client") Requests tokens from the AS and uses tokens at the RS. An instance of the RC software is identified by its key, which can be known to the AS prior to the first request. The AS determines which policies apply to a given RC, including what it can request and on whose behalf.

Resource Server (RS, aka "API") Accepts tokens from the RC issued by the AS and serves delegated resources on behalf of the RO. There could be multiple RSs protected by the AS that the RC will call.

Resource Owner (RO) Authorizes the request from the RC to the RS, often interactively at the AS.

Requesting Party (RQ, aka "user") Operates and interacts with the RC.

The G NAP protocol design does not assume any one deployment architecture, but instead attempts to define roles that can be fulfilled in a number of different ways for different use cases. As long as a given role fulfills all of its obligations and behaviors as defined by the protocol, G NAP does not make additional requirements on its structure or setup.

Multiple roles can be fulfilled by the same party, and a given party can switch roles in different instances of the protocol. For example, the RO and RQ in many instances are the same person, where a user is authorizing the RC to act on their own behalf at the RS. In this case, one party fulfills both of the RO and RQ roles, but the roles themselves are still defined separately from each other to allow for other use cases where they are fulfilled by different parties.

For another example, in some complex scenarios, an RS receiving requests from one RC can act as an RC for a downstream secondary RS in order to fulfill the original request. In this case, one piece of software is both an RS and an RC from different perspectives, and it fulfills these roles separately as far as the overall protocol is concerned.

A single role need not be deployed as a monolithic service. For example, An RC could have components that are installed on the RQ's device as well as a back-end system that it communicates with. If both of these components participate in the delegation protocol, they are both considered part of the RC.

For another example, an AS could likewise be built out of many constituent components in a distributed architecture. The component that the RC calls directly could be different from the component that the RO interacts with to drive consent, since API calls and user interaction have different security considerations in many environments. Furthermore, the AS could need to collect identity claims about the RO from one system that deals with user attributes while generating access tokens at another system that deals with security rights. From the perspective of GNAP, all of these are pieces of the AS and together fulfill the role of the AS as defined by the protocol.

[[Editor's note: The names for the roles are an area of ongoing discussion within the working group, as is the appropriate precision of what activities and expectations a particular role covers. In particular, the AS might be formally decomposed into delegation components, that the client talks to, and interaction components, that the user talks to. Several alternative names have been proposed for different roles and components, including:

- * Grant Server (for Authorization Server)
- * Grant Client (for Resource Client)
- * Operator (for Requesting Party)

]]

1.2. Elements

In addition to the roles above, the protocol also involves several elements that are acted upon by the roles throughout the process.

Access Token A credential representing a set of access rights delegated to the RC. The access token is created by the AS, consumed and verified by the RS, and issued to and carried by the RC. The contents and format of the access token are opaque to the RC.

Grant The process by which a the RC requests and is given delegated access to the RS by the AS through the authority of the RO.

Key A cryptographic element binding a request to the holder of the key. Access tokens and RC instances can be associated with specific keys.

Resource A protected API served by the RS and accessed by the RC. Access to this resource is delegated by the RO as part of the grant process.

Subject Information Information about the RO that is returned directly to the RC from the AS without the RC making a separate call to an RS. Access to this information is delegated by the RO as part of the grant process.

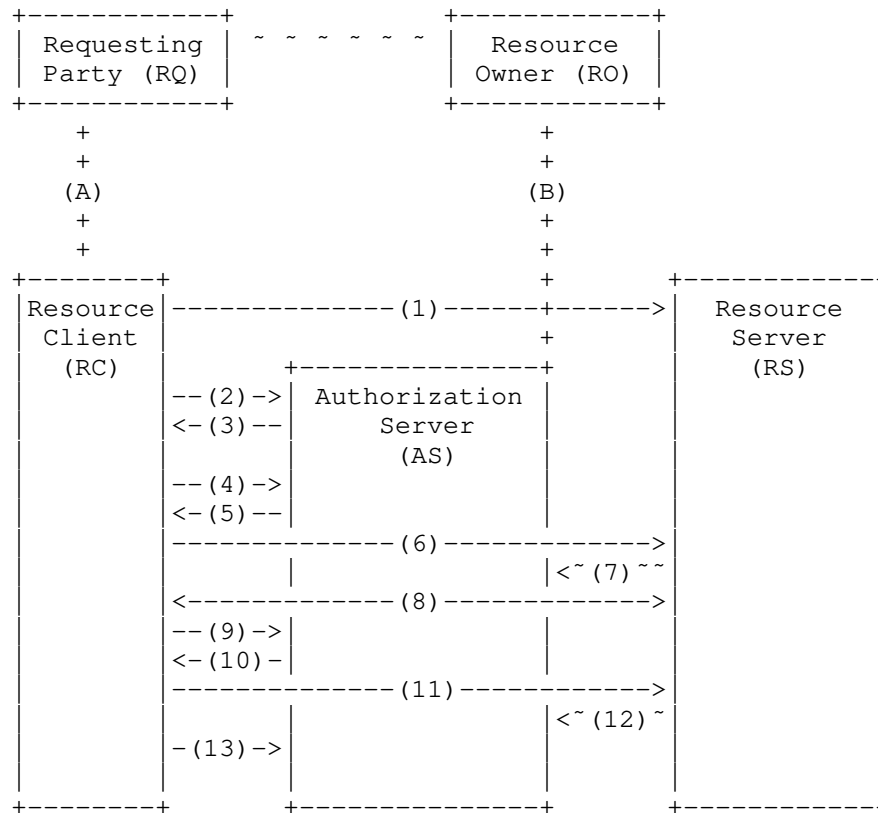
[[Editor's note: What other core elements need an introduction here? These aren't roles to be taken on by different parties, nor are they descriptions of the possible configurations of parties, but these are still important moving parts within the protocol.]]

1.3. Sequences

The GNAP protocol can be used in a variety of ways to allow the core delegation process to take place. Many portions of this process are conditionally present depending on the context of the deployments, and not every step in this overview will happen in all circumstances.

Note that a connection between roles in this process does not necessarily indicate that a specific protocol message is sent across the wire between the components fulfilling the roles in question, or that a particular step is required every time. For example, for an RC interested in only getting subject information directly, and not calling an RS, all steps involving the RS below do not apply.

In some circumstances, the information needed at a given stage is communicated out-of-band or is pre-configured between the components or entities performing the roles. For example, one entity can fulfil multiple roles, and so explicit communication between the roles is not necessary within the protocol flow.



Legend

+ + + indicates a possible interaction with a human
 ----- indicates an interaction between protocol roles
 ~ ~ ~ indicates a potential equivalence or out-of-band communication between roles

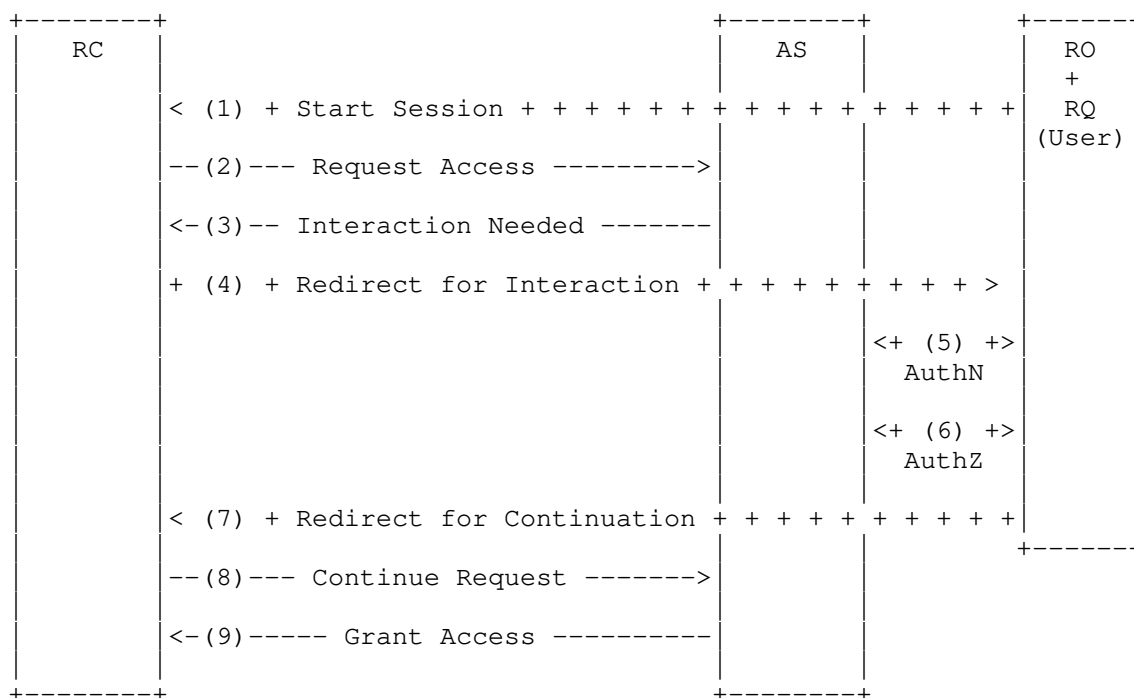
- * (A) The RQ interacts with the RC to indicate a need for resources on behalf of the RO. This could identify the RS the RC needs to call, the resources needed, or the RO that is needed to approve the request. Note that the RO and RQ are often the same entity in practice.
- * (1) The RC attempts to call the RS (Section 10.4) to determine what access is needed. The RS informs the RC that access can be granted through the AS. Note that for most situations, the RC already knows which AS to talk to and which kinds of access it needs.
- * (2) The RC requests access at the AS (Section 2).

- * (3) The AS processes the request and determines what is needed to fulfill the request. The AS sends its response to the RC (Section 3).
- * (B) If interaction is required, the AS interacts with the RO (Section 4) to gather authorization. The interactive component of the AS can function using a variety of possible mechanisms including web page redirects, applications, challenge/response protocols, or other methods. The RO approves the request for the RC being operated by the RQ. Note that the RO and RQ are often the same entity in practice.
- * (4) The RC continues the grant at the AS (Section 5).
- * (5) If the AS determines that access can be granted, it returns a response to the RC (Section 3) including an access token (Section 3.2) for calling the RS and any directly returned information (Section 3.4) about the RO.
- * (6) The RC uses the access token (Section 7) to call the RS.
- * (7) The RS determines if the token is sufficient for the request by examining the token, potentially calling the AS (Section 10.1). Note that the RS could also examine the token directly, call an internal data store, execute a policy engine request, or any number of alternative methods for validating the token and its fitness for the request.
- * (8) The RC to call the RS (Section 7) using the access token until the RS or RC determine that the token is no longer valid.
- * (9) When the token no longer works, the RC fetches an updated access token (Section 6.1) based on the rights granted in (5).
- * (10) The AS issues a new access token (Section 3.2) to the RC.
- * (11) The RC uses the new access token (Section 7) to call the RS.
- * (12) The RS determines if the new token is sufficient for the request by examining the token, potentially calling the AS (Section 10.1).
- * (13) The RC disposes of the token (Section 6.2) once the RC has completed its access of the RS and no longer needs the token.

The following sections and Appendix C contain specific guidance on how to use the GNAP protocol in different situations and deployments.

1.3.1. Redirect-based Interaction

In this example flow, the RC is a web application that wants access to resources on behalf of the current user, who acts as both the requesting party (RQ) and the resource owner (RO). Since the RC is capable of directing the user to an arbitrary URL and receiving responses from the user's browser, interaction here is handled through front-channel redirects using the user's browser. The RC uses a persistent session with the user to ensure the same user that is starting the interaction is the user that returns from the interaction.



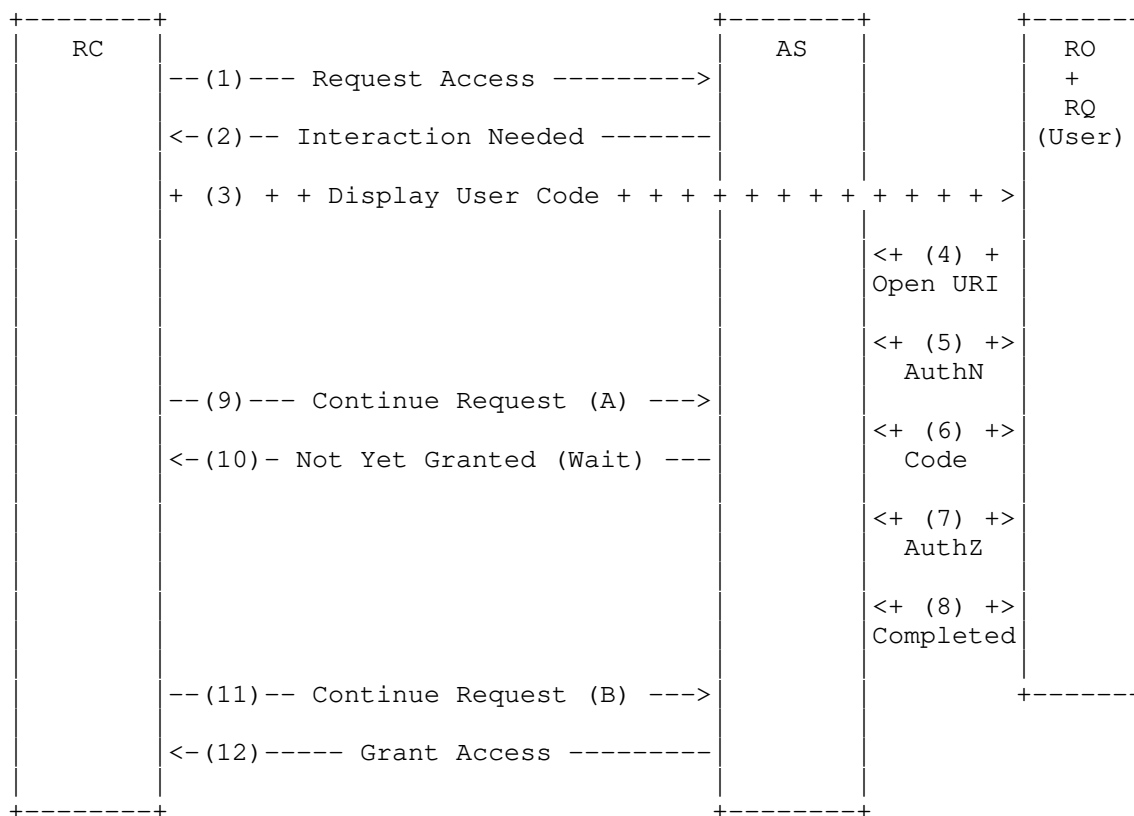
1. The RC establishes a verifiable session to the user, in the role of the RQ.
2. The RC requests access to the resource (Section 2). The RC indicates that it can redirect to an arbitrary URL (Section 2.5.1) and receive a callback from the browser (Section 2.5.3). The RC stores verification information for its callback in the session created in (1).

3. The AS determines that interaction is needed and responds (Section 3) with a URL to send the user to (Section 3.3.1) and information needed to verify the callback (Section 3.3.3) in (7). The AS also includes information the RC will need to continue the request (Section 3.1) in (8). The AS associates this continuation information with an ongoing request that will be referenced in (4), (6), and (8).
4. The RC stores the verification and continuation information from (3) in the session from (1). The RC then redirects the user to the URL (Section 4.1) given by the AS in (3). The user's browser loads the interaction redirect URL. The AS loads the pending request based on the incoming URL generated in (3).
5. The user authenticates at the AS, taking on the role of the RO.
6. As the RO, the user authorizes the pending request from the RC.
7. When the AS is done interacting with the user, the AS redirects the user back (Section 4.4.1) to the RC using the callback URL provided in (2). The callback URL is augmented with an interaction reference that the AS associates with the ongoing request created in (2) and referenced in (4). The callback URL is also augmented with a hash of the security information provided in (2) and (3). The RC loads the verification information from (2) and (3) from the session created in (1). The RC calculates a hash (Section 4.4.3) based on this information and continues only if the hash validates. Note that the RC needs to ensure that the parameters for the incoming request match those that it is expecting from the session created in (1). The RC also needs to be prepared for the RQ never being returned to the RC and handle time outs appropriately.
8. The RC loads the continuation information from (3) and sends the interaction reference from (7) in a request to continue the request (Section 5.1). The AS validates the interaction reference ensuring that the reference is associated with the request being continued.
9. If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the RC.

An example set of protocol messages for this method can be found in Appendix C.1.

1.3.2. User-code Interaction

In this example flow, the RC is a device that is capable of presenting a short, human-readable code to the user and directing the user to enter that code at a known URL. The RC is not capable of presenting an arbitrary URL to the user, nor is it capable of accepting incoming HTTP requests from the user's browser. The RC polls the AS while it is waiting for the RO to authorize the request. The user's interaction is assumed to occur on a secondary device. In this example it is assumed that the user is both the RQ and RO, though the user is not assumed to be interacting with the RC through the same web browser used for interaction at the AS.



1. The RC requests access to the resource (Section 2). The RC indicates that it can display a user code (Section 2.5.4).
2. The AS determines that interaction is needed and responds (Section 3) with a user code to communicate to the user (Section 3.3.4). This could optionally include a URL to direct

the user to, but this URL should be static and so could be configured in the RC's documentation. The AS also includes information the RC will need to continue the request (Section 3.1) in (8) and (10). The AS associates this continuation information with an ongoing request that will be referenced in (4), (6), (8), and (10).

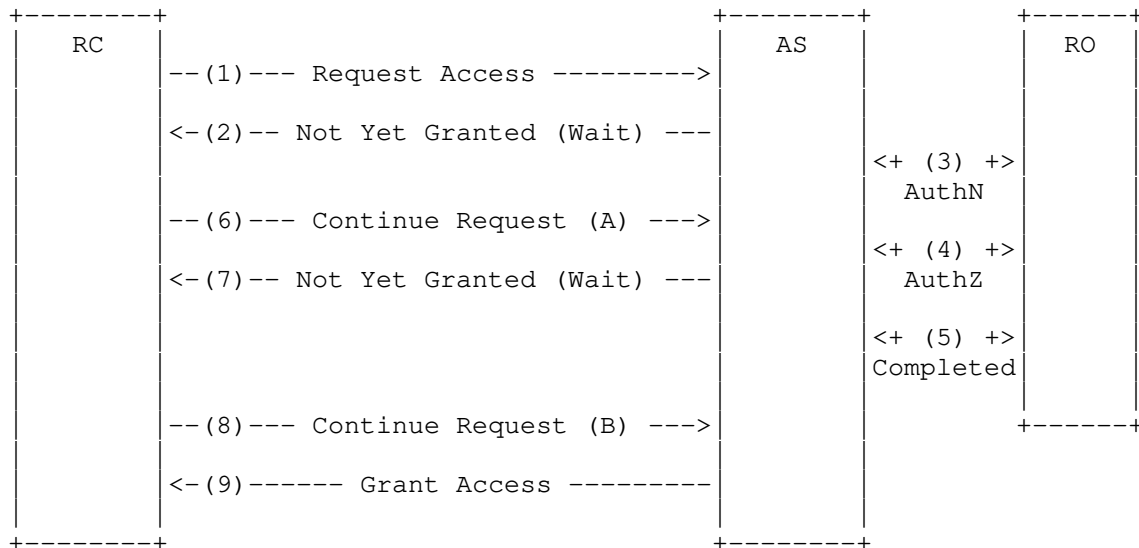
3. The RC stores the continuation information from (2) for use in (8) and (10). The RC then communicates the code to the user (Section 4.1) given by the AS in (2).
4. The user's directs their browser to the user code URL. This URL is stable and can be communicated via the RC's documentation, the AS documentation, or the RC software itself. Since it is assumed that the RO will interact with the AS through a secondary device, the RC does not provide a mechanism to launch the RO's browser at this URL.
5. The RQ authenticates at the AS, taking on the role of the RO.
6. The RO enters the code communicated in (3) to the AS. The AS validates this code against a current request in process.
7. As the RO, the user authorizes the pending request from the RC.
8. When the AS is done interacting with the user, the AS indicates to the RO that the request has been completed.
9. Meanwhile, the RC loads the continuation information stored at (3) and continues the request (Section 5). The AS determines which ongoing access request is referenced here and checks its state.
10. If the access request has not yet been authorized by the RO in (6), the AS responds to the RC to continue the request (Section 3.1) at a future time through additional polled continuation requests. This response can include updated continuation information as well as information regarding how long the RC should wait before calling again. The RC replaces its stored continuation information from the previous response (2). Note that the AS may need to determine that the RO has not approved the request in a sufficient amount of time and return an appropriate error to the RC.
11. The RC continues to poll the AS (Section 5.2) with the new continuation information in (9).

12. If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the RC.

An example set of protocol messages for this method can be found in Appendix C.2.

1.3.3. Asynchronous Authorization

In this example flow, the RQ and RO roles are fulfilled by different parties, and the RO does not interact with the RC. The AS reaches out asynchronously to the RO during the request process to gather the RO's authorization for the RC's request. The RC polls the AS while it is waiting for the RO to authorize the request.



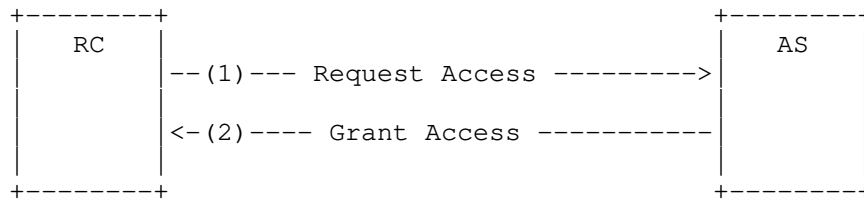
1. The RC requests access to the resource (Section 2). The RC does not send any interactions modes to the server, indicating that it does not expect to interact with the RO. The RC can also signal which RO it requires authorization from, if known, by using the user request section (Section 2.4).
2. The AS determines that interaction is needed, but the RC cannot interact with the RO. The AS responds (Section 3) with the information the RC will need to continue the request (Section 3.1) in (6) and (8), including a signal that the RC should wait before checking the status of the request again. The AS associates this continuation information with an ongoing request that will be referenced in (3), (4), (5), (6), and (8).

3. The AS determines which RO to contact based on the request in (1), through a combination of the user request (Section 2.4), the resources request (Section 2.1), and other policy information. The AS contacts the RO and authenticates them.
4. The RO authorizes the pending request from the RC.
5. When the AS is done interacting with the RO, the AS indicates to the RO that the request has been completed.
6. Meanwhile, the RC loads the continuation information stored at (3) and continues the request (Section 5). The AS determines which ongoing access request is referenced here and checks its state.
7. If the access request has not yet been authorized by the RO in (6), the AS responds to the RC to continue the request (Section 3.1) at a future time through additional polling. This response can include refreshed credentials as well as information regarding how long the RC should wait before calling again. The RC replaces its stored continuation information from the previous response (2). Note that the AS may need to determine that the RO has not approved the request in a sufficient amount of time and return an appropriate error to the RC.
8. The RC continues to poll the AS (Section 5.2) with the new continuation information from (7).
9. If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the RC.

An example set of protocol messages for this method can be found in Appendix D.1.

1.3.4. Software-only Authorization

In this example flow, the AS policy allows the RC to make a call on its own behalf, without the need for a RO to be involved at runtime to approve the decision. Since there is no explicit RO, the RC does not interact with an RO.

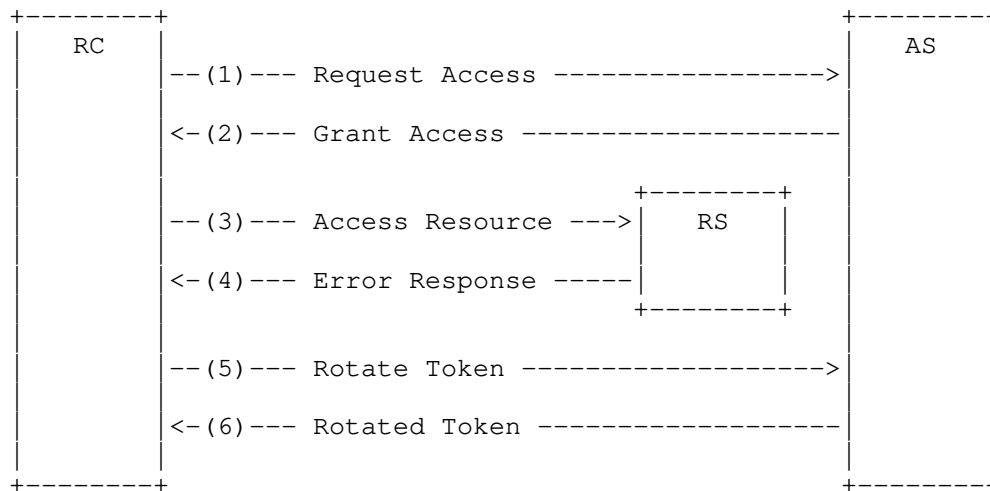


1. The RC requests access to the resource (Section 2). The RC does not send any interactions modes to the server.
2. The AS determines that the request is been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the RC.

An example set of protocol messages for this method can be found in Appendix D.

1.3.5. Refreshing an Expired Access Token

In this example flow, the RC receives an access token to access a resource server through some valid GNAP process. The RC uses that token at the RS for some time, but eventually the access token expires. The RC then gets a new access token by rotating the expired access token at the AS using the token's management URL.



1. The RC requests access to the resource (Section 2).

2. The AS grants access to the resource (Section 3) with an access token (Section 3.2) usable at the RS. The access token response includes a token management URI.
3. The RC presents the token (Section 7) to the RS. The RS validates the token and returns an appropriate response for the API.
4. When the access token is expired, the RS responds to the RC with an error.
5. The RC calls the token management URI returned in (2) to rotate the access token (Section 6.1). The RC presents the access token as well as the appropriate key.
6. The AS validates the rotation request including the signature and keys presented in (5) and returns a new access token (Section 3.2.1). The response includes a new access token and can also include updated token management information, which the RC will store in place of the values returned in (2).

2. Requesting Access

To start a request, the RC sends JSON [RFC8259] document with an object as its root. Each member of the request object represents a different aspect of the RC's request. Each field is described in detail in a section below.

resources Describes the rights that the RC is requesting for one or more access tokens to be used at RS's. Section 2.1

subject Describes the information about the RO that the RC is requesting to be returned directly in the response from the AS. Section 2.2

client Describes the RC that is making this request, including the key that the RC will use to protect this request and any continuation requests at the AS and any user-facing information about the RC used in interactions at the AS. Section 2.3

user Identifies the RQ to the AS in a manner that the AS can verify, either directly or by interacting with the RQ to determine their status as the RO. Section 2.4

interact Describes the modes that the RC has for allowing the RO to interact with the AS and modes for the RC to receive updates when interaction is complete. Section 2.5

capabilities Identifies named extension capabilities that the RC can use, signaling to the AS which extensions it can use. Section 2.6

existing_grant Identifies a previously-existing grant that the RC is extending with this request. Section 2.7

claims Identifies the identity claims to be returned as part of an OpenID Connect claims request. Section 2.8

Additional members of this request object can be defined by extensions to this protocol as described in Section 2.9

A non-normative example of a grant request is below:

```
{
  "resources": [
    {
      "type": "photo-api",
      "actions": [
        "read",
        "write",
        "dolphin"
      ],
      "locations": [
        "https://server.example.net/",
        "https://resource.local/other"
      ],
      "datatypes": [
        "metadata",
        "images"
      ]
    },
    "dolphin-metadata"
  ],
  "client": {
    "display": {
      "name": "My Client Display Name",
      "uri": "https://example.net/client"
    },
    "key": {
      "proof": "jwsd",
      "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "kid": "xyz-1",
        "alg": "RS256",
        "n": "kOB5rR4Jv0GMeL...."
      }
    }
  }
}
```

```

    }
  },
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.example.net/return/123455",
      "nonce": "LKLTi25DK82FX4T4QFzC"
    }
  },
  "capabilities": ["ext1", "ext2"],
  "subject": {
    "sub_ids": ["iss-sub", "email"],
    "assertions": ["id_token"]
  }
}

```

The request MUST be sent as a JSON object in the body of the HTTP POST request with Content-Type "application/json", unless otherwise specified by the signature mechanism.

2.1. Requesting Resources

If the RC is requesting one or more access tokens for the purpose of accessing an API, the RC MUST include a "resources" field. This field MUST be an array (for a single access token (Section 2.1.1)) or an object (for multiple access tokens (Section 2.1.3)), as described in the following sections.

2.1.1. Requesting a Single Access Token

When requesting an access token, the RC MUST send a "resources" field containing a JSON array. The elements of the JSON array represent rights of access that the RC is requesting in the access token. The requested access is the sum of all elements within the array.

The RC declares what access it wants to associated with the resulting access token using objects that describe multiple dimensions of access. Each object contains a "type" property that determines the type of API that the RC is calling.

type The type of resource request as a string. This field MAY define which other fields are allowed in the request object. This field is REQUIRED.

The value of this field is under the control of the AS. This field MUST be compared using an exact byte match of the string value against known types by the AS. The AS MUST ensure that there is no

collision between different authorization data types that it supports. The AS MUST NOT do any collation or normalization of data types during comparison. It is RECOMMENDED that designers of general-purpose APIs use a URI for this field to avoid collisions between multiple API types protected by a single AS.

While it is expected that many APIs will have its own properties, a set of common properties are defined here. Specific API implementations SHOULD NOT re-use these fields with different semantics or syntax. The available values for these properties are determined by the API being protected at the RS.

[[Editor's note: this will align with OAuth 2 RAR, but the details of exactly how it aligns are TBD. Since RAR needs to work in the confines of OAuth 2, RAR has to define how to interact with "scope", "resource", and other existing OAuth 2 mechanisms that don't exist in GNAP.]].

actions The types of actions the RC will take at the RS as an array of strings. For example, an RC asking for a combination of "read" and "write" access.

locations The location of the RS as an array of strings. These strings are typically URIs identifying the location of the RS.

datatypes The kinds of data available to the RC at the RS's API as an array of strings. For example, an RC asking for access to raw "image" data and "metadata" at a photograph API.

identifier A string identifier indicating a specific resource at the RS. For example, a patient identifier for a medical API or a bank account number for a financial API.

The following non-normative example shows the use of both common and API-specific fields as part of two different access "type" values.

```

"resources": [
  {
    "type": "photo-api",
    "actions": [
      "read",
      "write",
      "dolphin"
    ],
    "locations": [
      "https://server.example.net/",
      "https://resource.local/other"
    ],
    "datatypes": [
      "metadata",
      "images"
    ]
  },
  {
    "type": "financial-transaction",
    "actions": [
      "withdraw"
    ],
    "identifier": "account-14-32-32-3",
    "currency": "USD"
  }
]

```

If this request is approved, the resulting access token (Section 3.2.1) will include the sum of both of the requested types of access.

2.1.2. Requesting Resources By Reference

Instead of sending an object describing the requested resource (Section 2.1.1), a RC MAY send a string known to the AS or RS representing the access being requested. Each string SHOULD correspond to a specific expanded object representation at the AS.

[[Editor's note: we could describe more about how the expansion would work. For example, expand into an object where the value of the "type" field is the value of the string. Or we could leave it open and flexible, since it's really up to the AS/RS to interpret.]]

```

"resources": [
  "read", "dolphin-metadata", "some other thing"
]

```

This value is opaque to the RC and MAY be any valid JSON string, and therefore could include spaces, unicode characters, and properly escaped string sequences. However, in some situations the value is intended to be seen and understood by the RC developer. In such cases, the API designer choosing any such human-readable strings SHOULD take steps to ensure the string values are not easily confused by a developer.

This functionality is similar in practice to OAuth 2's "scope" parameter [RFC6749], where a single string represents the set of access rights requested by the RC. As such, the reference string could contain any valid OAuth 2 scope value as in Appendix D.2. Note that the reference string here is not bound to the same character restrictions as in OAuth 2's "scope" definition.

A single "resources" array MAY include both object-type and string-type resource items.

```
"resources": [
  {
    "type": "photo-api",
    "actions": [
      "read",
      "write",
      "dolphin"
    ],
    "locations": [
      "https://server.example.net/",
      "https://resource.local/other"
    ],
    "datatypes": [
      "metadata",
      "images"
    ]
  },
  "read",
  "dolphin-metadata",
  {
    "type": "financial-transaction",
    "actions": [
      "withdraw"
    ],
    "identifier": "account-14-32-32-3",
    "currency": "USD"
  },
  "some other thing"
]
```

[[Editor's note: passing resource requests by reference really is akin to a "scope", and we have many years of experience showing us that the simplicity of giving a developer a set of strings to send is a simple and powerful pattern. We could always require objects and just use the "type" field as a scope value, but that's a lot of complexity to pay for the simple case. Client developers will always know which kind they need to send, because they're picking from the API's documentation.]]

2.1.3. Requesting Multiple Access Tokens

When requesting multiple access tokens, the resources field is a JSON object. The names of the JSON object fields are token identifiers chosen by the RC, and MAY be any valid string. The values of the JSON object fields are JSON arrays representing a single access token request, as specified in requesting a single access token (Section 2.1.1).

The following non-normative example shows a request for two separate access tokens, "token1" and "token2".

```
"resources": {
  "token1": [
    {
      "type": "photo-api",
      "actions": [
        "read",
        "write",
        "dolphin"
      ],
      "locations": [
        "https://server.example.net/",
        "https://resource.local/other"
      ],
      "datatypes": [
        "metadata",
        "images"
      ]
    },
    "dolphin-metadata"
  ],
  "token2": [
    {
      "type": "walrus-access",
      "actions": [
        "foo",
        "bar"
      ],
      "locations": [
        "https://resource.other/"
      ],
      "datatypes": [
        "data",
        "pictures",
        "walrus whiskers"
      ]
    }
  ]
}
```

Any approved access requests are returned in the multiple access token response (Section 3.2.2) structure using the token identifiers in the request.

2.1.4. Signaling Token Behavior

While the AS is ultimately in control of how tokens are returned and bound to the RC, sometimes the RC has context about what it can support that can affect the AS's response. This specification defines several flags that are passed as resource reference strings (Section 2.1.2).

Each flag applies only to the single resource request in which it appears.

Support of all flags is optional, such as any other resource reference value.

multi_token The RC wishes to support multiple simultaneous access tokens through the token rotation process. When the RC rotates an access token (Section 6.1), the AS does not invalidate the previous access token. The old access token continues to remain valid until such time as it expires or is revoked through other means.

split_token The RC is capable of receiving multiple access tokens (Section 3.2.2) in response to any single token request (Section 2.1.1), or receiving a different number of tokens than specified in the multiple token request (Section 2.1.3). The labels of the returned additional tokens are chosen by the AS. The client **MUST** be able to tell from the token response where and how it can use the each access tokens. [[Editor's note: This functionality is controversial at best as it requires significantly more complexity on the client in order to solve one class of AS/RS deployment choices.]]

bind_token The RC wants the issued access token to be bound to the key the RC used (Section 2.3.2) to make the request. The resulting access token **MUST** be bound using the same "proof" mechanism used by the client with a "key" value of "true", indicating the client's presented key is to be used for binding. [[Editor's note: should there be a different flag and mechanism for the client to explicitly indicate which binding method it wants to use, especially if the client wants to use a different method at the AS than the RS?]]

The AS **MUST** respond with any applied flags in the token response (Section 3.2) "resources" section.

In this non-normative example, the requested access token is to be bound to the client's key and should be kept during rotation.

```
"resources": [
  {
    "type": "photo-api",
    "actions": [
      "read",
      "write",
      "dolphin"
    ],
    "locations": [
      "https://server.example.net/",
      "https://resource.local/other"
    ],
    "datatypes": [
      "metadata",
      "images"
    ]
  },
  "read",
  "bind_token",
  "multi_token"
]
```

Additional flags can be registered in a registry TBD (Section 12).

[[Editor's note: while these reference values are "reserved", the ultimate decider for what a reference means is the AS, which means an AS could arguably decide that one of these values means something else. Also, this kind of reservation potentially steps on API namespaces, which OAuth 2 is careful not to do but common extensions like OIDC do with their own scope definitions. However, in OIDC, several "scope" values have behavior similar to what's defined here, particularly "openid" turns on ID tokens in the response and "offline_access" signals for the return of a refresh token, and these can be used outside of OpenID Connect itself. However, to keep these flags out of the general API namespace, we could use a different syntax for sending them. In particular, they could be defined under a G NAP-specific "type" object, where all the flags are fields on the object.

```

resources: [
  {
    type: "gnap-flags",
    flag1: true,
    flag2: false,
    flag3: true ...
  },
  "reference1",
  "scope2", ...
]

```

Alternatively, all the flags could be sent in an array separate from the rest of the request.

```

resources: [
  "reference1",
  "scope2",
  ["flag1", "flag2", "flag3"] ...
]

```

This whole thing might also belong in an extension, as it's advanced behavior signaling for very specific cases. However, it seems other extensions would be likely to extend this kind of thing, like OIDC did with "offline_access".]]

2.2. Requesting User Information

If the RC is requesting information about the RO from the AS, it sends a "subject" field as a JSON object. This object MAY contain the following fields (or additional fields defined in a registry TBD (Section 12)).

sub_ids An array of subject identifier subject types requested for the RO, as defined by [I-D.ietf-secevent-subject-identifiers].

assertions An array of requested assertion formats. Possible values include "id_token" for an [OIDC] ID Token and "saml2" for a SAML 2 assertion. Additional assertion values are defined by a registry TBD (Section 12). [[Editor's note: These values are lifted from [RFC8693]'s "token type identifiers" list, but is there a better source?]]

```

"subject": {
  "sub_ids": [ "iss-sub", "email" ],
  "assertions": [ "id_token", "saml2" ]
}

```

The AS can determine the RO's identity and permission for releasing this information through interaction with the RO (Section 4), AS policies, or assertions presented by the RC (Section 2.4). If this is determined positively, the AS MAY return the RO's information in its response (Section 3.4) as requested.

Subject identifiers requested by the RC serve only to identify the RO in the context of the AS and can't be used as communication channels by the RC, as discussed in Section 3.4. One method of requesting communication channels and other identity claims are discussed in Section 2.8.

The AS SHOULD NOT re-use subject identifiers for multiple different ROs.

[[Editor's Note: What we're really saying here is that "even if the AS gives you an email address to identify the user, that isn't a claim that this is a valid email address for that current user, so don't try to email them." In order to get a workable email address, or anything that you can use to contact them, you'd need a full identity protocol and not just this. Also, subject identifiers are asserted by the AS and therefore naturally scoped to the AS. Would changing the name to "as_sub_ids" or "local_sub_ids" help convey that point?]]

Note: the "sub_ids" and "assertions" request fields are independent of each other, and a returned assertion MAY omit a requested subject identifier.

[[Editor's note: we're potentially conflating these two types in the same structure, so perhaps these should be split. There's also a difference between user information and authentication event information.]]

2.3. Identifying the RC

When sending a non-continuation request to the AS, the RC MUST identify itself by including the "client" field of the request and by signing the request as described in Section 8. Note that for a continuation request (Section 5), the RC instance is identified by its association with the request being continued and so this field is not sent under those circumstances.

When RC information is sent by value, the "client" field of the request consists of a JSON object with the following fields.

key The public key of the RC to be used in this request as described in Section 2.3.2. This field is REQUIRED.

`class_id` An identifier string that the AS can use to identify the software comprising this instance of the RC. The contents and format of this field are up to the AS. This field is OPTIONAL.

`display` An object containing additional information that the AS MAY display to the RO during interaction, authorization, and management. This field is OPTIONAL.

```
"client": {
  "key": {
    "proof": "httpsig",
    "jwk": {
      "kty": "RSA",
      "e": "AQAB",
      "kid": "xyz-1",
      "alg": "RS256",
      "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8xY..."
    },
    "cert": "MIIHDCCAwwSgAwIBAgIBATANBgkqhkiG9w0BAQsFA..."
  },
  "class_id": "web-server-1234",
  "display": {
    "name": "My Client Display Name",
    "uri": "https://example.net/client"
  }
}
```

Additional fields are defined in a registry TBD (Section 12).

The RC MUST prove possession of any presented key by the "proof" mechanism associated with the key in the request. Proof types are defined in a registry TBD (Section 12) and an initial set of methods is described in Section 8.

Note that the AS MAY know the RC's public key ahead of time, and the AS MAY apply different policies to the request depending on what has been registered against that key. If the same public key is sent by value on subsequent access requests, the AS SHOULD treat these requests as coming from the same RC software instance for purposes of identification, authentication, and policy application. If the AS does not know the RC's public key ahead of time, the AS MAY accept or reject the request based on AS policy, attestations within the client request, and other mechanisms.

[[Editor's note: additional client attestation frameworks will eventually need to be addressed here. For example, the organization the client represents, or a family of client software deployed in a cluster, or the posture of the device the client is installed on. These all need to be separable from the client's key and potentially the instance identifier.]]

2.3.1. Identifying the RC Instance

If the RC has an instance identifier that the AS can use to determine appropriate key information, the RC can send this value in the "instance_id" field. The instance identifier MAY be assigned to an RC instance at runtime through the Section 3.5 or MAY be obtained in another fashion, such as a static registration process at the AS.

instance_id An identifier string that the AS can use to identify the particular instance of this RC. The content and structure of this identifier is opaque to the RC.

```
"client": {
  "instance_id": "client-541-ab"
}
```

If there are no additional fields to send, the RC MAY send the instance identifier as a direct reference value in lieu of the object.

```
"client": "client-541-ab"
```

When the AS receives a request with an instance identifier, the AS MUST ensure that the key used to sign the request (Section 8) is associated with the instance identifier.

If the "instance_id" field is sent, it MUST NOT be accompanied by other fields unless such fields are explicitly marked safe for inclusion alongside the instance identifier.

[[Editor's note: It seems clear that an instance identifier is mutually exclusive with most of the fields in the request (eg, we don't want an attacker being able to swap out a client's registered key just by accessing the identifier). However, some proposed concepts might fit alongside an instance identifier that change at runtime, such as device posture or another dynamic attestation. Should these be sent in the "client" block alongside the instance identifier, should there be a separate top-level block for runtime attestations, or some other mechanism?]]

2.3.4. Authenticating the RC

If the presented key is known to the AS and is associated with a single instance of the RC software, the process of presenting a key and proving possession of that key is sufficient to authenticate the RC to the AS. The AS MAY associate policies with the RC software identified by this key, such as limiting which resources can be requested and which interaction methods can be used. For example, only specific RCs with certain known keys might be trusted with access tokens without the AS interacting directly with the RO as in Appendix D.

The presentation of a key allows the AS to strongly associate multiple successive requests from the same RC with each other. This is true when the AS knows the key ahead of time and can use the key to authenticate the RC software, but also if the key is ephemeral and created just for this request. As such the AS MAY allow for RCs to make requests with unknown keys. This pattern allows for ephemeral RCs, such as single-page applications, and RCs with many individual instances, such as mobile applications, to generate their own key pairs and use them within the protocol without having to go through a separate registration step. The AS MAY limit which capabilities are made available to RCs with unknown keys. For example, the AS could have a policy saying that only previously-registered RCs can request particular resources, or that all RCs with unknown keys have to be interactively approved by an RO.

2.4. Identifying the User

If the RC knows the identity of the RQ through one or more identifiers or assertions, the RC MAY send that information to the AS in the "user" field. The RC MAY pass this information by value or by reference.

sub_ids An array of subject identifiers for the RQ, as defined by [I-D.ietf-secevent-subject-identifiers].

assertions An object containing assertions as values keyed on the assertion type defined by a registry TBD (Section 12). Possible keys include "id_token" for an [OIDC] ID Token and "saml2" for a SAML 2 assertion. Additional assertion values are defined by a registry TBD (Section 12). [[Editor's note: These keys are lifted from [RFC8693]'s "token type identifiers" list, but is there a better source? Additionally: should this be an array of objects with internal typing like the sub_ids? Do we expect more than one assertion per user anyway?]]

```

"user": {
  "sub_ids": [ {
    "subject_type": "email",
    "email": "user@example.com"
  } ],
  "assertions": {
    "id_token": "eyJ..."
  }
}

```

Subject identifiers are hints to the AS in determining the RO and MUST NOT be taken as declarative statements that a particular RO is present at the RC and acting as the RQ. Assertions SHOULD be validated by the AS. [[editor's note: is this a MUST? Assertion validation is extremely specific to the kind of assertion in place, what other guidance and requirements can we put in place here?]]

If the identified RQ does not match the RO present at the AS during an interaction step, the AS SHOULD reject the request with an error.

[[Editor's note: we're potentially conflating identification (sub_ids) and provable presence (assertions and a trusted reference handle) in the same structure, so perhaps these should be split. The security parameters are pretty different here.]]

If the AS trusts the RC to present verifiable assertions, the AS MAY decide, based on its policy, to skip interaction with the RO, even if the RC provides one or more interaction modes in its request.

2.4.1. Identifying the User by Reference

User reference identifiers can be dynamically issued by the AS (Section 3.5) to allow the RC to represent the same RQ to the AS over subsequent requests.

If the RC has a reference for the RQ at this AS, the RC MAY pass that reference as a string. The format of this string is opaque to the RC.

```

"user": "XUT2MFM1XBIKJKSDU8QM"

```

User reference identifiers are not intended to be human-readable user identifiers or structured assertions. For the RC to send either of these, use the full user request object (Section 2.4) instead.

[[Editor's note: we might be able to fold this function into an unstructured user assertion reference issued by the AS to the RC. We could put it in as an assertion type of "gnap_reference" or something

like that. Downside: it's more verbose and potentially confusing to the client developer to have an assertion-like thing that's internal to the AS and not an assertion.]]

If the AS does not recognize the user reference, it MUST return an error.

2.5. Interacting with the User

Many times, the AS will require interaction with the RO in order to approve a requested delegation to the RC for both resources and direct claim information. Many times the RQ using the RC is the same person as the RO, and the RC can directly drive interaction with the AS by redirecting the RQ on the same device, or by launching an application. Other times, the RC can provide information to start the RO's interaction on a secondary device, or the RC will wait for the RO to approve the request asynchronously. The RC could also be signaled that interaction has completed by the AS making callbacks. To facilitate all of these modes, the RC declares the means that it can interact using the "interact" field.

The "interact" field is a JSON object with keys that declare different interaction modes. A RC MUST NOT declare an interaction mode it does not support. The RC MAY send multiple modes in the same request. There is no preference order specified in this request. An AS MAY respond to any, all, or none of the presented interaction modes (Section 3.3) in a request, depending on its capabilities and what is allowed to fulfill the request. This specification defines the following interaction modes:

redirect Indicates that the RC can direct the RQ to an arbitrary URL at the AS for interaction. Section 2.5.1

app Indicates that the RC can launch an application on the RQ's device for interaction. Section 2.5.2

callback Indicates that the RC can receive a callback from the AS after interaction with the RO has concluded. Section 2.5.3

user_code Indicates that the RC can communicate a human-readable short code to the RQ for use with a stable URL at the AS. Section 2.5.4

ui_locales Indicates the RQ's preferred locales that the AS can use during interaction, particularly before the RO has authenticated. Section 2.5.5

The following sections detail requests for interaction modes. Additional interaction modes are defined in a registry TBD (Section 12).

[[Editor's note: there need to be more examples (Appendix C) that knit together the interaction modes into common flows, like an authz-code equivalent. But it's important for the protocol design that these are separate pieces to allow such knitting to take place.]]

In this non-normative example, the RC is indicating that it can redirect (Section 2.5.1) the RQ to an arbitrary URL and can receive a callback (Section 2.5.3) through a browser request.

```
"interact": {
  "redirect": true,
  "callback": {
    "method": "redirect",
    "uri": "https://client.example.net/return/123455",
    "nonce": "LKLTi25DK82FX4T4QFZC"
  }
}
```

In this non-normative example, the RC is indicating that it can display a use code (Section 2.5.4) and direct the RQ to an arbitrary URL of maximum length (Section 2.5.1.1) 255 characters, but it cannot accept a callback.

```
"interact": {
  "redirect": 255,
  "user_code": true
}
```

If the RC does not provide a suitable interaction mechanism, the AS cannot contact the RO asynchronously, and the AS determines that interaction is required, then the AS SHOULD return an error since the RC will be unable to complete the request without authorization.

The AS SHOULD apply suitable timeouts to any interaction mechanisms provided, including user codes and redirection URLs. The RC SHOULD apply suitable timeouts to any callback URLs.

2.5.1. Redirect to an Arbitrary URL

If the RC is capable of directing the RQ to a URL defined by the AS at runtime, the RC indicates this by sending the "redirect" field with the boolean value "true". The means by which the RC will activate this URL is out of scope of this specification, but common methods include an HTTP redirect, launching a browser on the RQ's device, providing a scannable image encoding, and printing out a URL to an interactive console.

```
"interact": {
  "redirect": true
}
```

If this interaction mode is supported for this RC and request, the AS returns a redirect interaction response Section 3.3.1.

2.5.1.1. Redirect to an Arbitrary Shortened URL

If the RC would prefer to redirect to a shortened URL defined by the AS at runtime, the RC indicates this by sending the "redirect" field with an integer indicating the maximum character length of the returned URL. The AS MAY use this value to decide whether to return a shortened form of the response URL. If the AS cannot shorten its response URL enough to fit in the requested size, the AS SHOULD return an error. [[Editor's note: Or maybe just ignore this part of the interaction request?]]

```
"interact": {
  "redirect": 255
}
```

If this interaction mode is supported for this RC and request, the AS returns a redirect interaction response with short URL Section 3.3.1.

2.5.2. Open an Application-specific URL

If the RC can open a URL associated with an application on the RQ's device, the RC indicates this by sending the "app" field with boolean value "true". The means by which the RC determines the application to open with this URL are out of scope of this specification.

```
"interact": {
  "app": true
}
```

If this interaction mode is supported for this RC and request, the AS returns an app interaction response with an app URL payload Section 3.3.2.

[[Editor's note: this is similar to the "redirect" above today as most apps use captured URLs, but there seems to be a desire for splitting the web-based interaction and app-based interaction into different URIs. There's also the possibility of wanting more in the payload than can be reasonably put into the URL, or at least having separate payloads.]]

2.5.3. Receive a Callback After Interaction

If the RC is capable of receiving a message from the AS indicating that the RO has completed their interaction, the RC indicates this by sending the "callback" field. The value of this field is an object containing the following members.

uri REQUIRED. Indicates the URI to send the RO to after interaction. This URI MAY be unique per request and MUST be hosted by or accessible by the RC. This URI MUST NOT contain any fragment component. This URI MUST be protected by HTTPS, be hosted on a server local to the RO's browser ("localhost"), or use an application-specific URI scheme. If the RC needs any state information to tie to the front channel interaction response, it MUST use a unique callback URI to link to that ongoing state. The allowable URIs and URI patterns MAY be restricted by the AS based on the RC's presented key information. The callback URI SHOULD be presented to the RO during the interaction phase before redirect. [[Editor's note: should we enforce the callback URI to be unique per request? That helps with some fixation attacks, but not with others, and it would be problematic for an AS that wants to lock down each client instance to a single callback instead of a family/pattern of callbacks.]]

nonce REQUIRED. Unique value to be used in the calculation of the "hash" query parameter sent to the callback URL, must be sufficiently random to be unguessable by an attacker. MUST be generated by the RC as a unique value for this request.

method REQUIRED. The callback method that the AS will use to contact the RC. Valid values include "redirect" Section 2.5.3.1 and "push" Section 2.5.3.2, with other values defined by a registry TBD (Section 12).

hash_method OPTIONAL. The hash calculation mechanism to be used for

the callback hash in Section 4.4.3. Can be one of "sha3" or "sha2". If absent, the default value is "sha3". [[Editor's note: This should be expandable via a registry of cryptographic options, and it would be good if we didn't define our own identifiers here. See also note about cryptographic functions in Section 4.4.3.]]

```
"interact": {
  "callback": {
    "method": "redirect",
    "uri": "https://client.example.net/return/123455",
    "nonce": "LKLTi25DK82FX4T4QFZC"
  }
}
```

If this interaction mode is supported for this RC and request, the AS returns a nonce for use in validating the callback response (Section 3.3.3). Requests to the callback URI MUST be processed as described in Section 4.4, and the AS MUST require presentation of an interaction callback reference as described in Section 5.1.

[[Editor's note: There has been some call for a post-interaction redirect that is not tied to the underlying security model - specifically, sending the user over to a client-hosted page with client-specific instructions on how to continue. This would be something hosted externally to the client instance, so the client instance would never see this incoming call. We could accomplish that using this "callback" post-redirect mechanism but with "method": "static" or "nonce": false or some other signal to indicate that the client won't see the incoming request.]]

[[Editor's note: The callback information could alternatively be combined with other methods like "redirect", essentially putting everything in the "callback" object into the field for the other objects. However, this would require each method to define its own set of rules about how callbacks can be used, and we would want them all to be consistent with each other with clear information about how the AS is supposed to respond to all of these.

```
"interact" {
  "redirect": {
    "method": "redirect",
    "uri": "https://client.example.net/return/123455",
    "nonce": "LKLTi25DK82FX4T4QFZC"
  }
}
```

So if the object is there, you do the redirect on completion, if the object isn't there (it's a boolean, like today), you don't redirect when you're done. Previous versions of this specification used this structure, but it was abandoned in favor of the current setup to allow for different combinations of user interaction methods at the same time while still keeping a consistent security model. OAuth 2's "grant_type" model has proved to be limiting in unanticipated ways since it requires an entirely new grant type to be invented any time there is a new combination of aspects, or it requires each grant type to have many of the same optionalities. Combining these fields back into one, in this way, would allow a client to declare that it expects a callback in response to one kind of interaction method but not others, and include multiple combinations at once. For example, if a client wants to allow a user to redirect to the AS and back on the same device, or to use a usercode on a secondary device without a callback, and the client wants to offer both modes simultaneously. This could alternately be accomplished by allowing the client to "bundle" interaction parameters together, if desirable - for example, if "interact" were an array, the client would accept any combination represented by one object. This example binds the "callback" only to the first "redirect" method, and second (short) "redirect" and "user_code" method do not use a callback.

```
"interact": [
  {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.example.net/return/123455",
      "nonce": "LKLTi25DK82FX4T4QFZC"
    }
  },
  {
    "redirect": 255,
    "user_code": true
  }
]
```

It's not clear what a response to such an array would be. Would the AS pick one of these bundles? Would it be allowed to respond to any or all of them? Could an AS use different URIs for each bundle? (This seems likely, at least.) Would there be a security problem if the AS used the same URI for both bundles, since one requires a front channel redirect and the other does not?

```
]]
```

2.5.3.1. Receive an HTTP Callback Through the Browser

A callback "method" value of "redirect" indicates that the RC will expect a call from the RO's browser using the HTTP method GET as described in Section 4.4.1.

```
"interact": {
  "callback": {
    "method": "redirect",
    "uri": "https://client.example.net/return/123455",
    "nonce": "LKLTi25DK82FX4T4QFZC"
  }
}
```

Requests to the callback URI MUST be processed by the RC as described in Section 4.4.1.

Since the incoming request to the callback URL is from the RO's browser, this method is usually used when the RO and RQ are the same entity. As such, the RC MUST ensure the RQ is present on the request to prevent substitution attacks.

2.5.3.2. Receive an HTTP Direct Callback

A callback "method" value of "push" indicates that the RC will expect a call from the AS directly using the HTTP method POST as described in Section 4.4.2.

```
"interact": {
  "callback": {
    "method": "push",
    "uri": "https://client.example.net/return/123455",
    "nonce": "LKLTi25DK82FX4T4QFZC"
  }
}
```

Requests to the callback URI MUST be processed by the RC as described in Section 4.4.2.

Since the incoming request to the callback URL is from the AS and not from the RO's browser, the RC MUST NOT require the RQ to be present on incoming HTTP the request.

[[Editor's note: This post-interaction method can be used in advanced use cases like asynchronous authorization, or simply to signal the client that it should move to the next part of the protocol, even when there is no user present at the client. As such it can feel a little odd being inside the "interact" block of the

protocol, but it does align with the redirect-based "callback" method and it seems they really should be mutually-exclusive. Additionally, should there be a method for simply pushing the updated response directly to the client, instead?]]

2.5.4. Display a Short User Code

If the RC is capable of displaying or otherwise communicating a short, human-entered code to the RO, the RC indicates this by sending the "user_code" field with the boolean value "true". This code is to be entered at a static URL that does not change at runtime, as described in Section 3.3.4.

```
"interact": {
  "user_code": true
}
```

If this interaction mode is supported for this RC and request, the AS returns a user code and interaction URL as specified in Section 4.2.

2.5.5. Indicate Desired Interaction Locales

If the RC knows the RQ's locale and language preferences, the RC can send this information to the AS using the "ui_locales" field with an array of locale strings as defined by [RFC5646].

```
"interact": {
  "ui_locales": ["en-US", "fr-CA"]
}
```

If possible, the AS SHOULD use one of the locales in the array, with preference to the first item in the array supported by the AS. If none of the given locales are supported, the AS MAY use a default locale.

2.5.6. Extending Interaction Modes

Additional interaction modes are defined in a registry TBD (Section 12).

[[Editor's note: we should have guidance in here about how to define other interaction modes. There's already interest in defining message-based protocols like DIDCOMM and challenge-response protocols like FIDO, for example.]]

2.6. Declaring RC Capabilities

If the RC supports extension capabilities, it MAY present them to the AS in the "capabilities" field. This field is an array of strings representing specific extensions and capabilities, as defined by a registry TBD (Section 12).

```
"capabilities": ["ext1", "ext2"]
```

2.7. Referencing an Existing Grant Request

If the RC has a reference handle from a previously granted request, it MAY send that reference in the "existing_grant" field. This field is a single string consisting of the "value" of the "access_token" returned in a previous request's continuation response (Section 3.1).

```
"existing_grant": "80UPRY5NM33OMUKMKSKU"
```

The AS MUST dereference the grant associated with the reference and process this request in the context of the referenced one. The AS MUST NOT alter the existing grant associated with the reference.

[[Editor's note: this basic capability is to allow for both step-up authorization and downscoped authorization, but by explicitly creating a new request and not modifying an existing one. What's the best guidance for how an AS should process this? What are the use cases that help differentiate this from modification of an existing request?]]

2.8. Requesting OpenID Connect Claims

If the RC and AS both support OpenID Connect's claims query language as defined in [OIDC] Section 5.5, the RC sends the value of the OpenID Connect "claims" authorization request parameter as a JSON object under the name "claims" in the root of the request.

```
"claims": {
  "id_token" : {
    "email"           : { "essential" : true },
    "email_verified" : { "essential" : true }
  },
  "userinfo" : {
    "name"           : { "essential" : true },
    "picture"        : null
  }
}
```

The contents of the "claims" parameter have the same semantics as they do in OpenID Connect's "claims" authorization request parameter, including all extensions such as [OIDC4IA]. The AS MUST process the claims object in the same way that it would with an OAuth 2 based authorization request.

Note that because this is an independent query object, the "claims" value can augment or alter other portions of the request, namely the "resources" and "subject" fields. This query language uses the fields in the top level of the object to indicate the target for any requested claims. For instance, the "userinfo" target indicates that a returned access token would grant access to the given claims at the UserInfo Endpoint, while the "id_token" target indicates that the claims would be returned in an ID Token as described in Section 3.4.

[[Editor's note: in order to use the "claims" parameter as defined in OIDC, we have to violate the principle of orthogonality in Section 2.9. An alternative approach would be to split up the portions of the claims request, so that "id_token" claims would go into the "subject" field and "userinfo" claims would go into the "resources" request, but this violates the original field definition from OIDC and gets into the territory of defining an identity schema request. This approach would also invalidate extensions to the "claims" standard as each "target" would need to have its own separate mapping to some part of the G NAP protocol.]]

[[Editor's note: I'm not a fan of G NAP defining how OIDC would work at all and would rather that work be done by the O IDF in an extension. However, I think it is important for discussion to see this kind of thing in context with the rest of the protocol, for now. In the future, I would anticipate this would be defined by the O IDF as a relatively small but robust identity layer on top of G NAP.]]

2.9. Extending The Grant Request

The request object MAY be extended by registering new items in a registry TBD (Section 12). Extensions SHOULD be orthogonal to other parameters. Extensions MUST document any aspects where the extension item affects or influences the values or behavior of other request and response objects.

[[Editor's note: we should have more guidance and examples on what possible top-level extensions would look like.]]

3. Grant Response

In response to a RC's request, the AS responds with a JSON object as the HTTP entity body. Each possible field is detailed in the sections below

`continue` Indicates that the RC can continue the request by making an additional request using these parameters. Section 3.1

`access_token` A single access token that the RC can use to call the RS on behalf of the RO. Section 3.2.1

`multiple_access_token` Multiple named access tokens that the RC can use to call the RS on behalf of the RO. Section 3.2.2

`interact` Indicates that interaction through some set of defined mechanisms needs to take place. Section 3.3

`subject` Claims about the RO as known and declared by the AS. Section 3.4

`instance_id` An identifier this RC instance can use to identify itself when making future requests. Section 3.5

`user_handle` An identifier this RC instance can use to identify its current RQ when making future requests. Section 3.5

`error` An error code indicating that something has gone wrong. Section 3.6

In this example, the AS is returning an interaction URL (Section 3.3.1), a callback nonce (Section 3.3.3), and a continuation handle (Section 3.1).

```
{
  "interact": {
    "redirect": "https://server.example.com/interact/4CF492MLVMSW9MKMXKHQ",
    "callback": "MBDOFXG4Y5CVJCX821LH"
  },
  "continue": {
    "access_token": {
      "value": "80UPRY5NM33OMUKMKSKU",
      "key": true
    },
    "uri": "https://server.example.com/tx"
  }
}
```

In this example, the AS is returning a bearer access token (Section 3.2.1) with a management URL and a subject identifier (Section 3.4) in the form of an email address.

```
{
  "access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "key": false,
    "manage": "https://server.example.com/token/PRY5NM33OM4TB8N6BW7OZB8CDFONP
219RP1L"
  },
  "subject": {
    "sub_ids": [ {
      "subject_type": "email",
      "email": "user@example.com",
    } ]
  }
}
```

3.1. Request Continuation

If the AS determines that the request can be continued with additional requests, it responds with the "continue" field. This field contains a JSON object with the following properties.

uri REQUIRED. The URI at which the RC can make continuation requests. This URI MAY vary request, or MAY be stable at the AS if the AS includes an access token. The RC MUST use this value exactly as given when making a continuation request (Section 5).

wait RECOMMENDED. The amount of time in integer seconds the RC SHOULD wait after receiving this continuation handle and calling the URI.

access_token RECOMMENDED. A unique access token for continuing the request, in the format specified in Section 3.2.1. This access token MUST be bound to the RC's key used in the request and MUST NOT be a "bearer" token. This access token MUST NOT be usable at resources outside of the AS. [[Editor's note: Is this a restriction we want to enforce?]] If the AS includes an access token, the RC MUST present the access token in all requests to the continuation URI as described in Section 7.

```

{
  "continue": {
    "access_token": {
      "value": "80UPRY5NM33OMUKMKSKU",
      "key": true
    },
    "uri": "https://server.example.com/continue",
    "wait": 60
  }
}

```

The RC can use the values of this field to continue the request as described in Section 5. Note that the RC MUST sign all continuation requests with its key as described in Section 8. If the AS includes an "access_token", the RC MUST present the access token in its continuation request.

This field SHOULD be returned when interaction is expected, to allow the RC to follow up after interaction has been concluded.

[[Editor's note: The AS can use the optional "access_token" as a credential for the client to manage the grant request itself over time. This is in parallel with access token management as well as RS access in general. If the AS uses the access token, the continuation URL can be static, and potentially even the same as the initial request URL. If the AS does not use an access token here, it needs to use unique URLs in its response and bind the client's key to requests to those URLs - or potentially only allow one request per client at a time. The optionality adds a layer of complexity, but the client behavior is deterministic in all possible cases and it re-uses existing functions and structures instead of inventing something special just to talk to the AS. The optional access token represents a design compromise, but the working group can decide to either require the access token on all requests or to remove the access token functionality and require the security of the continuation requests be based on unique URLs.]]

3.2. Access Tokens

If the AS has successfully granted one or more access tokens to the RC, the AS responds with either the "access_token" or the "multiple_access_token" field. The AS MUST NOT respond with both the "access_token" and "multiple_access_token" fields.

[[Editor's note: I really don't like the dichotomy between "access_token" and "multiple_access_tokens" and their being mutually exclusive, and I think we should design away from this pattern toward something less error-prone.]]

3.2.1. Single Access Token

If the RC has requested a single access token and the AS has granted that access token, the AS responds with the "access_token" field. The value of this field is an object with the following properties.

value REQUIRED. The value of the access token as a string. The value is opaque to the RC. The value SHOULD be limited to ASCII characters to facilitate transmission over HTTP headers within other protocols without requiring additional encoding.

manage OPTIONAL. The management URI for this access token. If provided, the RC MAY manage its access token as described in Section 6. This management URI is a function of the AS and is separate from the RS the RC is requesting access to. This URI MUST NOT include the access token value and SHOULD be different for each access token issued in a request.

resources RECOMMENDED. A description of the rights associated with this access token, as defined in Section 3.2.1. If included, this MUST reflect the rights associated with the issued access token. These rights MAY vary from what was requested by the RC.

expires_in OPTIONAL. The number of seconds in which the access will expire. The RC MUST NOT use the access token past this time. An RS MUST NOT accept an access token past this time. Note that the access token MAY be revoked by the AS or RS at any point prior to its expiration.

key REQUIRED. The key that the token is bound to. If the boolean value "true" is used, the token is bound to the key used by the RC (Section 2.3.2) in its request for access. If the boolean value "false" is used, the token is a bearer token with no key bound to it. Otherwise, the key MUST be an object or string in a format described in Section 2.3.2, describing a public key to which the RC can use the associated private key. The RC MUST be able to dereference or process the key information in order to be able to sign the request.

The following non-normative example shows a single bearer token with a management URL that has access to three described resources.

```

"access_token": {
  "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
  "key": false,
  "manage": "https://server.example.com/token/PRY5NM33OM4TB8N6BW7OZB8CDFONP
219RP1L",
  "resources": [
    {
      "type": "photo-api",
      "actions": [
        "read",
        "write",
        "dolphin"
      ],
      "locations": [
        "https://server.example.net/",
        "https://resource.local/other"
      ],
      "datatypes": [
        "metadata",
        "images"
      ]
    },
    "read", "dolphin-metadata"
  ]
}

```

The following non-normative example shows a single access token bound to the RC's key, which was presented using the detached JWS (Section 8.1) binding method.

```

"access_token": {
  "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
  "key": true,
  "resources": [
    "finance", "medical"
  ]
}

```

If the RC requested multiple access tokens (Section 2.1.3), the AS MUST NOT respond with a single access token structure unless the RC sends the "split_token" flag as described in Section 2.1.4.

[[Editor's note: There has been interest in describing a way for the AS to tell the client both how and where to use the token. This kind of directed access token could allow for some interesting deployment patterns where the client doesn't know much]]

3.2.2. Multiple Access Tokens

If the RC has requested multiple access tokens and the AS has granted at least one of them, the AS responds with the "multiple_access_tokens" field. The value of this field is a JSON object, and the property names correspond to the token identifiers chosen by the RC in the multiple access token request (Section 2.1.3). The values of the properties of this object are access tokens as described in Section 3.2.1.

In this non-normative example, two bearer tokens are issued under the names "token1" and "token2", and only the first token has a management URL associated with it.

```
"multiple_access_tokens": {
  "token1": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "key": false,
    "manage": "https://server.example.com/token/PRY5NM33OM4TB8N6BW7OZB8CD
FONP219RP1L"
  },
  "token2": {
    "value": "UFGLO2FDAFG7VGZZPJ3IZEMN21EVU71FHCARP4J1",
    "key": false
  }
}
```

Each access token corresponds to the named resources arrays in the RC's request (Section 2.1.3).

The multiple access token response MUST be used when multiple access tokens are requested, even if only one access token is issued as a result of the request. The AS MAY refuse to issue one or more of the requested access tokens, for any reason. In such cases the refused token is omitted from the response and all of the other issued access tokens are included in the response the requested names appropriate names.

If the RC requested a single access token (Section 2.1.1), the AS MUST NOT respond with the multiple access token structure unless the RC sends the "split_token" flag as described in Section 2.1.4.

Each access token MAY have different proofing mechanisms. If management is allowed, each access token SHOULD have different management URIs.

[[Editor's note: Do we need to specify that the management URIs are different if we require the token to be presented?]]

When the RO completes interaction at the AS, the AS MUST call the RC's callback URL using the method indicated in the callback request (Section 2.5.3) as described in Section 4.4.1.

If the AS returns a "callback" nonce, the RC MUST NOT continue a grant request before it receives the associated interaction reference on the callback URI.

3.3.4. Display of a Short User Code

If the RC indicates that it can display a short user-typeable code (Section 2.5.4) and the AS supports this mode for the RC's request, the AS responds with a "user_code" field. This field is an object that contains the following members.

code REQUIRED. A unique short code that the user can type into an authorization server. This string MUST be case-insensitive, MUST consist of only easily typeable characters (such as letters or numbers). The time in which this code will be accepted SHOULD be short lived, such as several minutes. It is RECOMMENDED that this code be no more than eight characters in length.

url RECOMMENDED. The interaction URL that the RC will direct the RO to. This URL MUST be stable at the AS such that RCs can be statically configured with it.

```
"interact": {
  "user_code": {
    "code": "A1BC-3DFF",
    "url": "https://srv.ex/device"
  }
}
```

The RC MUST communicate the "code" to the RQ in some fashion, such as displaying it on a screen or reading it out audibly. The "code" is a one-time-use credential that the AS uses to identify the pending request from the RC. When the RO enters this code (Section 4.2) into the AS, the AS MUST determine the pending request that it was associated with. If the AS does not recognize the entered code, the AS MUST display an error to the user. If the AS detects too many unrecognized codes entered, it SHOULD display an error to the user.

The RC SHOULD also communicate the URL if possible to facilitate user interaction, but since the URL should be stable, the RC should be able to safely decide to not display this value. As this interaction mode is designed to facilitate interaction via a secondary device, it is not expected that the RC redirect the RQ to the URL given here at runtime. Consequently, the URL needs to be stable enough that a RC

could be statically configured with it, perhaps referring the RQ to the URL via documentation instead of through an interactive means. If the RC is capable of communicating an arbitrary URL to the RQ, such as through a scannable code, the RC can use the "redirect" (Section 2.5.1) mode for this purpose instead of or in addition to the user code mode.

The interaction URL returned represents a function of the AS but MAY be completely distinct from the URL the RC uses to request access (Section 2), allowing an AS to separate its user-interactive functionality from its back-end security functionality.

[[Editor's note: This is one aspect where the AS might actually be two separate roles. Namely, a delegation server (back end) and interaction server (user-facing).]]

3.3.5. Extending Interaction Mode Responses

Extensions to this specification can define new interaction mode responses in a registry TBD (Section 12). Extensions MUST document the corresponding interaction request.

3.4. Returning User Information

If information about the RO is requested and the AS grants the RC access to that data, the AS returns the approved information in the "subject" response field. This field is an object with the following OPTIONAL properties.

sub_ids An array of subject identifiers for the RO, as defined by [I-D.ietf-secevent-subject-identifiers]. [[Editor's note: privacy considerations are needed around returning identifiers.]]

assertions An object containing assertions as values keyed on the assertion type defined by a registry TBD (Section 12). [[Editor's note: should this be an array of objects with internal typing like the sub_ids? Do we expect more than one assertion per user anyway?]]

updated_at Timestamp in integer seconds indicating when the identified account was last updated. The RC MAY use this value to determine if it needs to request updated profile information through an identity API. The definition of such an identity API is out of scope for this specification.

```

"subject": {
  "sub_ids": [ {
    "subject_type": "email",
    "email": "user@example.com",
  } ],
  "assertions": {
    "id_token": "eyJ..."
  }
}

```

The AS MUST return the "subject" field only in cases where the AS is sure that the RO and the RQ are the same party. This can be accomplished through some forms of interaction with the RO (Section 4).

Subject identifiers returned by the AS SHOULD uniquely identify the RO at the AS. Some forms of subject identifier are opaque to the RC (such as the subject of an issuer and subject pair), while others forms (such as email address and phone number) are intended to allow the RC to correlate the identifier with other account information at the RC. The RC MUST NOT request or use any returned subject identifiers for communication purposes (see Section 2.2). That is, a subject identifier returned in the format of an email address or a phone number only identifies the RO to the AS and does not indicate that the AS has validated that the represented email address or phone number in the identifier is suitable for communication with the current user. To get such information, the RC MUST use an identity protocol to request and receive additional identity claims. While Section 2.8 specifies one such method, other identity protocols could also be used on top of GNAP to convey this information and the details of an identity protocol and associated schema are outside the scope of this specification.

[[Editor's note: subject identifiers here are naturally scoped to the AS; even though using an external identifier like an email address or phone number implies a global namespace in use, the association of that identifier to the current user is still under the view of the AS. Would changing the name to "as_sub_ids" or "local_sub_ids" help convey that point? Would it also be desirable to have an identifier that's globally unique by design? The "iss_sub" type almost gets us there by explicitly calling out the issuer URL, but tuples are hard to deal with in practice and so tend to get ignored in practice in the OIDC space.]]

[[Editor's note: This will need substantial privacy considerations, as this is releasing information about the current user that could be tied to other information at the RC or elsewhere. To facilitate this, should we have another form of identifier that's a globally unique identifier of some form? DIDs could facilitate that kind of namespace.]]

Extensions to this specification MAY define additional response properties in a registry TBD (Section 12).

3.5. Returning Dynamically-bound Reference Handles

Many parts of the RC's request can be passed as either a value or a reference. The use of a reference in place of a value allows for a client to optimize requests to the AS.

Some references, such as for the RC instance's identity (Section 2.3.1) or the requested resources (Section 2.1.2), can be managed statically through an admin console or developer portal provided by the AS or RS. The developer of the RC can include these values in their code for a more efficient and compact request.

If desired, the AS MAY also generate and return some of these references dynamically to the RC in its response to facilitate multiple interactions with the same software. The RC SHOULD use these references in future requests in lieu of sending the associated data value. These handles are intended to be used on future requests.

Dynamically generated handles are string values that MUST be protected by the RC as secrets. Handle values MUST be unguessable and MUST NOT contain any sensitive information. Handle values are opaque to the RC.

[[Editor's note: these constructs used to be objects to allow for expansion to future fields, like a management URI or different presentation types or expiration, but those weren't used in practice. Is that desirable anymore or is collapsing them like this the right direction?]]

All dynamically generated handles are returned as fields in the root JSON object of the response. This specification defines the following dynamic handle returns, additional handles can be defined in a registry TBD (Section 12).

`instance_id` A string value used to represent the information in the "client" object that the RC can use in a future request, as described in Section 2.3.1.

`user_handle` A string value used to represent the current user. The RC can use in a future request, as described in Section 2.4.1.

This non-normative example shows two handles along side an issued access token.

```
{
  "user_handle": "XUT2MFM1XBIKJKSDU8QM",
  "instance_id": "7C7C4AZ9KHRS6X63AJAO",
  "access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "key": false
  }
}
```

[[Editor's note: the ability to dynamically return reference handles allows for an inline version of dynamic registration without needing to go through a discrete registration step, for clients where that makes sense. Currently this is entirely up to the AS to decide when to issue these, but maybe the client should signal that it can receive these handles as part of the request? The new "token flags" construct in Section 2.1.4 almost gets at that, but for a different part of the request structure. Since the client is the component that will know if it's in a position to make use of such reference handles in the future (like a mobile app) or if it's just going to evaporate at the end of a session (like an SPA). Ultimately we need to deal with a range of dynamism, not just the "pre-registered" vs. "non-registered" use cases that OAuth forces us in to.]]

[[Editor's note: The client-bound "instance_id" could serve as the hook we would need for RFC7592 style dynamic client management, including additional components like key rotation. If the AS returns an object instead of a string here, that could include everything that the client would need in order to make REST-style management calls, similar to token management.

```
{
  "client": {
    "instance_id": "7C7C4AZ9KHRS6X63AJAO",
    "manage": "https://example.server.com/client/7C7C4AZ9KHRS6X63AJAO",
    "access_token": {
      "value": "4TB8N6BW7OZB8CDFONP219RP1LT0OS9M2PMHKUR6",
      "key": true
    }
  }
}
```

The client would sign all requests with its key and use the presented access token. A "POST" or "PATCH" request would update client information, including having a method for key rotation using nested signatures. A "DELETE" request would un-register the client, etc.]]

3.6. Error Response

If the AS determines that the request cannot be issued for any reason, it responds to the RC with an error message.

error The error code.

```
{  
  
  "error": "user_denied"  
  
}
```

The error code is one of the following, with additional values available in a registry TBD (Section 12):

user_denied The RO denied the request.

too_fast The RC did not respect the timeout in the wait response.

unknown_request The request referenced an unknown ongoing access request.

[[Editor's note: I think we will need a more robust error mechanism, and we need to be more clear about what error states are allowed in what circumstances. Additionally, is the "error" parameter exclusive with others in the return?]]

3.7. Extending the Response

Extensions to this specification MAY define additional fields for the grant response in a registry TBD (Section 12).

[[Editor's note: what guidance should we give to designers on this?]]

4. Interaction at the AS

If the RC indicates that it is capable of driving interaction with the RO in its request (Section 2.5), and the AS determines that interaction is required and responds to one or more of the RC's interaction modes, the RC SHOULD initiate one of the returned interaction modes in the response (Section 3.3).

When the RO is interacting with the AS, the AS MAY perform whatever actions it sees fit, including but not limited to:

- * authenticate the current user (who may be the RQ) as the RO
- * gather consent and authorization from the RO for access to requested resources and direct information
- * allow the RO to modify the parameters of the request (such as disallowing some requested resources or specifying an account or record)
- * provide warnings to the RO about potential attacks or negative effects of the requested information

[[Editor's note: there are some privacy and security considerations here but for the most part we don't want to be overly prescriptive about the UX, I think.]]

4.1. Interaction at a Redirected URI

When the RO is directed to the AS through the "redirect" (Section 3.3.1) mode, the AS can interact with the RO through their web browser to authenticate the user as an RO and gather their consent. Note that since the RC does not add any parameters to the URL, the AS MUST determine the grant request being referenced from the URL value itself. If the URL cannot be associated with a currently active request, the AS MUST display an error to the RO and MUST NOT attempt to redirect the RO back to any RC even if a callback is supplied (Section 2.5.3).

The interaction URL MUST be reachable from the RO's browser, though note that the RO MAY open the URL on a separate device from the RC itself. The interaction URL MUST be accessible from an HTTP GET request, and MUST be protected by HTTPS or equivalent means.

With this method, it is common for the RO to be the same party as the RQ, since the RC has to communicate the redirection URI to the RQ.

4.2. Interaction at the User Code URI

When the RO is directed to the AS through the "user_code" (Section 3.3.4) mode, the AS can interact with the RO through their web browser to collect the user code, authenticate the user as an RO, and gather their consent. Note that since the URL itself is static, the AS MUST determine the grant request being referenced from the user code value itself. If the user code cannot be associated with a currently active request, the AS MUST display an error to the RO and

MUST NOT attempt to redirect the RO back to any RC even if a callback is supplied (Section 2.5.3).

The user code URL MUST be reachable from the RO's browser, though note that the RO MAY open the URL on a separate device from the RC itself. The user code URL MUST be accessible from an HTTP GET request, and MUST be protected by HTTPS or equivalent means.

While it is common for the RO to be the same party as the RQ, since the RC has to communicate the user code to someone, there are cases where the RQ and RO are separate parties and the authorization happens asynchronously.

4.3. Interaction through an Application URI

When the RC successfully launches an application through the "app" mode (Section 3.3.2), the AS interacts with the RO through that application to authenticate the user as the RO and gather their consent. The details of this interaction are out of scope for this specification.

[[Editor's note: Should we have anything to say about an app sending information to a back-end to get details on the pending request?]]

4.4. Post-Interaction Completion

Upon completing an interaction with the RO, if a "callback" (Section 3.3.3) mode is available with the current request, the AS MUST follow the appropriate method at the end of interaction to allow the RC to continue. If this mode is not available, the AS SHOULD instruct the RO to return to their RC software upon completion. Note that these steps still take place in most error cases, such as when the RO has denied access. This pattern allows the RC to potentially recover from the error state without restarting the request from scratch by modifying its request or providing additional information directly to the AS.

[[Editor's note: there might be some other kind of push-based notification or callback that the client can use, or an out-of-band non-HTTP protocol. The AS would know about this if supported and used, but the guidance here should be written in such a way as to not be too restrictive in the next steps that it can take. Still, it's important that the AS not expect or even allow clients to poll if the client has stated it can take a callback of some form, otherwise that sets up a potential session fixation attack vector that the client is trying to and able to avoid. There has also been a call for post-interaction that doesn't tie into the security of the protocol, like redirecting to a static webpage hosted by the client's company. Would this fit here?]]

The AS MUST create an interaction reference and associate that reference with the current interaction and the underlying pending request. This value MUST be sufficiently random so as not to be guessable by an attacker. The interaction reference MUST be one-time-use.

The AS MUST calculate a hash value based on the RC and AS nonces and the interaction reference, as described in Section 4.4.3. The RC will use this value to validate the return call from the AS.

The AS then MUST send the hash and interaction reference based on the interaction finalization mode as described in the following sections.

4.4.1. Completing Interaction with a Browser Redirect to the Callback URI

When using the "callback" interaction mode (Section 3.3.3) with the "redirect" method, the AS signals to the RC that interaction is complete and the request can be continued by directing the RO (in their browser) back to the RC's callback URL sent in the callback request (Section 2.5.3.1).

The AS secures this callback by adding the hash and interaction reference as query parameters to the RC's callback URL.

hash REQUIRED. The interaction hash value as described in Section 4.4.3.

interact_ref REQUIRED. The interaction reference generated for this interaction.

The means of directing the RO to this URL are outside the scope of this specification, but common options include redirecting the RO from a web page and launching the system browser with the target URL.

```
https://client.example.net/return/123455
?hash=p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R2HZT8BOWYHcLmObM7XHPAdJzT
ZMtKBSaraJ64A
&interact_ref=4IFWWIKYBC2PQ6U56NL1
```

When receiving the request, the RC MUST parse the query parameters to calculate and validate the hash value as described in Section 4.4.3. If the hash validates, the RC sends a continuation request to the AS as described in Section 5.1 using the interaction reference value received here.

4.4.2. Completing Interaction with a Direct HTTP Request Callback

When using the "callback" interaction mode (Section 3.3.3) with the "push" method, the AS signals to the RC that interaction is complete and the request can be continued by sending an HTTP POST request to the RC's callback URL sent in the callback request (Section 2.5.3.2).

The entity message body is a JSON object consisting of the following two fields:

hash REQUIRED. The interaction hash value as described in Section 4.4.3.

interact_ref REQUIRED. The interaction reference generated for this interaction.

```
POST /push/554321 HTTP/1.1
Host: client.example.net
Content-Type: application/json

{
  "hash": "p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R2HZT8BOWYHcLmObM7XHPAd
JzTZMtKBSaraJ64A",
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

When receiving the request, the RC MUST parse the JSON object and validate the hash value as described in Section 4.4.3. If the hash validates, the RC sends a continuation request to the AS as described in Section 5.1 using the interaction reference value received here.

4.4.3. Calculating the interaction hash

The "hash" parameter in the request to the RC's callback URL ties the front channel response to an ongoing request by using values known only to the parties involved. This security mechanism allows the RC to protect itself against several kinds of session fixation and injection attacks. The AS MUST always provide this hash, and the RC MUST validate the hash when received.

[[Editor's note: If the client uses a unique callback URL per request, that prevents some of the same attacks, but without the same cryptographic binding between the interaction and delegation channels. A unique URI would allow the client to differentiate inputs, but it would not prevent an attacker from injecting an unrelated interaction reference into this channel.]]

To calculate the "hash" value, the party doing the calculation first takes the "nonce" value sent by the RC in the interaction section of the initial request (Section 2.5.3), the AS's nonce value from the callback response (Section 3.3.3), and the "interact_ref" sent to the RC's callback URL. These three values are concatenated to each other in this order using a single newline character as a separator between the fields. There is no padding or whitespace before or after any of the lines, and no trailing newline character.

```
VJLO6A4CAYLBXHTR0KRO
MBDOFXG4Y5CVJXCX821LH
4IFWWIKYBC2PQ6U56NL1
```

The party then hashes this string with the appropriate algorithm based on the "hash_method" parameter of the "callback". If the "hash_method" value is not present in the RC's request, the algorithm defaults to "sha3".

[[Editor's note: these hash algorithms should be pluggable, and ideally we shouldn't redefine yet another crypto registry for this purpose, but I'm not convinced an appropriate one already exists. Furthermore, we should be following best practices here whether it's a plain hash, a keyed MAC, an HMAC, or some other form of cryptographic function. I'm not sure what the defaults and options ought to be, but SHA512 and SHA3 were picked based on what was available to early developers.]]

4.4.3.1. SHA3-512

The "sha3" hash method consists of hashing the input string with the 512-bit SHA3 algorithm. The byte array is then encoded using URL Safe Base64 with no padding. The resulting string is the hash value.

```
p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R2HZT8BOWYHcLmObM7XHPAdJzTZMtKBSar
aJ64A
```

4.4.3.2. SHA2-512

The "sha2" hash method consists of hashing the input string with the 512-bit SHA2 algorithm. The byte array is then encoded using URL Safe Base64 with no padding. The resulting string is the hash value.

62SbcD3Xs7L40rjgALA-ymQujoh2LB2hPJyX9v1crlH6ecChZ8BNKkG_HrOKP_Bpj84rh4mC9aE9x7HPB
FcIHw

5. Continuing a Grant Request

While it is possible for the AS to return a Section 3 with all the RC's requested information (including access tokens (Section 3.2) and direct user information (Section 3.4)), it's more common that the AS and the RC will need to communicate several times over the lifetime of an access grant. This is often part of facilitating interaction (Section 4), but it could also be used to allow the AS and RC to continue negotiating the parameters of the original grant request (Section 2).

To enable this ongoing negotiation, the AS returns a "continue" field in the response (Section 3.1) that contains information the RC needs to continue this process with another request, including a URI to access as well as an optional access token to use during the continued requests.

When the RC makes any calls to the continuation URL, the RC MUST present proof of the most recent key associated with this ongoing request by signing the request as described in Section 8. The key in use will be either the key from the initial request (Section 2.3.2) or its most recent rotation. [[Editor's note: we need to have a secure way to rotate the key used for the continuation here. In most cases this will be a rotation for the client instance, since a client without an instance record would likely just present a new key for a new request. In that case it could go with the client management, above - but it doesn't necessarily have to be.]]

For example, here the RC makes a POST request and signs with detached JWS:

```
POST /continue/80UPRY5NM33OMUKMKSKU HTTP/1.1
Host: server.example.com
Detached-JWS: ejy0...
```

If the AS includes an "access_token" in the "continue" response in Section 3.1, the RC MUST include the access token the request as described in Section 7. Note that the access token is always bound to the RC's presented key (or its most recent rotation).

For example, here the RC makes a POST request with the interaction reference, includes the access token, and signs with detached JWS:

```
POST /continue HTTP/1.1
Host: server.example.com
Content-type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Detached-JWS: ejy0...
```

```
{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

The AS MUST be able to tell from the RC's request which specific ongoing request is being accessed. Common methods for doing so include using a unique, unguessable URL for each continuation response, associating the request with the provided access token, or allowing only a single ongoing grant request for a given RC instance at a time. If the AS cannot determine a single active grant request to map the continuation request to, the AS MUST return an error.

The ability to continue an already-started request allows the RC to perform several important functions, including presenting additional information from interaction, modifying the initial request, and getting the current state of the request.

If a "wait" parameter was included in the continuation response (Section 3.1), the RC MUST NOT call the continuation URI prior to waiting the number of seconds indicated. If no "wait" period is indicated, the RC SHOULD wait at least 5 seconds [[Editor's note: what's a reasonable amount of time so as not to DOS the server??]]. If the RC does not respect the given wait period, the AS MUST return an error.

The response from the AS is a JSON object and MAY contain any of the fields described in Section 3, as described in more detail in the sections below.

If the AS determines that the RC can make a further continuation request, the AS MUST include a new "continue" response (Section 3.1). If the continuation was previously bound to an access token, the new "continue" response MUST include a bound access token as well, and this token SHOULD be a new access token. [[Editor's note: this used to be a MUST, but is it safe to back off that requirement?]] If the AS does not return a new "continue" response, the RC MUST NOT make an additional continuation request. If a RC does so, the AS MUST return an error.

For continuation functions that require the RC to send a message body, the body MUST be a JSON object.

5.1. Continuing After a Completed Interaction

When the AS responds to the RC's "callback" parameter as in Section 4.4.1, this response includes an interaction reference. The RC MUST include that value as the field "interact_ref" in a POST request to the continuation URI.

```
POST /continue/80UPRY5NM33OMUKMKSKU HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...
```

```
{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

Since the interaction reference is a one-time-use value as described in Section 4.4.1, if the RC needs to make additional continuation calls after this request, the RC MUST NOT include the interaction reference. If the AS detects an RC submitting the same interaction reference multiple times, the AS MUST return an error and SHOULD invalidate the ongoing request.

The Section 3 MAY contain any newly-created access tokens (Section 3.2) or newly-released subject claims (Section 3.4). The response MAY contain a new "continue" response (Section 3.1) as described above. The response SHOULD NOT contain any interaction responses (Section 3.3). [[Editor's note: This last one might be overly restrictive, since some kinds of interaction could require multiple round trips. We need more examples and experience beyond redirect-based interaction here.]]

For example, if the request is successful in causing the AS to issue access tokens and release subject claims, the response could look like this:

```

{
  "access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "key": false,
    "manage": "https://server.example.com/token/PRY5NM33OM4TB8N6BW7OZB8CDFONP
219RP1L"
  },
  "subject": {
    "sub_ids": [ {
      "subject_type": "email",
      "email": "user@example.com",
    } ]
  }
}

```

With this example, the RC can not make an additional continuation request because a "continue" field is not included.

[[Editor's note: other interaction methods, such as a challenge-response cryptographic protocol, would use a similar construct as here, but have different rules. Would it be reasonable to allow them to be combined? Could this be combined further with the "update" method in Section 5.3?]]

5.2. Continuing During Pending Interaction

When the RC does not include a "callback" parameter, the RC will often need to poll the AS until the RO has authorized the request. To do so, the RC makes a POST request to the continuation URI as in Section 5.1, but does not include a message body.

```

POST /continue HTTP/1.1
Host: server.example.com
Content-type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Detached-JWS: ejy0...

```

The Section 3 MAY contain any newly-created access tokens (Section 3.2) or newly-released subject claims (Section 3.4). The response MAY contain a new "continue" response (Section 3.1) as described above. If a "continue" field is included, it SHOULD include a "wait" field to facilitate a reasonable polling rate by the RC. The response SHOULD NOT contain interaction responses (Section 3.3).

For example, if the request has not yet been authorized by the RO, the AS could respond by telling the RC to make another continuation request in the future. In this example, a new, unique access token has been issued for the call, which the RC will use in its next continuation request.

```
{
  "continue": {
    "access_token": {
      "value": "33OMUKMKSKU80UPRY5NM",
      "key": true
    },
    "uri": "https://server.example.com/continue",
    "wait": 30
  }
}
```

[[Editor's note: Do we want to be more precise about what's expected inside the "continue" object? I think that at least the URI is required, access token required IF used, etc. This is even if they haven't changed since last time, and the client will use whatever value comes back.]]

[[Editor's note: extensions to this might need to communicate to the client what the current state of the user interaction is. This has been done in similar proprietary protocols, but the details of that information tend to be highly application specific. Like "user hasn't logged in yet", "user has logged in but is still sitting at the page", or "user seems to have wandered off". We might be able to provide a decent framework for hanging this kind of stuff on.]]

If the request is successful in causing the AS to issue access tokens and release subject claims, the response could look like this example:

```
{
  "access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "key": false,
    "manage": "https://server.example.com/token/PRY5NM33OM4TB8N6BW7OZB8CDFONP219RP1L"
  },
  "subject": {
    "sub_ids": [ {
      "subject_type": "email",
      "email": "user@example.com",
    } ]
  }
}
```

5.3. Modifying an Existing Request

The RC might need to modify an ongoing request, whether or not tokens have already been issued or claims have already been released. In such cases, the RC makes an HTTP PATCH request to the continuation URI and includes any fields it needs to modify. Fields that aren't included in the request are considered unchanged from the original request.

The RC MAY include the "resources" and "subject" fields as described in Section 2.1 and Section 2.2. Inclusion of these fields override any values in the initial request, which MAY trigger additional requirements and policies by the AS. For example, if the RC is asking for more access, the AS could require additional interaction with the RO to gather additional consent. If the RC is asking for more limited access, the AS could determine that sufficient authorization has been granted to the RC and return the more limited access rights immediately. [[Editor's note: We could state something like "resources and subject MUST NOT be the same as in the initial or previous request" to enforce that this really is a change, but is there value in calling that out here? Somehow we do probably want to tell the AS to not let a client simply post the same request here to rotate access tokens now that we've got an explicit function for that, right?]]

The RC MAY include the "interact" field as described in Section 2.5. Inclusion of this field indicates that the RC is capable of driving interaction with the RO, and this field replaces any values from a previous request. The AS MAY respond to any of the interaction responses as described in Section 3.3, just like it would to a new request.

The RC MAY include the "user" field as described in Section 2.4 to present new assertions or information about the RQ. [[Editor's note: This would allow the client to do things like gather the user's identifiers post-request, or gather an assertion from an on-device element that the AS can verify. It opens up potential avenues for trouble if the user here is different from the RO that's already showed up at the AS or race conditions if the RQ's identity changes mid-stream. But that said, this seems important for multi-log-in cases and the like, probably.]]

The RC MUST NOT include the "client" section of the request. [[Editor's note: We do not want the client to be able to get swapped out from underneath the user, especially post-consent. However, including this field in a PATCH update request might be the place to define key rotation for the grant request itself, but we'd need to be very careful of how that works. And it feels like it might have

consequences outside of the request, such as rotating the key for all ongoing grants for a given client instance, which isn't really desirable here. We need a lot more discussion and engineering on this before including it.]]

The RC MAY include post-interaction responses such as described in Section 5.1. [[Editor's note: it seems a little odd to include this in a request but I can't see a reason to not allow it.]]

Modification requests MUST NOT alter previously-issued access tokens. Instead, any access tokens issued from a continuation are considered new, separate access tokens. The AS MAY revoke existing access tokens after a modification has occurred. [[Editor's note: this might be subject to the "multi_token" flag, but since we're creating a NEW access token and not rotating an existing one, this seems to be a different use case.]]

Modification requests MAY result in previously-issued access tokens being revoked. [[Editor's note: there is a solid argument to be made for always revoking old access tokens here, but we need more discussion on the boundaries for such a requirement. If they stick around, it does make a "read" request weird because now we've got multiple access tokens sticking around associated with a grant request and no good place to put them.]]

If the modified request can be granted immediately by the AS, the Section 3 MAY contain any newly-created access tokens (Section 3.2) or newly-released subject claims (Section 3.4). The response MAY contain a new "continue" response (Section 3.1) as described above. If interaction can occur, the response SHOULD contain interaction responses (Section 3.3) as well.

For example, an RC initially requests a set of resources using references:

```
POST /tx HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...
```

```
{
  "resources": [
    "read", "write"
  ],
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.example.net/return/123455",
      "nonce": "LKLTi25DK82FX4T4QFZC"
    }
  },
  "client": "987YHGRT56789IOLK"
}
```

Access is granted by the RO, and a token is issued by the AS. In its final response, the AS includes a "continue" field:

```
{
  "continue": {
    "access_token": {
      "value": "80UPRY5NM33OMUKMKSKU",
      "key": true
    },
    "uri": "https://server.example.com/continue",
    "wait": 30
  },
  "access_token": ...
}
```

This allows the RC to make an eventual continuation call. The RC realizes that it no longer needs "write" access and therefore modifies its ongoing request, here asking for just "read" access instead of both "read" and "write" as before.

```

PATCH /continue HTTP/1.1
Host: server.example.com
Content-type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Detached-JWS: ejy0...

```

```

{
  "resources": [
    "read"
  ]
  ...
}

```

The AS replaces the previous "resources" from the first request, allowing the AS to determine if any previously-granted consent already applies. In this case, the AS would likely determine that reducing the breadth of the requested access means that new access tokens can be issued to the RC. The AS would likely revoke previously-issued access tokens that had the greater access rights associated with them.

```

{
  "continue": {
    "access_token": {
      "value": "M33OMUK80UPRY5NMKSKU",
      "key": true
    },
    "uri": "https://server.example.com/continue",
    "wait": 30
  },
  "access_token": ...
}

```

For another example, the RC initially requests read-only access but later needs to step up its access. The initial request could look like this example.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...
```

```
{
  "resources": [
    "read"
  ],
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.example.net/return/123455",
      "nonce": "LKLTi25DK82FX4T4QFZC"
    }
  },
  "client": "987YHGRT56789IOLK"
}
```

Access is granted by the RO, and a token is issued by the AS. In its final response, the AS includes a "continue" field:

```
{
  "continue": {
    "access_token": {
      "value": "80UPRY5NM33OMUKMKSKU",
      "key": true
    },
    "uri": "https://server.example.com/continue",
    "wait": 30
  },
  "access_token": ...
}
```

This allows the RC to make an eventual continuation call. The RC later realizes that it now needs "write" access in addition to the "read" access. Since this is an expansion of what it asked for previously, the RC also includes a new interaction section in case the AS needs to interact with the RO again to gather additional authorization. Note that the RC's nonce and callback are different from the initial request. Since the original callback was already used in the initial exchange, and the callback is intended for one-time-use, a new one needs to be included in order to use the callback again.

[[Editor's note: the net result of this is that interaction requests are really only meant to be responded to exactly once by the AS. This isn't spelled out explicitly, but could be included in Section 2.5 and/or Section 3.3.]]

```
PATCH /continue HTTP/1.1
Host: server.example.com
Content-type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Detached-JWS: ejy0...
```

```
{
  "resources": [
    "read", "write"
  ],
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.example.net/return/654321",
      "nonce": "K82FX4T4LKLTi25DQFZC"
    }
  }
}
```

From here, the AS can determine that the RC is asking for more than it was previously granted, but since the RC has also provided a mechanism to interact with the RO, the AS can use that to gather the additional consent. The protocol continues as it would with a new request. Since the old access tokens are good for a subset of the rights requested here, the AS might decide to not revoke them. However, any access tokens granted after this update process are new access tokens and do not modify the rights of existing access tokens.

5.4. Getting the Current State of a Grant Request

If the RC needs to get the current state of an ongoing grant request, it makes an HTTP GET request to the continuation URI. This request MUST NOT alter the grant request in any fashion, including causing the issuance of new access tokens or modification of interaction parameters.

The AS MAY include existing access tokens and previously-released subject claims in the response. The AS MUST NOT issue a new access token or release a new subject claim in response to this request.

```
GET /continue HTTP/1.1
Host: server.example.com
Content-type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Detached-JWS: ejy0...
```

The response MAY include any fields described Section 3 that are applicable to this ongoing request, including the most recently issued access tokens, any released subject claims, and any currently active interaction modes. The response MAY contain a new "continue" response (Section 3.1) as described above.

[[Editor's note: I'm a little dubious about the need for this particular function in reality, but including it for completeness sake. There are a lot of questions we need to answer, such as whether it's safe to include access tokens and claims in the response of this kind of "read" at all, and whether it makes sense to include items like interaction nonces in the response. This discussion should be driven by the use cases calling for this "read" functionality. There have been similar functions within proprietary protocols where the client calls an endpoint at the AS to figure out where the user is in the interaction process at the AS, letting the client provide a smarter UI. It doesn't seem like we could do that in depth here since it would be highly application specific, but that might be a good example of how to extend a response and give a client extra information.]]

5.5. Canceling a Grant Request

If the RC wishes to cancel an ongoing grant request, it makes an HTTP DELETE request to the continuation URI.

```
DELETE /continue HTTP/1.1
Host: server.example.com
Content-type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Detached-JWS: ejy0...
```

If the request is successfully cancelled, the AS responds with an HTTP 202. The AS MUST revoke all associated access tokens, if possible.

6. Token Management

If an access token response includes the "manage" parameter as described in Section 3.2.1, the RC MAY call this URL to manage the access token with any of the actions defined in the following sections. Other actions are undefined by this specification.

The access token being managed acts as the access element for its own management API. The RC MUST present proof of an appropriate key along with the access token.

If the token is sender-constrained (i.e., not a bearer token), it MUST be sent with the appropriate binding for the access token (Section 7).

If the token is a bearer token, the RC MUST present proof of the same key identified in the initial request (Section 2.3.2) as described in Section 8.

The AS MUST validate the proof and assure that it is associated with either the token itself or the RC the token was issued to, as appropriate for the token's presentation type.

[[Editor's note: Should we allow for "update" to an access token by the client posting new information from a "request"? It seems this might make things weird since an access token is generally considered an unchanging thing, and the client could always request a new access token if they're allowed to continue the grant request post-issuance as in Section 5.3. There's also a possibility of being able to "read" a token's state with a GET, much like token introspection but using the token's/client's key instead of the RS key. But would a client need to "read" a token state after issuance? Is there a security risk to offering that functionality? Introspection is nearly always relegated to RS calls in practice since the client is focused on using the token at the RS. The client can always read the state of the grant itself, separately.]]

6.1. Rotating the Access Token

The RC makes an HTTP POST to the token management URI, sending the access token in the appropriate header and signing the request with the appropriate key.

```
POST /token/PRY5NM33OM4TB8N6BW7OZB8CDFONP219RP1L HTTP/1.1
Host: server.example.com
Authorization: GNAP OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
Detached-JWS: eyj0....
```

[[Editor's note: This could alternatively be an HTTP PUT verb, since we are telling the AS that we want to replace the token. However, we are not providing the information we want to replace the token with, and in fact that's up to the AS entirely, not the client. For that reason, I think a POST still makes the most sense.]]

The AS validates that the token presented is associated with the management URL, that the AS issued the token to the given RC, and that the presented key is appropriate to the token.

If the access token has expired, the AS SHOULD honor the rotation request to the token management URL since it is likely that the RC is attempting to refresh the expired token. To support this, the AS MAY apply different lifetimes for the use of the token in management vs. its use at an RS. An AS MUST NOT honor a rotation request for an access token that has been revoked, either by the AS or by the RC through the token management URI (Section 6.2).

If the token is validated and the key is appropriate for the request, the AS MUST invalidate the current access token associated with this URL, if possible, and return a new access token response as described in Section 3.2.1, unless the "multi_token" flag is specified in the request. [[Editor's note: We could also use different verbs to signal whether or not the old token should be kept around or not, instead of using a token flag to do this.]] The value of the access token MUST NOT be the same as the current value of the access token used to access the management API. The response MAY include an updated access token management URL as well, and if so, the RC MUST use this new URL to manage the new access token.

[[Editor's note: the net result is that the client's always going to use the management URL that comes back. But should we let the server omit it from the response if it doesn't change? That seems like an odd optimization that doesn't help the client.]]

```

{
  "access_token": {
    "value": "FP6A8H6HY37MH13CK76LBZ6Y1UADG6VEUPEER5H2",
    "key": false,
    "manage": "https://server.example.com/token/PRY5NM33OM4TB8N6BW7OZB8CDFONP
219RP1L",
    "resources": [
      {
        "type": "photo-api",
        "actions": [
          "read",
          "write",
          "dolphin"
        ],
        "locations": [
          "https://server.example.net/",
          "https://resource.local/other"
        ],
        "datatypes": [
          "metadata",
          "images"
        ]
      },
      "read", "dolphin-metadata"
    ]
  }
}

```

[[Editor's note: If the client is using its own key as the proof, like with a bearer access token, the AS is going to need to know if the client's key has been rotated. We don't have a mechanism for rotating the token's key or the client's key yet either - so that could occur through this management function as well.]]

6.2. Revoking the Access Token

If the RC wishes to revoke the access token proactively, such as when a user indicates to the RC that they no longer wish for it to have access or the RC application detects that it is being uninstalled, the RC can use the token management URI to indicate to the AS that the AS should invalidate the access token for all purposes.

The RC makes an HTTP DELETE request to the token management URI, presenting the access token and signing the request with the appropriate key.

```
DELETE /token/PRY5NM33OM4TB8N6BW7OZB8CDFONP219RP1L HTTP/1.1
Host: server.example.com
Authorization: GNAP OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
Detached-JWS: eyj0....
```

If the key presented is associated with the token (or the RC, in the case of a bearer token), the AS MUST invalidate the access token, if possible, and return an HTTP 204 response code.

204 No Content

Though the AS MAY revoke an access token at any time for any reason, the token management function is specifically for the RC's use. If the access token has already expired or has been revoked through other means, the AS SHOULD honor the revocation request to the token management URL as valid, since the end result is still the token not being usable.

7. Using Access Tokens

The method the RC uses to send an access token to the RS depends on the value of the "key" and "proof" parameters in the access token response (Section 3.2.1).

If the key value is the boolean "false", the access token is a bearer token sent using the HTTP Header method defined in [RFC6750].

```
Authorization: Bearer OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
```

The form parameter and query parameter methods of [RFC6750] MUST NOT be used.

If the "key" value is the boolean "true", the access token MUST be sent to the RS using the same key and proofing mechanism that the RC used in its initial request.

If the "key" value is an object, the value of the "proof" field within the key indicates the particular proofing mechanism to use. The access token is sent using the HTTP authorization scheme "GNAP" along with a key proof as described in Section 8 for the key bound to the access token. For example, a "jwsd"-bound access token is sent as follows:

```
Authorization: GNAP OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
Detached-JWS: eyj0....
```

[[Editor's note: I don't actually like the idea of using only one header type for differently-bound access tokens. Perhaps instead these values should somehow reflect the key binding types. Maybe there can be multiple fields after the "GNAP" keyword using structured headers? Or a set of derived headers like GNAP-mtls? This might also be better as a separate specification, like it was in OAuth 2. However, access tokens should be able to use any key binding mechanisms here, plus bearer.]]

8. Binding Keys

Any keys presented by the RC to the AS or RS MUST be validated as part of the request in which they are presented. The type of binding used is indicated by the proof parameter of the key section in the initial request Section 2.3.2. Values defined by this specification are as follows:

jwsd A detached JWS signature header

jws Attached JWS payload

mtls Mutual TLS certificate verification

dpop OAuth Demonstration of Proof-of-Possession key proof header

httpsig HTTP Signing signature header

oauthpop OAuth PoP key proof authentication header

Additional proofing methods are defined by a registry TBD (Section 12).

All key binding methods used by this specification MUST cover all relevant portions of the request, including anything that would change the nature of the request, to allow for secure validation of the request by the AS. Relevant aspects include the URI being called, the HTTP method being used, any relevant HTTP headers and values, and the HTTP message body itself. The recipient of the signed message MUST validate all components of the signed message to ensure that nothing has been tampered with or substituted in a way that would change the nature of the request.

When used in the GNAP delegation protocol, these key binding mechanisms allow the AS to ensure that the keys presented by the RC in the initial request are in control of the party calling any follow-up or continuation requests. To facilitate this requirement, all keys in the initial request Section 2.3.2 MUST be proved in all continuation requests Section 5 and token management requests

Section 6, modulo any rotations on those keys over time that the AS knows about. The AS MUST validate all keys presented by the RC (Section 2.3.2) or referenced in an ongoing request for each call within that request.

[[Editor's note: We are going to need a way for a client to rotate its keys securely, even while an ongoing grant is in effect.]]

When used to bind to an access token, the

8.1. Detached JWS

This method is indicated by "jwsd" in the "proof" field. A JWS [RFC7515] signature object is created as follows:

The header of the JWS MUST contain the "kid" field of the key bound to this RC for this request. The JWS header MUST contain an "alg" field appropriate for the key identified by kid and MUST NOT be "none". The "b64" field MUST be set to "false" and the "crit" field MUST contain at least "b64" as specified in [RFC7797]

To protect the request, the JWS header MUST contain the following additional fields.

htm The HTTP Method used to make this request, as an uppercase ASCII string.

htu The HTTP URI used for this request, including all path and query components.

ts A timestamp of the request in integer seconds

at_hash When to bind a request to an access token, the access token hash value. Its value is the base64url encoding of the left-most half of the hash of the octets of the ASCII representation of the "access_token" value, where the hash algorithm used is the hash algorithm used in the "alg" header parameter of the JWS's JOSE Header. For instance, if the "alg" is "RS256", hash the "access_token" value with SHA-256, then take the left-most 128 bits and base64url encode them.

[[Editor's note: It's not the usual practice to put additional information into the header of a JWS, but this keeps us from having to normalize the body serialization. Alternatively, we could add all these fields to the body of the request, but then it gets awkward for non-body requests like GET/DELETE.]]

The payload of the JWS object is the serialized body of the request, and the object is signed according to detached JWS [RFC7797].

The RC presents the signature in the Detached-JWS HTTP Header field.
[[Editor's Note: this is a custom header field, do we need this? It seems like the best place to put this.]]

```

POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Detached-JWS: eyJiNjQiOmZhbnHN1LCJhbGciOiJSUzI1NiIsImtpZCI6Inh5ei0xIn0.
.Y287HMTaY0EegEjoTd_04a4GC6qV48GgVbGKOhHdJnDtD0VuU1VjLfwne8AuUY3U7e8
9zUWwXLnAYK_BiS84M8Esrfvmv8yDLWzqveeIpcN5_ysveQnYt9Dqi32w6IOtAywkNUD
ZeJEdc3z5s9Ei8qrYFN2fxcu28YS4e8e_cHTK57003WJu-wFn2TJUAbHuqvUsyTb-nz
YOKxuCKlqQItJF7E-cwSb_xULu-3f77BEU_vGbNYo5ZBa2B7UHO-kWNMSgbW2yeNNLbL
C18Kv80GF22Y7SbZt0e2TwnR2Aa2zksuUbnntQ5c7a1-gxtnXzuIKa340ekrnyqElhmVW
peQ

```

```

{
  "resources": [
    "dolphin-metadata"
  ],
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.foo",
      "nonce": "VJLO6A4CAYLBXHTR0KRO"
    }
  },
  "client": {
    "proof": "jwsd",
    "key": {
      "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "kid": "xyz-1",
        "alg": "RS256",
        "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8
xYJCNaOKNJn_Oz0YhdHbXTeW05AcoyspDWJbN5w_7bdWDxgpD-y6jnD1u9YhBOCWObNPF
vpkTM8LC7SdXGRKx2k8Me2r_GssYlyRpqvpBLY5-ejCywKRBfctRcnhTTGNztbbDBUyD
SWmFMVChE5mXT4cL0BwrZC6S-uu-LAx06aKwQOPwYOGos1K8WPmlyGdkaA1uF_FpS6LS
63WYPHi_Ap2B7_8Wbw4ttzbMS_doJvuDagW8A1Ip3fXFAHtRACk7rdI4_Xln66hJxFe
kpdfWdiPQddQ6YlcK2U3obvUg7w"
      }
    },
    "display": {
      "name": "My Client Display Name",
      "uri": "https://example.net/client"
    }
  }
}

```

If the request being made does not have a message body, such as an HTTP GET, OPTIONS, or DELETE method, the JWS signature is calculated over an empty payload.

When the server (AS or RS) receives the Detached-JWS header, it MUST parse its contents as a detached JWS object. The HTTP Body is used as the payload for purposes of validating the JWS, with no transformations.

[[Editor's note: this is a potentially fragile signature mechanism. It doesn't protect arbitrary headers or other specific aspects of the request, but it's simple to calculate and useful for body-driven requests, like the client to the AS. Additionally it is potentially fragile since a multi-tier system could parse the payload and pass the parsed payload downstream with potential transformations, making downstream signature validation impossible. We might want to remove this in favor of general-purpose HTTP signing, or at least provide guidance on its use.]]

8.2. Attached JWS

This method is indicated by "jws" in the "proof" field. A JWS [RFC7515] signature object is created as follows:

The header of the JWS MUST contain the "kid" field of the key bound to this RC for this request. The JWS header MUST contain an "alg" field appropriate for the key identified by kid and MUST NOT be "none".

To protect the request, the JWS header MUST contain the following additional fields.

htm The HTTP Method used to make this request, as an uppercase ASCII string.

htu The HTTP URI used for this request, including all path and query components.

ts A timestamp of the request in integer seconds

at_hash When to bind a request to an access token, the access token hash value. Its value is the base64url encoding of the left-most half of the hash of the octets of the ASCII representation of the "access_token" value, where the hash algorithm used is the hash algorithm used in the "alg" header parameter of the JWS's JOSE Header. For instance, if the "alg" is "RS256", hash the "access_token" value with SHA-256, then take the left-most 128 bits and base64url encode them.

907/p6BW/LV1NCgYB1QtFSfGxowqb9FRIMD2kvMSm00EMxgwZ6k6spa+jk0IsI3k
lwLW9b+Tfn/daUbIDctxeJneq2anQyU2znBgQl6KILDSF4eaOq1But/KNZHHazJh

```
{
  "resources": [
    "dolphin-metadata"
  ],
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.foo",
      "nonce": "VJLO6A4CAYLBXHTR0KRO"
    }
  },
  "client": {
    "display": {
      "name": "My Client Display Name",
      "uri": "https://example.net/client"
    },
    "key": {
      "proof": "mtls",
      "cert": "MII EHDCCA wSgAwIBAgIBATANBgkqhkiG9w0BAQsFADCBmJE3
MDUGA1UEAwuQmVzcG9rZSBFbmdpbmVlcmluZyBSb290IENlcnRpZmljYXRlIEF1d
Ghvcml0eTELMAKGA1UECAwCTUExCzAJBgNVBAYTA1VTMRkwFwYJKoZIhvcNAQkBFg
p jYUBic3BrLmlvMRwwGgYDVQQKDBNCZXNwb2t1IEVuz2luZWVyaW5nMQwwCgYDVQQ
LDANNVEkwHhcNMTEwMDI5WhcNMjQwNDA4MjE0MDI5WjB8MRIwEAYDVQQDD
DA1sb2NhbGhvc3QxQzAJBgNVBAGMAK1BMQswCQYDVQQGEwJVUzEgMB4GCSqGSIb3D
QEJARYRdGxzY2xpZW50QGJzcGsuaw8xHDAaBgNVBAoME0Jlc3Bva2UgRW5naW5lZX
JpbmcxDDAKBgNVBASMA01USTCCASIdQYJKoZIhvcNAQEBBQADggEPADCCAQoCggE
BAMmaXQHbs/wc1RpsQ6Orzf6rN+q2i jaZbQxD8oi+XaaN0P/gnEl3JqQduvdq77Om
J4bQLokqs d0BexnI07Njs18nkDDYpe8rNve5TjyUDCfbwgS7U1CluYenXmNQbaYND
OmCdHwwUjV4kKREg6DGAX220q7+VHPTeeFgyw4kQgWRSfDENWY3KUXJ1b/vKR6lQ+
aOJytkvj8kVZQtWupPbvwoJe0na/ISNAOhL74w20DWWoDKoNltXsEtf1N1jVoi5nq
smZQc jfjt6LO0T701OX3Cwu2xWx8KZ3n/2ocuRqKEJHqUGfeDtuQNT6Jz79v/OTr8
puLWaD+uyk6NbtGjoQsCAwEAAaOBiTCBhjAJBgNVHRMEA jAAMAsGA1UdDwQEAwIF4
DBsBgNVHREEZTBjgglsb2NhbGhvc3SCD3Rsc2NsaWVudC5sb2NhbIcEwKgBBIERdG
xzY2xpZW50QGJzcGsuaw+GF2h0dHA6Ly90bHNjbG11bnQubG9jYXVwvhhNzc2g6dGx
zY2xpZW50LmxvY2FsMA0GCSqGSIb3DQEBCwUAA4IBAQCCKv8W1LrT4Z5NazaUrYt1
TF+2v0tvZBQ7qzJQjlOqAcvxry/d2zyhiRCRS/v318YCJBEv4Iq2W3I3JMMYAYEe2
573HzT7rH3xQP12yZyRQnetdiVM1Z1KaXwfrPDLs72hUeELtxIcfZ0M085jLboXhu
fHI6kqm3NCyCCTihe2ck5RmCc5l2KBO/vAHF0ihhFOOObylv6qbPHQcxAU6rEb907
/p6BW/LV1NCgYB1QtFSfGxowqb9FRIMD2kvMSm00EMxgwZ6k6spa+jk0IsI3klwLW
9b+Tfn/daUbIDctxeJneq2anQyU2znBgQl6KILDSF4eaOq1But/KNZHHazJh"
    }
  }
}
```

[[Editor's note: This method requires no changes to the HTTP message itself, since the security relies on the TLS layer. However, the application level will need to validate that the certificate key used in the request is the one expected for the specific request.]]

8.4. Demonstration of Proof-of-Possession (DPoP)

This method is indicated by "dpop" in the "proof" field. The RC creates a Demonstration of Proof-of-Possession signature header as described in [I-D.ietf-oauth-dpop] section 2. In addition to the required fields, the DPoP body MUST also contain a digest of the request body:

digest Digest of the request body as the value of the Digest header defined in [RFC3230].

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IlJTMjU2IiwiaWVudCIsImFsZyI6IlJTMjU2IiwibG9wI6Inp3Q1RfM2J4LWdsYmJlcmhlWXBZcFJXaVksSS1uRWFNUnBablJySWpDczZiX2VteVRrQmtEREVqU3lzaTM4T0M3M2hqMS1XZ3hjUGRLTkdaU1vSDNRWmVuMU1leXloUXBMSKcxLW9MTkxxbTdWVh0ZFl6U2RDOU8zLW9peXk4eWtPNF1VeU5aclJSZlBjaWhkUUNiTl9PQzhRdWdtZz1yZ05ETlNxcHBkYU51YXMxb3Y5UHhZdnhxcnoxLThiYTdna0QwMFlFQ1h1YU1wNXVNYVZhZUhYLU9fV012WVhpY2c2STVqN1M0NFZOVTY1VkZ3dS1BbHluVHhRZE1BV1AzY1l4VlZ5NnAzLTdlVEpva3ZqWVRGcWdEVkRaOGxVWGJyNXl1DVG5SaG5oSmZjNWakRfbWFSUmU4LXRPcUs1TlNEbEhUeTznRDlOcWRHQ20tUG0zUSJ9fQ.eyJodHRwX21ldGhvZCI6IlBPU1QiLCJodHRwX3VyaSI6Imh0dHA6XC9cL2hvc3QuZG9ja2VyLmludGVybWFSOjk4MzRcL2FwaVwvYXNcL3RyYW5zYWN0aW9uIiwiaWF0IjoxNTcyNjQyNjEzLCJqdGkiOiJlIam9IcmpnbTJ5QjR4N2pBNX15RyJ9.aUhftvfw2NoW3M7durkopReTvONng1fOzbWjAlKNSLL0qIwDgFG39XUyNvwQ230BIwe6IuvTQ2UBBPk1PAfJhDTKd8KHEAfidNB-LzU0zhDetLg30yLFzIpcEBMLCjb0TEsmXadvxuNkEzFRL-Q-QCg0AXSF1h57eAqZV8SYF4CQK9OUV6fIWwxLDd3cVTx83MgyCNnvFlG_HDyim1Xx-rxV4ePd1vgDeRubFb6QWjiKEO7vj1APv32dsux67gZYiUpjm0wEZprjLG0a07R984KLeK1XPjXgViEwEdlirUmpVyT9tyEYqGrTfm5uautELgMls9sgSyE929woZ59elg
```

```
{
  "resources": [
    "dolphin-metadata"
  ],
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.foo",
      "nonce": "VJLO6A4CAYLBXHTROKRO"
    }
  }
}
```

```

    },
    "client": {
      "display": {
        "name": "My Client Display Name",
        "uri": "https://example.net/client"
      },
      "proof": "dpop",
      "key": {
        "jwk": {
          "kty": "RSA",
          "e": "AQAB",
          "kid": "xyz-1",
          "alg": "RS256",
          "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8xYJ
CCNaOKNjN_Oz0YhdHbXTeW05AoyspDWJbN5w_7bdWDxgpD-y6jnD1u9YhBOCWObNPFvvpkTM
8LC7SdXGRKx2k8Me2r_GssYlyRpqvvpBLY5-ejCywKRBfctRcnhTTGNztbbDBUyDSWmFMVCH
e5mXT4cL0BwrZC6S-uu-LAx06aKwQOPwYOGos1K8WPmlyGdkaAluF_FpS6LS63WYPHi_Ap2
B7_8Wbw4ttzbMS_doJvuDagW8A1Ip3fXFAHtRAcKw7rdI4_Xln66hJxFekpdfWdiPQddQ6Y
1cK2U3obvUg7w"
        }
      }
    }
  }
}

```

[[Editor's note: this method requires duplication of the key in the header and the request body, which is redundant and potentially awkward. The signature also doesn't protect the body of the request.]]

8.5. HTTP Signing

This method is indicated by "httpsig" in the "proof" field. The RC creates an HTTP Signature header as described in [I-D.ietf-httpbis-message-signatures] section 4. The RC MUST calculate and present the Digest header as defined in [RFC3230] and include this header in the signature.

```

POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Content-Length: 716
Signature: keyId="xyz-client", algorithm="rsa-sha256",
  headers="(request-target) digest content-length",
  signature="TkehmgK7GD/z4jGkmcHS67cjVRgm3zVQN1NrrXW32Wv7d
u0VNEIVI/dMhe0WlHC93NP3ms91i2WOW5r5B6qow6TNx/82/6W84p5jqF
YuYfTkKYZ69GbfgXkYV9gaT++dl5kvZQjVk+KZTldzpAzv8hdk9n087Xi
rj7qe2mdAGE1LLc3YvXwNxCQh82sa5rXHqtNT1077fiDvSVYeced0UEm
rWwErVgr7sijtbtTohC4FJLuJ0nG/KJUcIG/FTchW9rd6dHoBnY43+3Dzj

```

CithXpdH5u4VX3TBe6GJDO6Mkzc6vB+67OWzPwhYTplUiFFV6UZCsDEeu
 Sa/UelyLEAMg==""]}

Digest: SHA=oZz2O3kg5SEFAhmr0xEBbc4jEfo=

```
{
  "resources": [
    "dolphin-metadata"
  ],
  "interact": {
    "redirect": true,
    "callback": {
      "method": "push",
      "uri": "https://client.foo",
      "nonce": "VJLO6A4CAYLBXHTR0KRO"
    }
  },
  "client": {
    "display": {
      "name": "My Client Display Name",
      "uri": "https://example.net/client"
    },
    "proof": "httpsig",
    "key": {
      "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "kid": "xyz-1",
        "alg": "RS256",
        "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_J
tffXyaSx8xYJCCNaOKNJn_Oz0YhdHbXTeW05AoyspDWJbN5w_7bdWDxgpD-
y6jnDlu9YhBOCWObNPFvpkTM8LC7SdXGRKx2k8Me2r_GssYlyRpgvpBlY5-
ejCywKRBfctRcnhTTGNztbbDBUyDSWmFMVCHe5mXT4cL0BwrZC6S-uu-LAx
06aKwQOPwYOGos1K8WPmlyGdkaA1uF_FpS6LS63WYPHi_Ap2B7_8Wbw4ttz
bMS_doJvuDagW8A1Ip3fXFAHtRACKw7rdI4_Xln66hJxFekpdfWdiPQddQ6
Y1cK2U3obvUg7w"
      }
    }
  }
}
```

When used to present an access token as in Section 7, the Authorization header MUST be included in the signature.

8.6. OAuth Proof of Possession (PoP)

This method is indicated by "oauthpop" in the "proof" field. The RC creates an HTTP Authorization PoP header as described in [I-D.ietf-oauth-signed-http-request] section 4, with the following additional requirements:

- * The at (access token) field MUST be omitted unless this method is being used in conjunction with an access token as in Section 7. [[Editor's note: this is in contradiction to the referenced spec which makes this field mandatory.]]
- * The b (body hash) field MUST be calculated and supplied, unless there is no entity body (such as a GET, OPTIONS, or DELETE request).
- * All components of the URL MUST be calculated and supplied
- * The m (method) field MUST be supplied

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
PoP: eyJhbGciOiJSUzI1NiIsImp3ayI6eyJrdHkiOiJSU0EiLCJlIjoi
QVFBQIiIsImtpZCI6Inh5e1ljbjGllbnQiLCJhbGciOiJSUzI1NiIsIm4iOi
iJ6d0NUXzNieClnbGJiSHJoZVlwWXBSV21ZOUktbkVhTVJwWm5ScklqQ3
M2Yl9lbXlUa0JrRERFa1N5c2kzOE9DNzNoajEtV2d4Y1BkS05HWn1Jb0g
zUVp1bjFNS3l5aFFwTEpHMS1vTE5McW03cFhYdGRZelNkQz1PMylvaXl5
OHlrTzRZVXl0WnJSUmZQY2loZFFDYk9fT0M4UXVnbWc5cmdORE9TcXBwZ
GFOZWZfZW92OVb4WXZ4cXJ6MS04SGE3Z2tEMDBZRUNYSGFCMDV1TWVYVW
RlC1PXldJd1lYawnNkkl1ajZTNDRTlU2NVZCd3UtQWx5b1R4UWRNQVd
QM2JZeFZWeTZwMy03ZVRKb2t2a1lURnFnRFZEWjhsVVhicjV5Q1RuUmhu
aEpndmYzVmpEX2lhbE5lOC10T3FLNU9TRGxIVHk2Z0Q5TnFkR0NtLVBtM
1EifX0.eyJwIjoiXC9hcGlzL2FzXC90cmFuc2FjdGlvbiIsImIiOiJxa0
lPYkdOeERhZVBVTZnc3NnFjamtqSXNFRmxDb3g5bTU5NFMOM0RkU0xBiwi
idSI6Imhvc3QuZG9ja2VyLmludGVybmFsIiwiaCI6W1siQWNjZXB0Iiw1
Q29udGVudC1UeXB1Iiw1Q29udGVudC1MZW5ndGgiXSwiVjQ2OUhFWGx6S
k9kQTZmQU50cmppKdFhTd3pjSGRqMULoOGk5M0h3bEVHYyJdLCJtIjoiUE
9TVCIiInRzIjoxNTcyNjQyNjEwZjQ4NjY4Y1BkS05HWn1Jb0g5bTU5NFMOM0RkU0xBiwi
5rfAKnTQQU92AUUU07I2iKoBL2tipBcNCC5zLH5j_WUyjlN15oi_lLHym
fPdih8_Jibjfb5J15U1ifakjQ0rHX04tPal9PvcjwnyZHFcKn-So
Y3wsARn-gWxpzbsPhiKQP70d2eG0CYQMA6rTLs1T7GgdQheelhVFW29i
27NcvqtKJmiAG6Swrq4uUgCY3zRotROkJl3qo86t2DXklV-eES4-2dCxf
cWFkzBAR6oC4Qp7HnY_5UT6IWkRjt3efwYprWcYouOVjtRan3kEtWkaWr
G0J4bPVnTI5St9hJYvvh7FE8JirIg
```

```
{
  "resources": [
```

```

    "dolphin-metadata"
  },
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.foo",
      "nonce": "VJLO6A4CAYLBXHTR0KRO"
    }
  },
  "client": {
    "display": {
      "name": "My Client Display Name",
      "uri": "https://example.net/client"
    },
    "proof": "oauthpop",
    "key": {
      "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "kid": "xyz-1",
        "alg": "RS256",
        "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_J
tffXyaSx8xYJCCNaOKNjN_Oz0YhdHbXTeW05AoyspDWJbN5w_7bdWDxgpD-
y6jnDlu9YhBOCWObNPFvpkTM8LC7SdXGRKx2k8Me2r_GssYlyRpgvpBlY5-
ejCywKRBfctRcnhTTGNztbbDBUyDSWmFMVCHe5mXT4cL0BwrZC6S-uu-LAx
06aKwQOPwYOGos1K8WPmlyGdkaAluF_FpS6LS63WYPHi_Ap2B7_8Wbw4ttz
bMS_doJvuDagW8A1Ip3fXFAHtRACk7rdI4_Xln66hJxFekpdfWdiPQddQ6
Y1cK2U3obvUg7w"
      }
    }
  }
}

```

[[Editor's note: This is a stale draft from the OAuth working group, but it does at least provide some basic functionality for protecting HTTP messages with a signature. This work is likely to be subsumed by the general-purpose HTTP message signature mechanism in Section 8.5.]]

9. Discovery

By design, the protocol minimizes the need for any pre-flight discovery. To begin a request, the RC only needs to know the endpoint of the AS and which keys it will use to sign the request. Everything else can be negotiated dynamically in the course of the protocol.

However, the AS can have limits on its allowed functionality. If the RC wants to optimize its calls to the AS before making a request, it MAY send an HTTP OPTIONS request to the grant request endpoint to retrieve the server's discovery information. The AS MUST respond with a JSON document containing the following information:

`grant_request_endpoint` REQUIRED. The full URL of the AS's grant request endpoint. This MUST match the URL the RC used to make the discovery request.

`capabilities` OPTIONAL. A list of the AS's capabilities. The values of this result MAY be used by the RC in the capabilities section (Section 2.6) of the request.

`interaction_methods` OPTIONAL. A list of the AS's interaction methods. The values of this list correspond to the possible fields in the interaction section (Section 2.5) of the request.

`key_proofs` OPTIONAL. A list of the AS's supported key proofing mechanisms. The values of this list correspond to possible values of the "proof" field of the key section (Section 2.3.2) of the request.

`sub_ids` OPTIONAL. A list of the AS's supported identifiers. The values of this list correspond to possible values of the subject identifier section (Section 2.2) of the request.

`assertions` OPTIONAL. A list of the AS's supported assertion formats. The values of this list correspond to possible values of the subject assertion section (Section 2.2) of the request.

The information returned from this method is for optimization purposes only. The AS MAY deny any request, or any portion of a request, even if it lists a capability as supported. For example, a given RC can be registered with the "mtls" key proofing mechanism, but the AS also returns other proofing methods, then the AS will deny a request from that RC using a different proofing mechanism.

10. Resource Servers

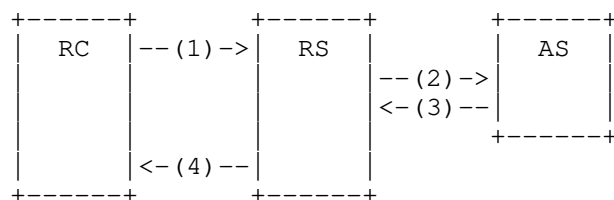
In some deployments, a resource server will need to be able to call the AS for a number of functions.

[[Editor's note: This section is for discussion of possible advanced functionality. It seems like it should be a separate document or set of documents, and it's not even close to being well-baked. This also adds additional endpoints to the AS, as this is separate from the token request process, and therefore would require RS-facing

discovery or configuration information to make it work. Also-also, it does presume the RS can sign requests in the same way that a client does, but hopefully we can be more consistent with this than RFC7662 was able to do.]]

10.1. Introspecting a Token

When the RS receives an access token, it can call the introspection endpoint at the AS to get token information. [[Editor's note: this isn't super different from the token management URIs, but the RS has no way to get that URI, and it's bound to the RS's keys instead of the RC's or token's keys.]]



1. The RC calls the RS with its access token.
2. The RS introspects the access token value at the AS. The RS signs the request with its own key (not the RC's key or the token's key).
3. The AS validates the token value and the RC's request and returns the introspection response for the token.
4. The RS fulfills the request from the RC.

The RS signs the request with its own key and sends the access token as the body of the request.

```
POST /introspect HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...
```

```
{
  "access_token": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
}
```

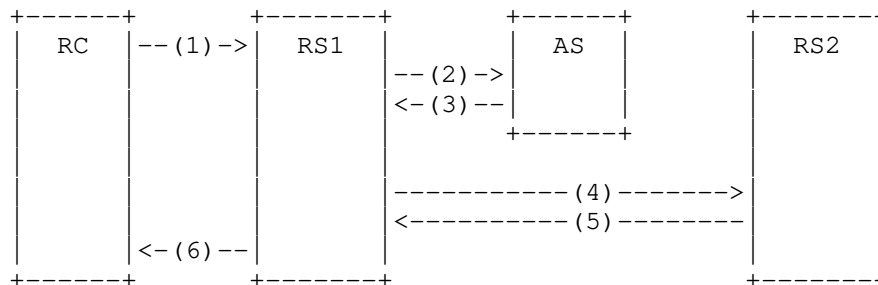
The AS responds with a data structure describing the token's current state and any information the RS would need to validate the token's presentation, such as its intended proofing mechanism and key material.

Content-type: application/json

```
{
  "active": true,
  "resources": [
    "dolphin-metadata", "some other thing"
  ],
  "client": {
    "key": {
      "proof": "httpsig",
      "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "kid": "xyz-1",
        "alg": "RS256",
        "n": "kOB5rR4Jv0GMeL...."
      }
    }
  }
}
```

10.2. Deriving a downstream token

Some architectures require an RS to act as an RC and request a derived access token for a secondary RS. This internal token is issued in the context of the incoming access token.



1. The RC calls RS1 with an access token.
2. RS1 presents that token to the AS to get a derived token for use at RS2. RS1 indicates that it has no ability to interact with the RO. RS1 signs its request with its own key, not the token's key or the RC's key.
3. The AS returns a derived token to RS1 for use at RS2.
4. RS1 calls RS2 with the token from (3).

5. RS2 fulfills the call from RS1.

6. RS1 fulfills the call from RC.

If the RS needs to derive a token from one presented to it, it can request one from the AS by making a token request as described in Section 2 and presenting the existing access token's value in the "existing_access_token" field.

The RS MUST identify itself with its own key and sign the request.

[[Editor's note: this is similar to Section 2.7 but based on the access token and not the grant. We might be able to re-use that function: the fact that the keys presented are not the ones used for the access token should indicate that it's a different party and a different kind of request, but there might be some subtle security issues there.]]

```
POST /tx HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...
```

```
{
  "resources": [
    {
      "actions": [
        "read",
        "write",
        "dolphin"
      ],
      "locations": [
        "https://server.example.net/",
        "https://resource.local/other"
      ],
      "datatypes": [
        "metadata",
        "images"
      ]
    },
    "dolphin-metadata"
  ],
  "client": "7C7C4AZ9KHRS6X63AJAO",
  "existing_access_token": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0"
}
```

The AS responds with a token as described in Section 3.

10.3. Registering a Resource Handle

If the RS needs to, it can post a set of resources as described in Section 2.1.1 to the AS's resource registration endpoint.

The RS MUST identify itself with its own key and sign the request.

```
POST /resource HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...
```

```
{
  "resources": [
    {
      "actions": [
        "read",
        "write",
        "dolphin"
      ],
      "locations": [
        "https://server.example.net/",
        "https://resource.local/other"
      ],
      "datatypes": [
        "metadata",
        "images"
      ]
    },
    "dolphin-metadata"
  ],
  "client": "7C7C4AZ9KHRS6X63AJAO"
}
```

The AS responds with a handle appropriate to represent the resources list that the RS presented.

```
Content-type: application/json
```

```
{
  "resource_handle": "FWWIKYBQ6U56NL1"
}
```

The RS MAY make this handle available as part of a response (Section 10.4) or as documentation to developers.

[[Editor's note: It's not an exact match here because the "resource_handle" returned now represents a collection of objects instead of a single one. Perhaps we should let this return a list of strings instead? Or use a different syntax than the resource request? Also, this borrows heavily from UMA 2's "distributed authorization" model and, like UMA, might be better suited to an extension than the core protocol.]]

10.4. Requesting a Resources With Insufficient Access

If the RC calls an RS without an access token, or with an invalid access token, the RS MAY respond to the RC with an authentication header indicating that GNAP. The address of the GNAP endpoint MUST be sent in the "as_uri" parameter. The RS MAY additionally return a resource reference that the RC MAY use in its resource request (Section 2.1). This resource reference handle SHOULD be sufficient for at least the action the RC was attempting to take at the RS. The RS MAY use the dynamic resource handle request (Section 10.3) to register a new resource handle, or use a handle that has been pre-configured to represent what the AS is protecting. The content of this handle is opaque to the RS and the RC.

WWW-Authenticate: GNAP as_uri=http://server.example/tx,resource=FWWIKYBQ6U56NL1

The RC then makes a call to the "as_uri" as described in Section 2, with the value of "resource" as one of the members of a "resources" array Section 2.1.1. The RC MAY request additional resources and other information, and MAY request multiple access tokens.

[[Editor's note: this borrows heavily from UMA 2's "distributed authorization" model and, like UMA, might be better suited to an extension than the core protocol.]]

11. Acknowledgements

The author would like to thank the feedback of the following individuals for their reviews, implementations, and contributions: Aaron Parecki, Annabelle Backman, Dick Hardt, Dmitri Zagidulin, Dmitry Barinov, Fabien Imbault, Francis Pouatcha, George Fletcher, Haardik Haardik, Hamid Massaoud, Jacky Yuan, Joseph Heenan, Kathleen Moriarty, Mike Jones, Mike Varley, Nat Sakimura, Takahiko Kawasaki, Takahiro Tsuchiya.

In particular, the author would like to thank Aaron Parecki and Mike Jones for insights into how to integrate identity and authentication systems into the core protocol, and to Dick Hardt for the use cases, diagrams, and insights provided in the XAuth proposal that have been incorporated here. The author would like to especially thank Mike Varley and the team at SecureKey for feedback and development of early versions of the XYZ protocol that fed into this standards work.

12. IANA Considerations

[[TBD: There are a lot of items in the document that are expandable through the use of value registries.]]

13. Security Considerations

[[TBD: There are a lot of security considerations to add.]]

All requests have to be over TLS or equivalent as per [BCP195]. Many handles act as shared secrets, though they can be combined with a requirement to provide proof of a key as well.

14. Privacy Considerations

[[TBD: There are a lot of privacy considerations to add.]]

Handles are passed between parties and therefore should not contain any private data.

When user information is passed to the RC, the AS needs to make sure that it has the permission to do so.

15. Normative References

[BCP195] "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", 2015, <<http://www.rfc-editor.org/info/bcp195>>.

[I-D.ietf-httpbis-message-signatures]
Backman, A., Richer, J., and M. Sporny, "Signing HTTP Messages", Work in Progress, Internet-Draft, draft-ietf-httpbis-message-signatures-00, 10 April 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-message-signatures-00.txt>>.

[I-D.ietf-oauth-dpop]
Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstration of Proof-of-Possession at the Application Layer (DPoP)", Work

in Progress, Internet-Draft, draft-ietf-oauth-dpop-01, 1 May 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-oauth-dpop-01.txt>>.

- [I-D.ietf-oauth-signed-http-request] Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", Work in Progress, Internet-Draft, draft-ietf-oauth-signed-http-request-03, 8 August 2016, <<http://www.ietf.org/internet-drafts/draft-ietf-oauth-signed-http-request-03.txt>>.
- [I-D.ietf-secevent-subject-identifiers] Backman, A. and M. Scurtescu, "Subject Identifiers for Security Event Tokens", Work in Progress, Internet-Draft, draft-ietf-secevent-subject-identifiers-06, 4 September 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-secevent-subject-identifiers-06.txt>>.
- [OIDC] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.
- [OIDC4IA] Lodderstedt, T. and D. Fett, "OpenID Connect for Identity Assurance 1.0", October 2019, <https://openid.net/specs/openid-connect-4-identity-assurance-1_0.html>.
- [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>>.
- [RFC3230] Mogul, J. and A. Van Hoff, "Instance Digests in HTTP", RFC 3230, DOI 10.17487/RFC3230, January 2002, <<https://www.rfc-editor.org/info/rfc3230>>.
- [RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <<https://www.rfc-editor.org/info/rfc5646>>.
- [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>>.

- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.
- [RFC7797] Jones, M., "JSON Web Signature (JWS) Unencoded Payload Option", RFC 7797, DOI 10.17487/RFC7797, February 2016, <<https://www.rfc-editor.org/info/rfc7797>>.
- [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>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8693] Jones, M., Nadalin, A., Campbell, B., Ed., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", RFC 8693, DOI 10.17487/RFC8693, January 2020, <<https://www.rfc-editor.org/info/rfc8693>>.
- [RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <<https://www.rfc-editor.org/info/rfc8705>>.

Appendix A. Document History

- * -14
 - Editorial clarification from design team meetings.
 - Added "at_hash" to both JWS methods for use with an access token.
 - Allow attached-JWS method to defer to detached-JWS method for presentation on a non-body request.
- * -13
 - Clarified that "subject" request and response aren't for identity claims, just identifiers.
 - Reworked continuation definition in terms of endpoint actions.

- Defined concrete methods for updating an ongoing grant request using PATCH.
- Defined method for reading current status of grant request using GET.
- Expanded editorial comments and strawman examples for alternatives.

* -12

- Collapsed "key" and "display" fields into "client" field.
- Changed continuation to use optional access token.
- Defined flags for special behavior of tokens.
- Defined "key": true to mean access token is bound to client's key.
- Defined "key": false to indicate an access token.
- Added "Elements" section to list and discuss non-role parts of the protocol.

* -11

- Updated based on Design Team feedback and reviews.
- Removed oidc_ prefix from several values and used RFC8693 assertion types.
- Changed "client" to "RC" throughout.
- Changed "user" to "RQ" or "RO" as appropriate.
- Added expansions for request and response section introductions.
- Added refresh examples.
- Added diagrams to RS examples.
- Added ui_locales hint to interaction block.
- Added section on JSON polymorphism.

- Added numerous editorial notes to describe why elements are in place.
 - Added discussion on composition of roles.
 - Added requirements for signature methods to cover all aspects of request and mechanisms to do so.
- * -10
- Switched to xml2rfc v3 and markdown source.
 - Updated based on Design Team feedback and reviews.
 - Added acknowledgements list.
 - Added sequence diagrams and explanations.
 - Collapsed "short_redirect" into regular redirect request.
 - Separated pass-by-reference into subsections.
 - Collapsed "callback" and "pushback" into a single mode-switched method.
 - Add OIDC Claims request object example.
- * -09
- Major document refactoring based on request and response capabilities.
 - Changed from "claims" language to "subject identifier" language.
 - Added "pushback" interaction capability.
 - Removed DIDCOMM interaction (better left to extensions).
 - Excised "transaction" language in favor of "Grant" where appropriate.
 - Added token management URLs.
 - Added separate continuation URL to use continuation handle with.
 - Added RS-focused functionality section.

- Added notion of extending a grant request based on a previous grant.
- Simplified returned handle structures.
- * -08
 - Added attached JWS signature method.
 - Added discovery methods.
- * -07
 - Marked sections as being controlled by a future registry TBD.
- * -06
 - Added multiple resource requests and multiple access token response.
- * -05
 - Added "claims" request and response for identity support.
 - Added "capabilities" request for inline discovery support.
- * -04
 - Added crypto agility for callback return hash.
 - Changed "interaction_handle" to "interaction_ref".
- * -03
 - Removed "state" in favor of "nonce".
 - Created signed return parameter for front channel return.
 - Changed "client" section to "display" section, as well as associated handle.
 - Changed "key" to "keys".
 - Separated key proofing from key presentation.
 - Separated interaction methods into booleans instead of "type" field.

- * -02
 - Minor editorial cleanups.
- * -01
 - Made JSON multimodal for handle requests.
 - Major updates to normative language and references throughout document.
 - Allowed interaction to split between how the user gets to the AS and how the user gets back.
- * -00
 - Initial submission.

Appendix B. Component Data Models

While different implementations of this protocol will have different realizations of all the components and artifacts enumerated here, the nature of the protocol implies some common structures and elements for certain components. This appendix seeks to enumerate those common elements.

TBD: Client has keys, allowed requested resources, identifier(s), allowed requested subjects, allowed

TBD: AS has "grant endpoint", interaction endpoints, store of trusted client keys, policies

TBD: Token has RO, user, client, resource list, RS list,

Appendix C. Example Protocol Flows

The protocol defined in this specification provides a number of features that can be combined to solve many different kinds of authentication scenarios. This section seeks to show examples of how the protocol would be applied for different situations.

Some longer fields, particularly cryptographic information, have been truncated for display purposes in these examples.

C.1. Redirect-Based User Interaction

In this scenario, the user is the RO and has access to a web browser, and the client can take front-channel callbacks on the same device as the user. This combination is analogous to the OAuth 2 Authorization Code grant type.

The client initiates the request to the AS. Here the client identifies itself using its public key.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...
```

```
{
  "resources": [
    {
      "actions": [
        "read",
        "write",
        "dolphin"
      ],
      "locations": [
        "https://server.example.net/",
        "https://resource.local/other"
      ],
      "datatypes": [
        "metadata",
        "images"
      ]
    }
  ],
  "client": {
    "key": {
      "proof": "jwsd",
      "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "kid": "xyz-1",
        "alg": "RS256",
        "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8xY..."
      }
    }
  },
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.example.net/return/123455",
      "nonce": "LKLTi25DK82FX4T4QFZC"
    }
  }
}
```

The AS processes the request and determines that the RO needs to interact. The AS returns the following response giving the client the information it needs to connect. The AS has also indicated to the client that it can use the given instance identifier to identify itself in future requests (Section 2.3.1).

Content-type: application/json

```
{
  "interact": {
    "redirect": "https://server.example.com/interact/4CF492MLVMSW9MKMXKHQ",
    "callback": "MBDOFXG4Y5CVJCX821LH"
  }
  "continue": {
    "access_token": {
      "value": "80UPRY5NM33OMUKMKSKU",
      "key": true
    },
    "uri": "https://server.example.com/continue"
  },
  "instance_id": "7C7C4AZ9KHRS6X63AJAO"
}
```

The client saves the response and redirects the user to the `interaction_url` by sending the following HTTP message to the user's browser.

```
HTTP 302 Found
Location: https://server.example.com/interact/4CF492MLVMSW9MKMXKHQ
```

The user's browser fetches the AS's interaction URL. The user logs in, is identified as the RO for the resource being requested, and approves the request. Since the AS has a callback parameter, the AS generates the interaction reference, calculates the hash, and redirects the user back to the client with these additional values added as query parameters.

```
HTTP 302 Found
Location: https://client.example.net/return/123455
?hash=p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jzQ1R2HZT8BOWYHcLmObM7XHPAdJzTZMtKBSaraJ64A
&interact_ref=4IFWWIKYBC2PQ6U56NL1
```

The client receives this request from the user's browser. The client ensures that this is the same user that was sent out by validating session information and retrieves the stored pending request. The client uses the values in this to validate the hash parameter. The client then calls the continuation URL and presents the handle and interaction reference in the request body. The client signs the request as above.

```
POST /continue HTTP/1.1
Host: server.example.com
Content-type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Detached-JWS: ejy0...
```

```
{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

The AS retrieves the pending request based on the handle and issues a bearer access token and returns this to the client.

Content-type: application/json

```
{
  "access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "key": false,
    "manage": "https://server.example.com/token/PRY5NM33OM4TB8N6BW7OZB8CDFONP
219RP1L",
    "resources": [{
      "actions": [
        "read",
        "write",
        "dolphin"
      ],
      "locations": [
        "https://server.example.net/",
        "https://resource.local/other"
      ],
      "datatypes": [
        "metadata",
        "images"
      ]
    }]
  },
  "continue": {
    "access_token": {
      "value": "80UPRY5NM33OMUKMKSKU",
      "key": true
    },
    "uri": "https://server.example.com/continue"
  }
}
```

C.2. Secondary Device Interaction

In this scenario, the user does not have access to a web browser on the device and must use a secondary device to interact with the AS. The client can display a user code or a printable QR code. The client prefers a short URL if one is available, with a maximum of 255 characters in length. The is not able to accept callbacks from the AS and needs to poll for updates while waiting for the user to authorize the request.

The client initiates the request to the AS.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...
```

```
{
  "resources": [
    "dolphin-metadata", "some other thing"
  ],
  "client": "7C7C4AZ9KHRS6X63AJAO",
  "interact": {
    "redirect": 255,
    "user_code": true
  }
}
```

The AS processes this and determines that the RO needs to interact. The AS supports both long and short redirect URIs for interaction, so it includes both. Since there is no "callback" the AS does not include a nonce, but does include a "wait" parameter on the continuation section because it expects the client to poll for results.

```
Content-type: application/json
```

```
{
  "interact": {
    "redirect": "https://srv.ex/MXKHQ",
    "user_code": {
      "code": "A1BC-3DFF",
      "url": "https://srv.ex/device"
    }
  },
  "continue": {
    "uri": "https://server.example.com/continue/80UPRY5NM33OMUKMKSKU",
    "wait": 60
  }
}
```

The client saves the response and displays the user code visually on its screen along with the static device URL. The client also displays the short interaction URL as a QR code to be scanned.

If the user scans the code, they are taken to the interaction endpoint and the AS looks up the current pending request based on the incoming URL. If the user instead goes to the static page and enters the code manually, the AS looks up the current pending request based on the value of the user code. In both cases, the user logs in, is

identified as the RO for the resource being requested, and approves the request. Once the request has been approved, the AS displays to the user a message to return to their device.

Meanwhile, the client periodically polls the AS every 60 seconds at the continuation URL. The client signs the request using the same key and method that it did in the first request.

```
POST /continue/80UPRY5NM33OMUKMKSKU HTTP/1.1
Host: server.example.com
Detached-JWS: ejy0...
```

The AS retrieves the pending request based on the handle and determines that it has not yet been authorized. The AS indicates to the client that no access token has yet been issued but it can continue to call after another 60 second timeout.

Content-type: application/json

```
{
  "continue": {
    "uri": "https://server.example.com/continue/BI9QNW6V9W3XFJK4R02D",
    "wait": 60
  }
}
```

Note that the continuation URL has been rotated since it was used by the client to make this call. The client polls the continuation URL after a 60 second timeout using the new handle.

```
POST /continue/BI9QNW6V9W3XFJK4R02D HTTP/1.1
Host: server.example.com
Authorization: GNAP
Detached-JWS: ejy0...
```

The AS retrieves the pending request based on the URL, determines that it has been approved, and issues an access token.

Content-type: application/json

```
{
  "access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "key": false,
    "manage": "https://server.example.com/token/PRY5NM33OM4TB8N6BW7OZB8CDFONP
219RP1L",
    "resources": [
      "dolphin-metadata", "some other thing"
    ]
  }
}
```

Appendix D. No User Involvement

In this scenario, the client is requesting access on its own behalf, with no user to interact with.

The client creates a request to the AS, identifying itself with its public key and using MTLS to make the request.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-type: application/json
```

```
{
  "resources": [
    "backend service", "nightly-routine-3"
  ],
  "client": {
    "key": {
      "proof": "mtls",
      "cert#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
    }
  }
}
```

The AS processes this and determines that the client can ask for the requested resources and issues an access token.

Content-type: application/json

```
{
  "access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "key": true,
    "manage": "https://server.example.com/token",
    "resources": [
      "backend service", "nightly-routine-3"
    ]
  }
}
```

D.1. Asynchronous Authorization

In this scenario, the client is requesting on behalf of a specific RO, but has no way to interact with the user. The AS can asynchronously reach out to the RO for approval in this scenario.

The client starts the request at the AS by requesting a set of resources. The client also identifies a particular user.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...
```

```
{
  "resources": [
    {
      "type": "photo-api",
      "actions": [
        "read",
        "write",
        "dolphin"
      ],
      "locations": [
        "https://server.example.net/",
        "https://resource.local/other"
      ],
      "datatypes": [
        "metadata",
        "images"
      ]
    },
    "read", "dolphin-metadata",
    {
      "type": "financial-transaction",
      "actions": [
        "withdraw"
      ],
      "identifier": "account-14-32-32-3",
      "currency": "USD"
    },
    "some other thing"
  ],
  "client": "7C7C4AZ9KHRS6X63AJAO",
  "user": {
    "sub_ids": [ {
      "subject_type": "email",
      "email": "user@example.com"
    } ]
  }
}
```

The AS processes this and determines that the RO needs to interact. The AS determines that it can reach the identified user asynchronously and that the identified user does have the ability to approve this request. The AS indicates to the client that it can poll for continuation.

Content-type: application/json

```
{
  "continue": {
    "access_token": {
      "value": "80UPRY5NM33OMUKMKSKU",
      "key": true
    },
    "uri": "https://server.example.com/continue",
    "wait": 60
  }
}
```

The AS reaches out to the RO and prompts them for consent. In this example, the AS has an application that it can push notifications in to for the specified account.

Meanwhile, the client periodically polls the AS every 60 seconds at the continuation URL.

```
POST /continue HTTP/1.1
Host: server.example.com
Authorization: G NAP 80UPRY5NM33OMUKMKSKU
Detached-JWS: ejy0...
```

The AS retrieves the pending request based on the handle and determines that it has not yet been authorized. The AS indicates to the client that no access token has yet been issued but it can continue to call after another 60 second timeout.

Content-type: application/json

```
{
  "continue": {
    "access_token": {
      "value": "BI9QNW6V9W3XFJK4R02D",
      "key": true
    },
    "uri": "https://server.example.com/continue",
    "wait": 60
  }
}
```

Note that the continuation handle has been rotated since it was used by the client to make this call. The client polls the continuation URL after a 60 second timeout using the new handle.

```
POST /continue HTTP/1.1
Host: server.example.com
Authorization: GNAP BI9QNW6V9W3XFJK4R02D
Detached-JWS: ejy0...
```

The AS retrieves the pending request based on the handle and determines that it has been approved and it issues an access token.

Content-type: application/json

```
{
  "access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "key": false,
    "manage": "https://server.example.com/token/PRY5NM33OM4TB8N6BW7OZB8CDFONP
219RP1L",
    "resources": [
      "dolphin-metadata", "some other thing"
    ]
  }
}
```

D.2. Applying OAuth 2 Scopes and Client IDs

While the GNAP protocol is not designed to be directly compatible with OAuth 2 [RFC6749], considerations have been made to enable the use of OAuth 2 concepts and constructs more smoothly within the GNAP protocol.

In this scenario, the client developer has a "client_id" and set of "scope" values from their OAuth 2 system and wants to apply them to the new protocol. Traditionally, the OAuth 2 client developer would put their "client_id" and "scope" values as parameters into a redirect request to the authorization endpoint.

```
HTTP 302 Found
Location: https://server.example.com/authorize
?client_id=7C7C4AZ9KHRS6X63AJAO
&scope=read%20write%20dolphin
&redirect_uri=https://client.example.net/return
&response_type=code
&state=123455
```

Now the developer wants to make an analogous request to the AS using the new protocol. To do so, the client makes an HTTP POST and places the OAuth 2 values in the appropriate places.

```

POST /tx HTTP/1.1
Host: server.example.com
Content-type: application/json
Detached-JWS: ejy0...

```

```

{
  "resources": [
    "read", "write", "dolphin"
  ],
  "client": "7C7C4AZ9KHRS6X63AJAO",
  "interact": {
    "redirect": true,
    "callback": {
      "method": "redirect",
      "uri": "https://client.example.net/return?state=123455",
      "nonce": "LKLTi25DK82FX4T4QFZC"
    }
  }
}

```

The `client_id` can be used to identify the client's keys that it uses for authentication, the scopes represent resources that the client is requesting, and the `"redirect_uri"` and `"state"` value are pre-combined into a `"callback"` URI that can be unique per request. The client additionally creates a nonce to protect the callback, separate from the state parameter that it has added to its return URL.

From here, the protocol continues as above.

Appendix E. JSON Structures and Polymorphism

The GNAp protocol makes use of polymorphism within the JSON [RFC8259] structures used for the protocol. Each portion of this protocol is defined in terms of the JSON data type that its values can take, whether it's a string, object, array, boolean, or number. For some fields, different data types offer different descriptive capabilities and are used in different situations for the same field. Each data type provides a different syntax to express the same underlying semantic protocol element, which allows for optimization and simplification in many common cases.

Even though JSON is often used to describe strongly typed structures, JSON on its own is naturally polymorphic. In JSON, the named members of an object have no type associated with them, and any data type can be used as the value for any member. In practice, each member has a semantic type that needs to make sense to the parties creating and consuming the object. Within this protocol, each object member is defined in terms of its semantic content, and this semantic content

might have expressions in different concrete data types for different specific purposes. Since each object member has exactly one value in JSON, each data type for an object member field is naturally mutually exclusive with other data types within a single JSON object.

For example, a resource request for a single access token is composed of an array of resource request descriptions while a request for multiple access tokens is composed of an object whose member values are all arrays. Both of these represent requests for access, but the difference in syntax allows the RC and AS to differentiate between the two request types in the same request.

Another form of polymorphism in JSON comes from the fact that the values within JSON arrays need not all be of the same JSON data type. However, within this protocol, each element within the array needs to be of the same kind of semantic element for the collection to make sense, even when the data types are different from each other.

For example, each aspect of a resource request can be described using an object with multiple dimensional components, or the aspect can be requested using a string. In both cases, the resource request is being described in a way that the AS needs to interpret, but with different levels of specificity and complexity for the RC to deal with. An API designer can provide a set of common access scopes as simple strings but still allow RC developers to specify custom access when needed for more complex APIs.

Extensions to this specification can use different data types for defined fields, but each extension needs to not only declare what the data type means, but also provide justification for the data type representing the same basic kind of thing it extends. For example, an extension declaring an "array" representation for a field would need to explain how the array represents something akin to the non-array element that it is replacing.

Author's Address

Justin Richer (editor)
Bespoke Engineering

Email: ietf@justin.richer.org
URI: <https://bspk.io/>