Workgroup: Network Working Group
Internet-Draft: draft-hardt-xauth-protocol-02
Published: 8 February 2020
Intended Status: Standards Track
Expires: 11 August 2020
Authors: D. Hardt, Ed.
SignIn.Org

The XAuth Protocol

Abstract

Client software often desires resources or identity claims that are independent of the client. This protocol allows a user and/or resource owner to delegate resource authorization and/or release of identity claims to a server. Client software can then request access to resources and/or identity claims by calling the server. The server acquires consent and authorization from the user and/or resource owner if required, and then returns to the client software the authorization and identity claims that were approved. This protocol can be extended to support alternative authorizations, claims, interactions, and client authentication mechanisms.

Note to Readers

Source for this draft and an issue tracker can be found at https://github.com/dickhardt/hardt-xauth-protocol.

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 11 August 2020.

Table of Contents

1. Introduction

This protocol supports the widely deployed use cases supported by OAuth 2.0 [RFC6749] & [RFC6750], and OpenID Connect [OIDC], an extension of OAuth 2.0, as well as other extensions, and other use cases that are not supported, such as acquiring multiple access tokens at the same time, and updating the request during user interaction. This protocol also addresses many of the security issues in OAuth 2.0 by having parameters securely sent directly between parties, rather than via a browser redirection.

The technology landscape has changed since OAuth 2.0 was initially drafted. More interactions happen on mobile devices than PCs. Modern browsers now directly support asymetric cryptographic functions. Standards have emerged for signing and encrypting tokens with rich payloads (JOSE) that are widely deployed.

Additional use cases are now being served with extensions to OAuth 2.0: OpenID Connect added support for authentication and releasing identity claims; [RFC8252] added support for native apps; [RFC8628] added support for smart devices; and support for [browser_based_apps] is being worked on. There are numerous efforts on adding proof-of-possession to resource access.

This protocol simplifies the overall architectural model, takes advantage of today's technology landscape, provides support for all the widely deployed use cases, and offers numerous extension points.

While this protocol is not backwards compatible with OAuth 2.0, it strives to minimize the migration effort.

This protocol centers around a Grant, a representation of the collection of user identity claims and/or resource authorizations the Client is requesting, and the resulting identity claims and/or resource authorizations granted by the Grant Server.

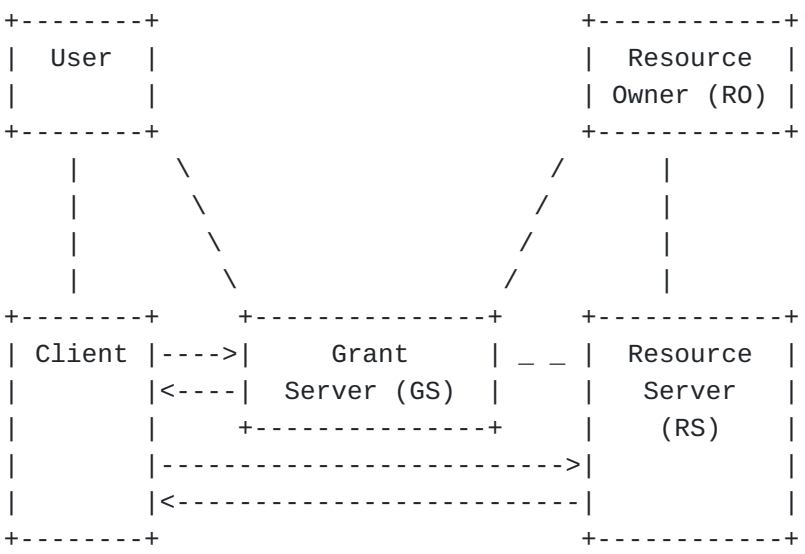
[Editor: suggestions on how to improve this are welcome!]

[Editor: suggestions for other names than XAuth are welcome!]

2. Terminology

2.1. Parties

The parties and their relationships to each other:



* **User** - the person interacting with the Client who has delegated access to identity claims about themselves to the Grant Server (GS), and can authenticate at the GS.

* **Client** - requests a Grant from the GS to access one or more Resource Servers (RSs), and/or identity claims about the User. The Client can create, retrieve, update, and delete a Grant. When a Grant is created, the Client receives from the GS the granted access token(s) for the RS(s), and identity claims about the User. The Client uses an access token to access the RS. There are two types of Clients: Registered Clients and Dynamic Clients. All Clients have a key to authenticate with the Grant Server.

* **Registered Client** - a Client that has registered with the GS and has a Client ID to identify itself, and can prove it possesses a key that is linked to the Client ID. The GS may have different

policies for what different Registered Clients can request. A Registered Client MAY be interacting with a User.

* **Dynamic Client** - a Client that has not been registered with the GS, and each instance will generate it's own key pair so it can prove it is the same instance of the Client on subsequent requests, and to requests of a Resource Server. A single-page application with no server is an example of a Dynamic Client. A Dynamic Client MUST be interacting with a User.

* **Grant Server** (GS) - manages Grants for access to APIs at RSs and release of identity claims about the User. The GS may require explicit consent from the RO or User to provide these to the Client. An GS may support Registered Clients and/or Dynamic Clients. The GS is a combination of the Authorization Server (AS) in OAuth 2.0, and the OpenID Provider (OP) in OpenID Connect.

* **Resource Server** (RS) - has API resources that require an access token from the GS. Owned by the Resource Owner.

* **Resource Owner** (RO) - owns the RS, and has delegated RS access management to the GS. The RO may be the same entity as the User, or may be a different entity that the GS interacts with independently. GS and RO interactions are out of scope of this document.

2.2. Reused Terms

* **access token** - an access token as defined in [RFC6749] Section 1.4.

* **Claim** - a Claim as defined in [OIDC] Section 5. Claims may be issued by the GS, or by other issuers.

* **Client ID** - a GS unique identifier for a Registered Client as defined in [RFC6749] Section 2.2.

* **ID Token** - an ID Token as defined in [OIDC] Section 2.

* **NumericDate** - a NumericDate as defined in [RFC7519] Section 2.

* **authN** - short for authentication.

* **authZ** - short for authorization.

2.3. New Terms

* **GS URI** - the endpoint at the GS the Client calls to create a Grant. The unique identifier for the GS.

*__Grant__ - the user identity claims and/or RS authorizations the GS has granted to the Client.

*__Grant URI__ - the URI that represents the Grant. The Grant URI MUST start with the GS URI.

*__Authorization__ - the access granted by the RO to the Client. Contains an access token.

*__AuthZ URI__ - the URI that represents the Authorization the Client was granted by the RO. The AuthZ URI MUST start with the GS URI. The AuthZ URI is used to refresh, update, and delete an access token.

*__interaction__ - [Editor: what do we really mean by this term?]

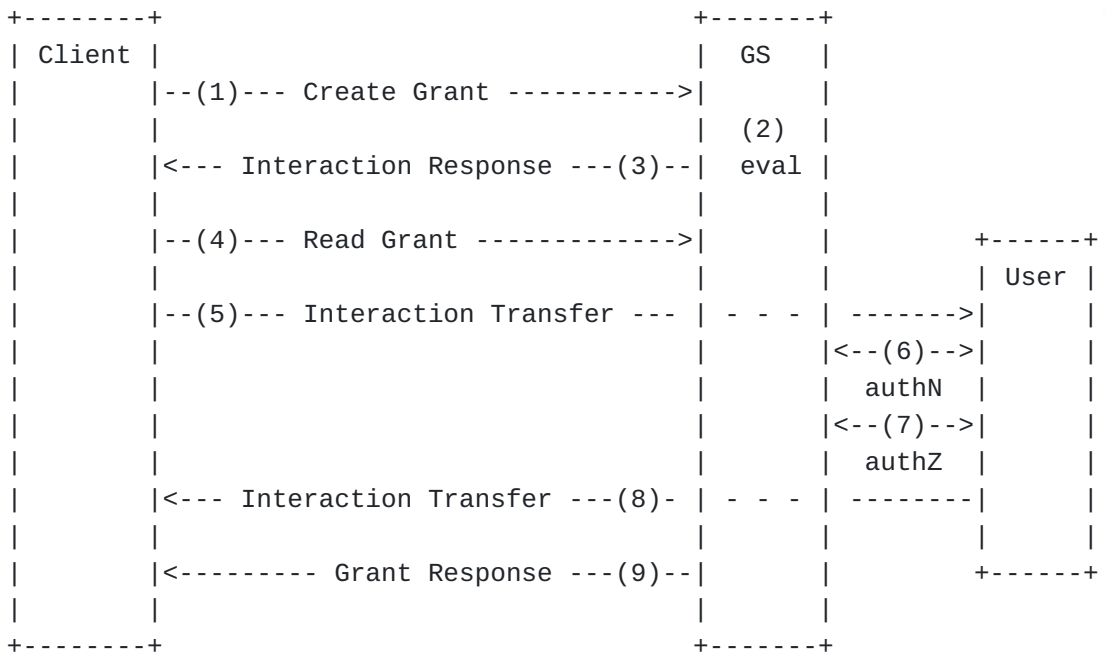
3. Sequences

Before any sequence, the Client needs to be manually or programmatically configured for the GS. See GS Options Section 5.9 for details on acquiring GS metadata.

[Editor: a plethora of sequences are included to illustrate all the different actions. Many of these could potentially be moved to a use case document in the future.]

3.1. Create Grant

The Client requests a Grant from the GS that requires User interaction:



1. **Create Grant** The Client creates a Grant Request (Section 5.1) and sends it with an HTTP POST to the GS GS URI.

2. **Grant Request Evaluation** The GS processes the request to determine if it will send a Interaction Response, Wait Response, or a Grant Response. The GS determines that interaction with the User is required and sends an Interaction Response. (For readability, this step is not described in the following sequences)

3. **Interaction Response** The GS sends an Interaction Response (Section 7.2) containing the Grant URI and an interaction object.

4. **Read Grant** The Client does an HTTP GET of the Grant URI (Section 5.2).

5. **Interaction Transfer** The Client transfers User interaction to the GS.

6. **User Authentication** The GS authenticates the User.

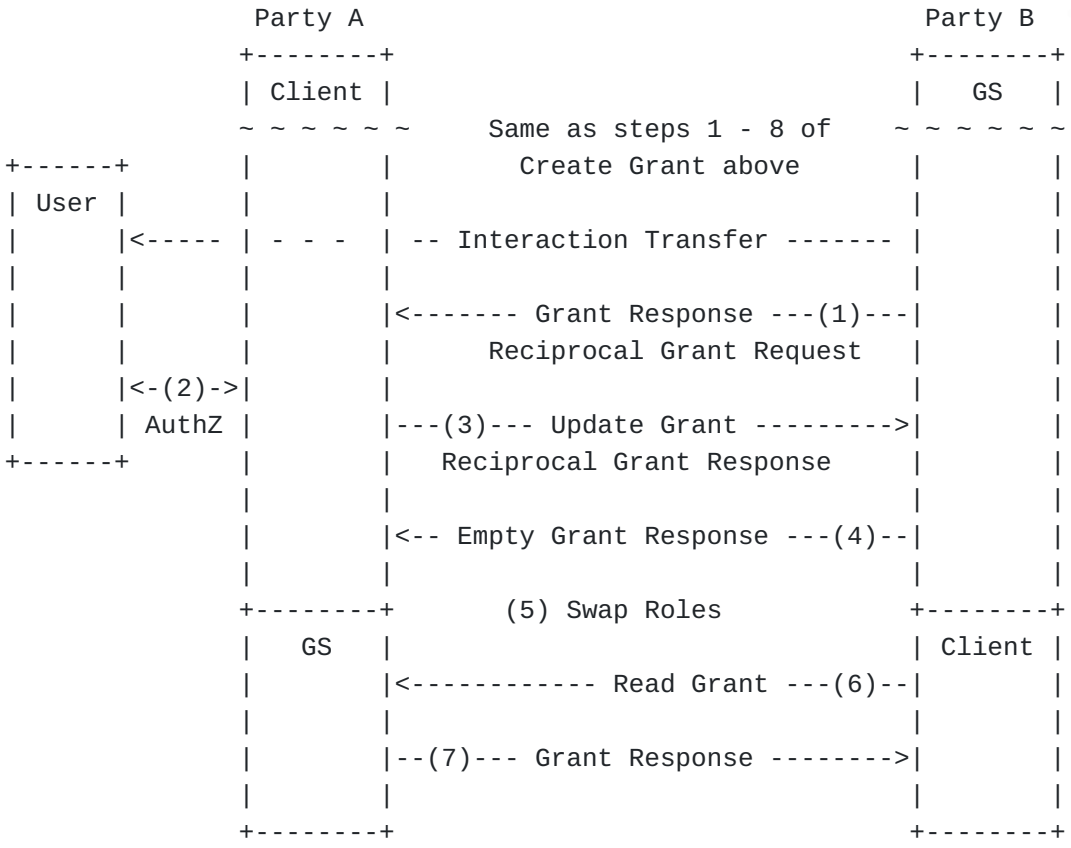
7. **User Authorization** If required, the GS interacts with the User to determine which identity claims and/or authorizations in the Grant Request are to be granted.

8. **Interaction Transfer** The GS transfers User interaction to the Client.

9. **Grant Response** The GS responds with a Grant Response (Section 7.1).

3.2. Reciprocal Grant

Party A and Party B are both a Client and a GS, and each Client would like a Grant for the other GS. Party A starts off being the Client per Create Grant Section 3.1. Party B then includes a Reciprocal Request in its Grant Response. Party A then gets authorization from the User and returns a Grant URI to Party B. Party A and B swap roles, and Party B's Client obtains the Grant from Party A's GS.



1. **Grant Response** Party B responds with a Grant Response including a Reciprocal Object Section 7.4.6 requesting its own Grant.

2. **User Authorization** If required, Party A interacts with the User to determine which identity claims and/or authorizations in the Grant Request are to be granted to Party B.

3. **Update Grant** Party A sends an Update Grant request containing the Grant URI in the Reciprocal object Section 5.5.7.

4. **Grant Response** Party B responds with an Empty Grant Response as there were no other requests in the Update Grant.

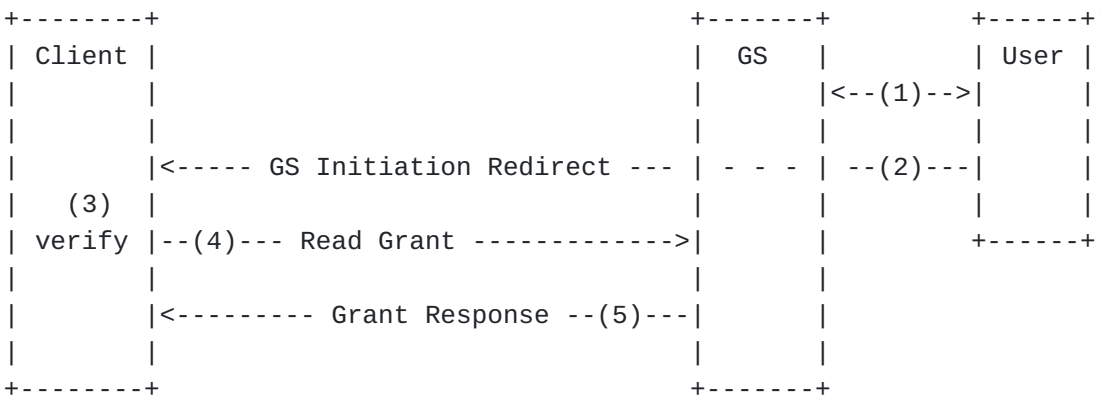
5. **Swap Roles** Party A now acts as a GS, Party B as a Client.

6. **Read Grant** Party B does an HTTP GET of the Grant URI (Section 5.2).

7. **Grant Response** Party A responds with a Grant Response (Section 7.1).

3.3. GS Initiated Grant

The User is at the GS, and wants to interact with a Registered Client. The GS can redirect the User to the Client:



1. **User Interaction** The GS interacts with the User to determine the Client and what identity claims and authorizations to provide. The GS creates a Grant and corresponding Grant URI.

2. **GS Initiated Redirect** The GS redirects the User to the Client's interaction_uri, adding a query parameter with the name "Grant URI" and the value being the URL encoded Grant URI.

3. **Client Verification** The Client verifies the Grant URI is from an GS the Client trusts, and starts with the GS GS URI.

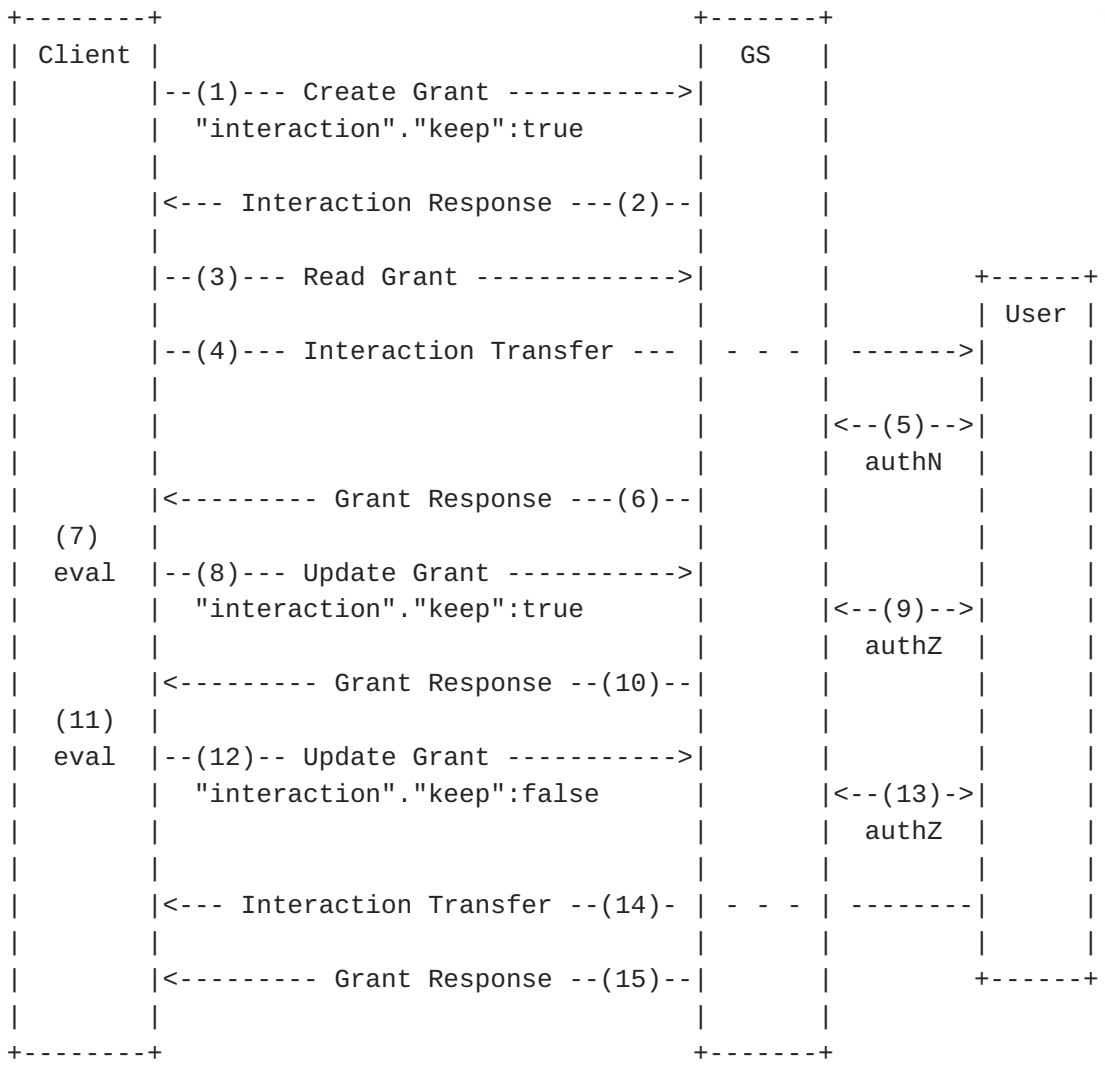
4. **Read Grant** The Client does an HTTP GET of the Grant URI (Section 5.2).

5. **Grant Response** The GS responds with a Grant Response (Section 7.1).

See Section 6 for more details.

3.4. Create and Update

The Client requests an identity claim to determine who the User is. Once the Client learns who the User is, and the Client updates the Grant for additional identity claims which the GS prompts the User for and returns to the Client. Once those are received, the Client updates the Grant with the remaining identity claims required.



1. **Create Grant** The Client creates a Grant Request (Section 5.1) including an identity claim and "interaction"."keep":true, and sends it with an HTTP POST to the GS GS URI.

2. **Interaction Response** The GS sends an Interaction Response (Section 7.2) containing the Grant URI, an interaction object, and "interaction"."keep":true.

3. **Read Grant** The Client does an HTTP GET of the Grant URI (Section 5.2).

4. **Interaction Transfer** The Client transfers User interaction to the GS.

5. **User Authentication** The GS authenticates the User.

6. **Grant Response** The GS responds with a Grant Response (Section 7.1) including the identity claim from User authentication and "interaction"."keep":true.

7. **Grant Evaluation** The Client queries its User database and does not find a User record matching the identity claim.

8. **Update Grant** The Client creates an Update Grant Request (Section 5.3) including the initial identity claims required and "interaction"."keep":true, and sends it with an HTTP PUT to the Grant URI.

9. **User AuthN** The GS interacts with the User to determine which identity claims in the Update Grant Request are to be granted.

10. **Grant Response** The GS responds with a Grant Response (Section 7.1) including the identity claims released by the User and "interaction"."keep":true.

11. **Grant Evaluation** The Client evaluates the identity claims in the Grant Response and determines the remaining User identity claim required.

12. **Update Grant** The Client creates an Update Grant Request (Section 5.3) including the remaining required identity claims and "interaction"."keep":false, and sends it with an HTTP PUT to the Grant URI.

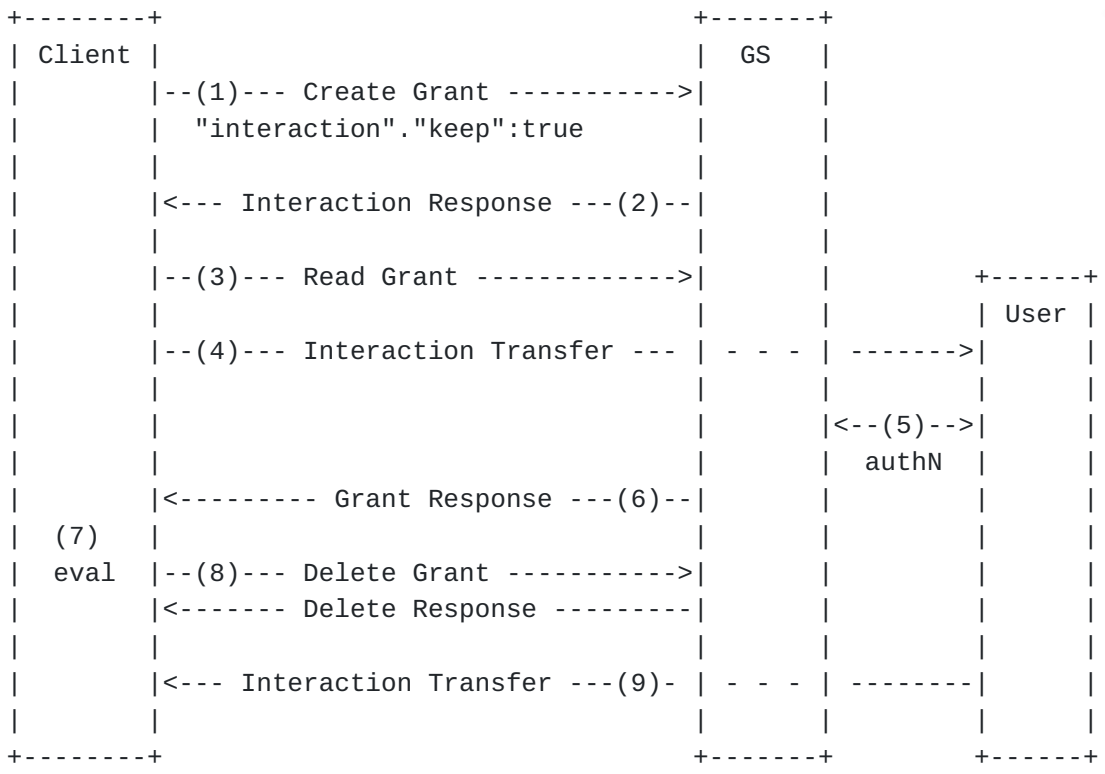
13. **User AuthZ** The GS interacts with the User to determine which identity claims in the Update Grant Request are to be granted.

14. **Interaction Transfer** The GS transfers User interaction to the Client.

15. **Grant Response** The GS responds with a Grant Response (Section 7.1) including the identity claims released by the User.

3.5. Create and Delete

The Client requests an identity claim to determine who the User is. Once the Client learns who the User is, and the Client knows it has all the identity claims and authorizations needed, the Client deletes the Grant which prompts the GS to transfer the interaction back to the Client.



1. **Create Grant** The Client creates a Grant Request (Section 5.1) including an identity claim and "interaction"."keep":true, and sends it with an HTTP POST to the GS GS URI.

2. **Interaction Response** The GS sends an Interaction Response (Section 7.2) containing the Grant URI, an interaction object, and "interaction"."keep":true.

3. **Read Grant** The Client does an HTTP GET of the Grant URI (Section 5.2).

4. **Interaction Transfer** The Client transfers User interaction to the GS.

5. **User Authentication** The GS authenticates the User.

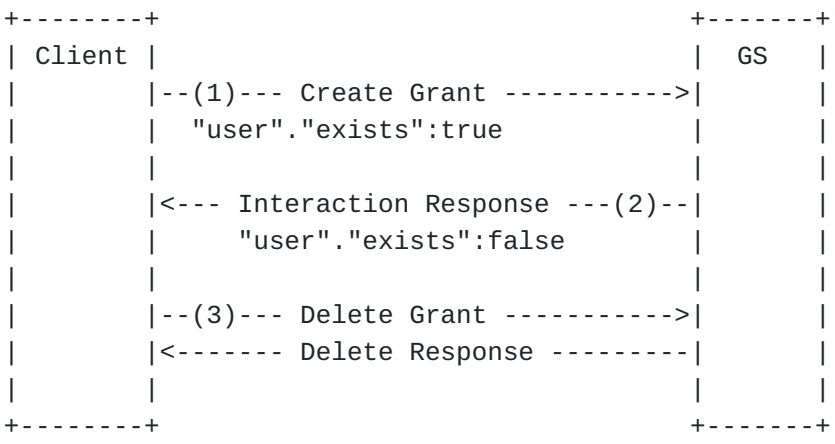
6. **Grant Response** The GS responds with a Grant Response (Section 7.1) including the identity claim from User authentication and "interaction"."keep":true.

7. **Grant Evaluation** The Client queries its User database and finds the User record matching the identity claim, and that no additional claims or authorizations are required.

8. **Delete Grant** The Client no longer needs the Grant and decides to Delete Grant (Section 5.4) by sending an HTTP DELETE to the Grant URI. If the GS responds with success the Grant no longer exists.

3.6. Create, Discover, and Delete

The Client wants to discover if the GS has a User with a given identifier. If not, it will abort the request and not transfer interaction to the GS.



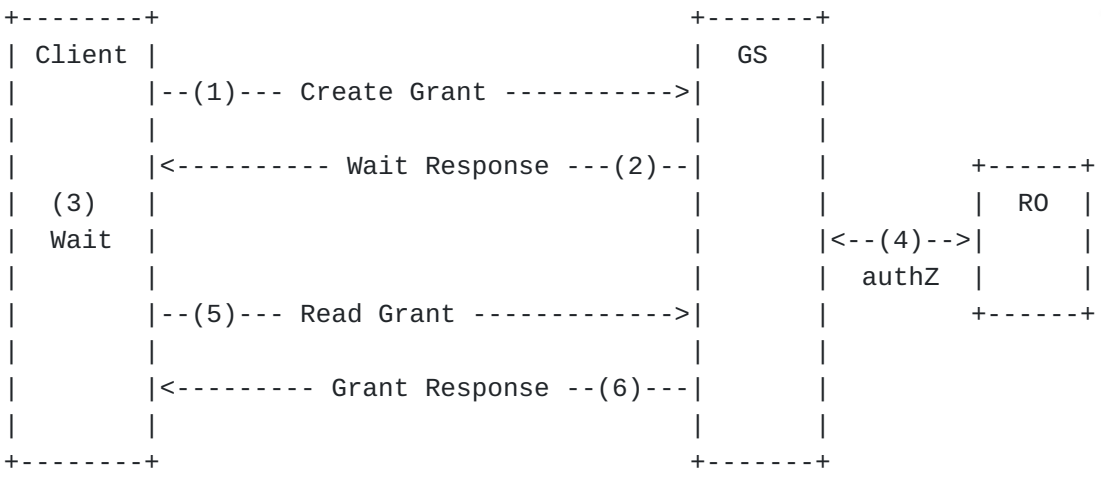
1. **Create Grant** The Client creates a Grant Request (Section 5.1) including an identity claim request, a User identifier, and "user"."exists":true. The Client sends it with an HTTP POST to the GS GS URI.

2. **Interaction Response** The GS sends an Interaction Response (Section 7.2) containing the Grant URI, an interaction object, and "user"."exists":false.

3. **Delete Grant** The Client determines the GS cannot fulfil its Grant Request, and decides to Delete Grant (Section 5.4) by sending an HTTP DELETE to the Grant URI. If the GS responds with success the Grant no longer exists.

3.7. Create and Wait

The Client wants access to resources that require the GS to interact with the RO, which may not happen immediately, so the GS instructs the Client to wait and check back later.



1. **Create Grant** The Client creates a Grant Request (Section 5.1) and sends it with an HTTP POST to the GS GS URI.

2. **Wait Response** The GS sends an Interaction Response (Section 7.3) containing the Grant URI and wait time.

3. **Client Waits** The Client waits the wait time.

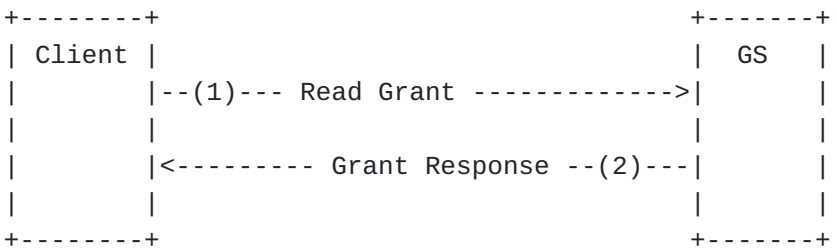
4. **RO AuthZ** The GS interacts with the RO to determine which identity claims in the Grant Request are to be granted.

5. **Read Grant** The Client does an HTTP GET of the Grant URI (Section 5.2).

6. **Grant Response** The GS responds with a Grant Response (Section 7.1).

3.8. Read Grant

The Client wants to acquire fresh identity claims and authorizations in the Grant. No User or RO interaction is required as no new consent or authorization is required.

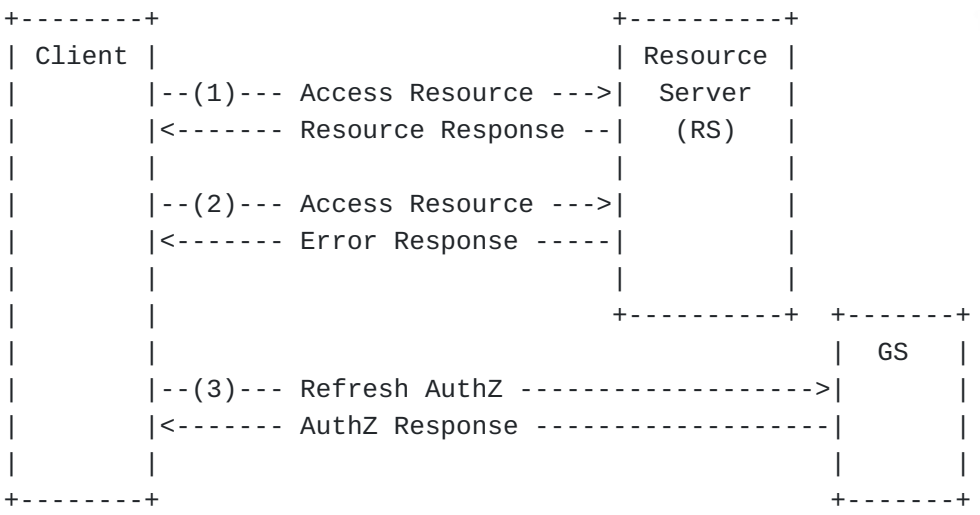


1. **Read Grant** The Client does an HTTP GET of the Grant URI (Section 5.2).

2. **Grant Response** The GS responds with a Grant Response (Section 7.1) containing updated identity claims and authorizations.

3.9. Access Token Refresh

The Client has an access token and uses it to access resources at an RS. The access token expires, and the Client acquires a fresh access token from the GS.



1. **Resource Request** The Client accesses the RS with the access token per Section 8 and receives a response from the RS.

2. **Resource Request** The Client attempts to access the RS, but receives an error indicating the access token has expired.

3. **Refresh AuthZ** If the Client received an AuthZ URI in the Response JSON "authorization" object (Section 7.4.3), the Client can Refresh AuthZ (Section 5.6) with an HTTP GET to the AuthZ URI and receive an Response JSON "authorization" object" (Section 7.4.3) with a fresh access token.

3.10. GS API Table

request	http verb	uri	response
Create Grant	POST	GS URI	interaction, wait, or grant
Read Grant	GET	Grant URI	wait, or grant
Update Grant	PUT	Grant URI	interaction, wait, or grant
Delete Grant	DELETE	Grant URI	success
Refresh AuthZ	GET	AuthZ URI	authorization
Update AuthZ	PUT	AuthZ URI	authorization
Delete AuthZ	DELETE	AuthZ URI	success
GS Options	OPTIONS	GS URI	metadata
Grant Options	OPTIONS	Grant URI	metadata
AuthZ Options	OPTIONS	AuthZ URI	metadata

Table 1

[Editor: is there value in an API for listing a Client's Grants? eg:]

List Grants	GET	GS URI	JSON array of Grant URIs

4. Grant and AuthZ Life Cycle

[Editor: straw man for life cycles.]

Grant life Cycle

The Client MAY create, read, update, and delete Grants. A Grant persists until it has expired, is deleted, or another Grant is created for the same GS, Client, and User tuple.

At any point in time, there can only be one Grant for the GS, Client, and User tuple. When a Client creates a Grant at the same GS for the same User, the GS MUST invalidate a previous Grant for the Client at that GS for that User.

Authorization Life Cycle

Authorizations are OPTIONALLY included in a Grant Response "authorization" Object (Section 7.4.3), and are represented by an AuthZ URI. If an AuthZ URI is included, the Client MAY refresh, update, and delete Authorizations.

An Authorization will persist independent of the Grant, and persist until invalidated by the GS per GS policy, or deleted by the Client.

5. GS APIs

Client Authentication

All APIs except for GS Options require the Client to authenticate.

This document defines the JOSE Authentication mechanism in Section 10. Other mechanisms include [TBD].

5.1. Create Grant

The Client creates a Grant by doing an HTTP POST of a JSON [RFC8259] document to the GS URI.

The JSON document MUST include the following from the Request JSON Section 5.5:

- *iat
- *nonce

*uri set to the GS URI

*client

and MAY include the following from Request JSON Section 5.5

*user

*interaction

*authorization or authorizations

*claims

*reciprocal

The GS MUST respond with one of Grant Response Section 7.1, Interaction Response Section 7.2, Wait Response Section 7.3, or one of the following errors:

*TBD

from Error Responses Section 9.

Following is a non-normative example where the Client wants to interact with the User with a popup and is requesting identity claims about the User and read access to the User's contacts:

Example 1

```
{
    "iat"       : 15790460234,
    "uri"       : "https://as.example/endpoint",
    "nonce"     : "f6a60810-3d07-41ac-81e7-b958c0dd21e4",
    "client": {
        "display": {
            "name"  : "SPA Display Name",
            "uri"   : "https://spa.example/about"
        }
    },
    "interaction": {
        "type"      : "popup"
    },
    "authorization": {
        "type"      : "oauth_scope",
        "scope"     : "read_contacts"
    },
    "claims": {
        "oidc": {
            "id_token" : {
                "email"          : { "essential" : true },
                "email_verified" : { "essential" : true }
            },
            "userinfo" : {
                "name"           : { "essential" : true },
                "picture"        : null
            }
        }
    }
}
```

Following is a non-normative example where the Client is requesting the GS to keep the interaction with the User after returning the ID Token so the Client can update the Grant, and is also asking if the user exists:

Example 2

```
{
    "iat"       : 15790460234,
    "uri"       : "https://as.example/endpoint",
    "nonce"     : "5c9360a5-9065-4f7b-a330-5713909e06c6",
    "client": {
        "id"        : "di3872h34dkJW"
    },
    "interaction": {
        "keep"      : true,
        "type"      : "redirect",
        "uri"       : "https://web.example/return"
    },
    "user": {
        "identifiers": {
            "email" : "jane.doe@example.com"
        },
        "exists"    : true
    },
    "claims"    : { "oidc": { "id_token" : {} } }
}
```

5.2. Read Grant

The Client reads a Grant by doing an HTTP GET of the corresponding Grant URI.

The GS MUST respond with one of Grant Response Section 7.1, Interaction Response Section 7.2, Wait Response Section 7.3, or one of the following errors:

- *TBD

from Error Responses Section 9.

5.3. Update Grant

The Client updates a Grant by doing an HTTP PUT of a JSON document to the corresponding Grant URI.

The JSON document MUST include the following from the Request JSON Section 5.5

- *iat

- *uri set to the Grant URI

and MAY include the following from Request JSON Section 5.5

- *user
- *interaction
- *authorization or authorizations
- *claims
- *reciprocal

The GS MUST respond with one of Grant Response Section 7.1, Interaction Response Section 7.2, Wait Response Section 7.3, or one of the following errors:

- *TBD

from Error Responses Section 9.

Following is a non-normative example where the Client made an "interaction"."keep":true request, and now wants to update the request with additional claims:

Example 3

```
{
    "iat"       : 15790460234,
    "uri"       : "https://as.example/endpoint/grant/example3",
    "claims": {
        "oidc": {
            "userinfo" : {
                "email"          : { "essential" : true },
                "name"           : { "essential" : true },
                "picture"        : null
            }
        }
    }
}
```

5.4. Delete Grant

The Client deletes a Grant by doing an HTTP DELETE of the corresponding Grant URI.

The GS MUST respond with OK 200, or one of the following errors:

- *TBD

from Error Responses Section 9.

5.5. Request JSON

[Editor: do we want to reuse the JWT claims "iat", "jti", etc.?]

*__iat__ - the time of the request as a NumericDate.

*__nonce__ - a unique identifier for this request. Note the Grant Response MUST contain a matching nonce attribute value.

*__uri__ - the GS URI if in a Create Grant Section 5.1, or the Grant URI if in an Update Grant Section 5.3.

5.5.1. "client" Object

The client object MUST contain either the id attribute for Registered Clients, or the display object for Dynamic Clients.

*__id__ - the Client ID the GS has for the Registered Client.

*__display__ - the display object contains the following attributes:

-__name__ - a string that represents the Dynamic Client

-__uri__ - a URI representing the Dynamic Client

[Editor: a max length for the name?] [Editor: a max length for the URI?]

The name and uri will be displayed by the GS when prompting for authorization.

5.5.2. "interaction" Object

The interaction object contains the type of interaction the Client will provide the User. Other attributes

*__keep__ - a JSON boolean. If set to the JSON value true, the GS will not transfer the User interaction back to the Client after processing the Grant request. The JSON value false is equivalent to the attribute not being present, and the GS will transfer the User interaction back to the Client after processing the request. This attribute is OPTIONAL

-__type__ - contains one of the following values: "popup", "redirect", or "qrcode". Details in Section 7.4.7. This attribute is REQUIRED.

-__redirect_uri__ - this attribute is REQUIRED if the type is "redirect". It is the URI that the Client requests the GS to redirect the User to after the GS has completed interacting

with the User. If the Client manages session state in URIs, then the redirect_uri SHOULD contain that state.

- **ui_locales** - End-User's preferred languages and scripts for the user interface, represented as a space-separated list of [RFC5646] language tag values, ordered by preference. This attribute is OPTIONAL.

[Editor: do we need max pixels or max chars for qrcode interaction? Either passed to GS, or max specified values here?]

[Editor: other possible interaction models could be a "webview", where the Client can display a web page, or just a "message", where the client can only display a text message]

[Editor: we may need to include interaction types for iOS and Android as the mobile OS APIs evolve.]

5.5.3. "user" Object

* **exists** - MUST contain the JSON true value. Indicates the Client requests the GS to return a "user"."exists" value in an Interaction Response Section 7.2. This attribute is OPTIONAL, and MAY be ignored by the GS.

* **identifiers** - REQUIRED if the exists attribute is present. The values MAY be used by the GS to improve the User experience. Contains one or more of the following identifiers for the User:

  - **phone_number** - contains a phone number per Section 5 of [RFC3966].

  - **email** - contains an email address per [RFC5322].

  - **oidc** - is an object containing both the "iss" and "sub" attributes from an OpenID Connect ID Token per [OIDC] Section 2.

5.5.4. "authorization" Object

* **type** - one of the following values: "oauth_scope" or "oauth_rich". This attribute is REQUIRED.

* **scope** - a string containing the OAuth 2.0 scope per [RFC6749] section 3.3. MUST be included if type is "oauth_scope" or "oauth_rich".

* **authorization_details** - an authorization_details object per [RAR]. MUST be included if type is "oauth_rich".

[Editor: details may change as the [RAR] document evolves]

5.5.5. "authorizations" Object

A JSON array of "authorization" objects. Only one of "authorization" or "authorizations" may be in the Request JSON.

[Editor: instead of an array, we could have a Client defined dictionary of "authorization" objects]

5.5.6. "claims" Object

Includes one or more of the following:

- **oidc** - an object that contains one or both of the following objects:
  - **userinfo** - Claims that will be returned as a JSON object
  - **id_token** - Claims that will be included in the returned ID Token. If the null value, an ID Token will be returned containing no additional Claims.

The contents of the userinfo and id_token objects are Claims as defined in [OIDC] Section 5.

- **oidc4ia** - OpenID Connect for Identity Assurance claims request per [OIDC4IA].
- **vc** - [Editor: define how W3C Verifiable Credentials [W3C_VC] can be requested.]

5.5.7. "reciprocal" Object

- **uri** - the Grant URI for the Reciprocal Grant. This attribute is REQUIRED.
- **client** - the client object must contain the "id" attribute with the Client ID the Grant was issued to. This attribute is REQUIRED.
- **authorization** - an authorization object per Section 7.4.3 in the Response JSON.
- **authorizations** - an authorizations object per Section 7.4.4 in the Response JSON.
- **claims** - a claims object per Section 7.4.5 in the Response JSON.

[Editor: parameters for the Client to request it wants the Grant Response signed and/or encrypted?]

5.6. Refresh Authorization

The Client updates an Authorization by doing an HTTP GET to the corresponding AuthZ URI.

The GS MUST respond with an Response JSON "authorization" object Section 7.4.3, or one of the following errors:

- *TBD

from Error Responses Section 9.

5.7. Update Authorization

The Client updates an Authorization by doing an HTTP PUT to the corresponding AuthZ URI of the following JSON. All of the following MUST be included.

- ***iat** - the time of the response as a NumericDate.
- ***uri** - the AuthZ URI.
- ***authorization** - the new authorization requested per the Request JSON "authorization" object Section 5.5.4.

The GS MUST respond with an Response JSON "authorization" object Section 7.4.3, or one of the following errors:

- *TBD

from Error Responses Section 9.

5.8. Delete Authorization

The Client deletes an Authorization by doing an HTTP DELETE to the corresponding AuthZ URI.

The GS MUST respond with OK 200, or one of the following errors:

- *TBD

from Error Responses Section 9.

5.9. GS Options

The Client can get the metadata for the GS by doing an HTTP OPTIONS of the corresponding GS URI. This is the only API where the GS MAY respond to an unauthenticated request.

The GS MUST respond with the the following JSON document:

[Editor: this section is a work in progress]

- ***uri** - the GS URI.

- ***client_authentication** - an array of the Client Authentication mechanisms supported by the GS

- ***interactions** - an array of the interaction types supported by the GS.

- ***authorization** - an object containing the authorizations the Client may request from the GS, if any.

  - Details TBD

- ***claims** - an object containing the identity claims the Client may request from the GS, if any, and what public keys the claims will be signed with.

  - Details TBD

- ***algorithms** - a list of the cryptographic algorithms supported by the GS.

- ***features** - an object containing feature support

  - **user_exists** - boolean indicating if "user"."exists" is supported.

  - **authorizations** - boolean indicating if a request for multiple authorizations is supported.

[Editor: keys used by Client to encrypt requests, or verify signed responses?]

[Editor: namespace metadata for extensions?]

or one of the following errors:

- *TBD

from Error Responses Section 9.

5.10. Grant Options

The Client can get the metadata for the Grant by doing an HTTP OPTIONS of the corresponding Grant URI.

The GS MUST respond with the the following JSON document:

*__verbs__ - an array of the HTTP verbs supported at the GS URI.

or one of the following errors:

*TBD

from Error Responses Section 9.

5.11. AuthZ Options

The Client can get the metadata for the AuthZ by doing an HTTP OPTIONS of the corresponding AuthZ URI.

The GS MUST respond with the the following JSON document:

*__verbs__ - an array of the HTTP verbs supported at the GS URI.

or one of the following errors:

*TBD

from Error Responses Section 9.

5.12. Request Verification

On receipt of a request, the GS MUST verify the following:

*TBD

6. GS Initiated Grant

[Editor: In OAuth 2.0, all flows are initiated at the Client. If the AS wanted to initiate a flow, it redirected to the Client, that redirected back to the AS to initiate a flow.

Here is a proposal to support GS initiated: authentication; just-in-time (JIT) provisioning; and authorization]

initiation_uri A URI at the Client that contains no query or fragment. How the GS learns the Client initiation_uri is out of scope.

The GS creates a Grant and Grant URI, and redirects the User to the initiation_uri with the query parameter "grant" and the value of Grant URI.

See Section 3.3 for the sequence diagram.

7. GS API Responses

7.1. Grant Response

The Grant Response MUST include the following from the Response JSON Section 7.4

- *iat
- *nonce
- *uri
- *expires_in

and MAY include the following from the Response JSON Section 7.4

- *authorization or authorizations
- *claims
- *reciprocal

Example non-normative Grant Response JSON document for Example 1 in Section 3.1:

```
{
    "iat"           : 15790460234,
    "nonce"         : "f6a60810-3d07-41ac-81e7-b958c0dd21e4",
    "uri"           : "https://as.example/endpoint/grant/example1",
    "expires_in"    : 300
    "authorization": {
        "type"          : "oauth_scope",
        "scope"         : "read_contacts",
        "expires_in"    : 3600,
        "mechanism"     : "bearer",
        "token"         : "eyJJ2D6.example.access.token.mZf9p"
    },
    "claims": {
        "oidc": {
            "id_token"      : "eyJhbUzI1N.example.id.token.YRw5DFdbW",
            "userinfo" : {
                "name"      : "John Doe",
                "picture"   : "https://photos.example/p/eyJzdkiO"
            }
        }
    }
}
```

Example non-normative Grant Response JSON document for Example 2 in Section 3.1:

```
{
    "iat"   : 15790460234,
    "nonce" : "5c9360a5-9065-4f7b-a330-5713909e06c6",
    "uri"   : "https://as.example/endpoint/grant/example2",
    "authorization": {
        "type"          : "oauth_scope",
        "scope"         : "read_calendar write_calendar",
        "expires_in"    : 3600,
        "mechanism"     : "jose",
        "token"         : "eyJJ2D6.example.access.token.mZf9p"
        "certificate": {
            "x5u"   : "https://as.example/cert/example2"
        },
        "uri"       : "https://as.example/endpoint/authz/example2"
    }
}
```

7.2. Interaction Response

The Interaction Response MUST include the following from the Response JSON Section 7.4

- *iat
- *nonce
- *uri
- *interaction

and MAY include the following from the Response JSON Section 7.4

- *user
- *wait

A non-normative example of an Interaction Response follows:

```
{
    "iat"       : 15790460234,
    "nonce"     : "0d1998d8-fbfa-4879-b942-85a88bff1f3b",
    "uri"       : "https://as.example/endpoint/grant/example4",
    "interaction" : {
        "type"      : "popup",
        "uri"       : "https://as.example/popup/example4"
    },
    "user": {
        "exists" : true
    }
}
```

7.3. Wait Response

The Wait Response MUST include the following from the Response JSON Section 7.4

- *iat
- *nonce
- *uri
- *wait

A non-normative example of an Wait Response follows:

```
{
    "iat"       : 15790460234,
    "nonce"     : "0d1998d8-fbfa-4879-b942-85a88bff1f3b",
    "uri"       : "https://as.example/endpoint/grant/example5",
    "wait"      : 300
}
```

7.4. Response JSON

Details of the JSON document:

- ***iat** - the time of the response as a NumericDate.
- ***nonce** - the nonce that was included in the Request JSON Section 5.5.
- ***uri** - the Grant URI.
- ***wait** - a numeric value representing the number of seconds the Client should want before making a Read Grant request to the Grant URI.

*__expires_in__ - a numeric value specifying how many seconds until the Grant expires. This attribute is OPTIONAL.

7.4.1. "interaction" Object

If the GS wants the Client to start the interaction, the GS MUST select one of the interaction mechanisms provided by the Client in the Grant Request, and include the matching attribute in the interaction object:

*__type__ - this MUST match the type provided by the Client in the Grant Request client.interaction object.

*__uri__ - the URI to interact with the User per the type. This may be a temporary short URL if the type is qrcode so that it is easy to scan.

*__message__ - a text string to display to the User if type is qrcode.

[Editor: do we specify a maximum length for the uri and message so that a device knows the maximum it needs to support? A smart device may have limited screen real estate.]

7.4.2. "user" Object

*__exists__ - a boolean value indicating if the GS has a user with one or more of the provided identifiers in the Request "user"."identifiers" object Section 5.5.3

7.4.3. "authorization" Object

The "authorization" object is a response to the Request "authorization" object Section 5.5.4, the Refresh Authorization Section 5.6, or the Update Authorization Section 5.7.

*__type__ - the type of claim request: "oauth_scope" or "oauth_rich". See the "type" object in Section 5.5.4 for details.

*__scope__ - the scopes the Client was granted authorization for. This will be all, or a subset, of what was requested. This attribute is OPTIONAL.

*__authorization_details__ - the authorization details granted per [RAR]. Included if type is "oauth_rich".

*__mechanism__ - one of the access mechanisms: "bearer", "jose", or "jose+body". See Section 8 for details.

*__token__ - the access token for accessing an RS. This attribute is REQUIRED.

*__expires_in__ - a numeric value specifying how many seconds until the access token expires. This attribute is OPTIONAL.

*__certificate__ - MUST be included if the mechanism is "jose" or "jose+body". Contains the jwk header values for the Client to include in the JWS header when calling the RS using the "jose" or "jose+body" mechanisms. See Section 10.2.1.

*__uri__ - the AuthZ URI. Used to refresh, update, and delete the authorization. This attribute is OPTIONAL.

[Editor: any value in an expiry for the Authorization?]

7.4.4. "authorizations" Object

A JSON array of authorization objects. Support for the authorizations object is OPTIONAL.

7.4.5. "claims" Object

The claims object is a response to the Request "claims" object Section 5.5.4.

*__oidc__

-__id_token__ - an OpenID Connect ID Token containing the Claims the User consented to be released.

-__userinfo__ - the Claims the User consented to be released.

Claims are defined in [OIDC] Section 5.

*__oidc4ia__ - OpenID Connect for Identity Assurance claims response per [OIDC4IA].

*__vc__

The verified claims the user consented to be released. [Editor: details TBD]

7.4.6. "reciprocal" Object

The following MUST be included

*__nonce__ - a unique identifier for this request. Note the Grant Response MUST contain a matching nonce attribute value.

*__client__

-__id__ - the Client ID making the request

One or more of the following objects from the Request JSON Section 5.5 are included:

* **authorization** Section 7.4.3

* **authorizations** Section 7.4.4

* **claims** Section 7.4.5

7.4.7. Interaction Types

If the GS wants the Client to initiate the interaction with the User, then the GS will return an Interaction Response. The Client will initiate the interaction with the User in one of the following ways:

* **popup** The Client will create a new popup child browser window containing the "interaction"."uri" attribute. [Editor: more details on how to do this]

The GS will close the popup window when the interactions with the User are complete. [Editor: confirm GS can do this still on all browsers, or does Client need to close]

* **redirect** The Client will redirect the User to the "interaction"."uri" attribute. When the GS interactions with the User are complete, the GS will redirect the User to the "interaction"."redirect_uri" attribute the Client provided in the Grant Request.

* **qrcode** The Client will create a [QR Code] of the "interaction"."uri" attribute and display the resulting graphic and the "interaction"."message" attribute as a character string.

An GS MUST support the "popup", "redirect", and "qrcode" interaction types.

7.4.8. Signing and Encryption

[Editor: TBD - how response is signed and/or encrypted by the GS. Is there a generalized description, or is it mechanism specific?]

7.5. Response Verification

On receipt of a response, the Client MUST verify the following:

* TBD

8. RS Access

This document specifies three different mechanisms for the Client to access an RS ("bearer", "jose", and "jose+body"). The "bearer" mechanism is defined in {BearerToken}. The "jose" and "jose+body" mechanisms are proof-of-possession mechanisms and are defined in Section 10.2.2 and Section 10.2.3 respectively. Additional proof-of-possession mechanisms may be defined in other documents. The mechanism the Client is to use with an RS is the Response JSON "authorization"."mechanism" attribute Section 7.4.3.

8.1. Bearer Token

If the access mechanism is "bearer", then the Client accesses the RS per Section 2.1 of [RFC6750]

A non-normative example of the HTTP request headers follows:

```
GET /calendar HTTP/2
Host: calendar.example
Authorization: bearer eyJJ2D6.example.access.token.mZf9pTSpA
```

9. Error Responses

*TBD

10. JOSE Authentication

How the Client authenticates to the GS and RS are independent of each other. One mechanism can be used to authenticate to the GS, and a different mechanism to authenticate to the RS.

Other documents that specify other Client authentication mechanisms will replace this section.

In the JOSE Authentication Mechanism, the Client authenticates by using its private key to sign a JSON document with JWS per [RFC7515] which results in a token using JOSE compact serialization.

[Editor: are there advantages to using JSON serialization in the body?]

Different instances of a Registered Clients MAY have different private keys, but certificates bind them to a public key the GS has for the Client ID. An instance of a Client will use the same private key for all signing.

The Client and the GS MUST both use HTTP/2 ([RFC7540]) or later, and TLS 1.3 ([RFC8446]) or later, when communicating with each other.

[Editor: too aggressive to mandate HTTP/2 and TLS 1.3?]

The token may be included in an HTTP header, or as the HTTP message body.

The following sections specify how the Client uses JOSE to authenticate to the GS and RS.

10.1. GS Access

The Client authenticates to the GS by passing either a signed header parameter, or a signed message body. The following table shows the verb, uri and token location for each Client request to the GS:

request	http verb	uri	token in
Create Grant	POST	GS URI	body
Read Grant	GET	Grant URI	header
Update Grant	PUT	Grant URI	body
Delete Grant	DELETE	Grant URI	success
Refresh AuthZ	GET	AuthZ URI	body
Update AuthZ	PUT	AuthZ URI	body
Delete AuthZ	DELETE	AuthZ URI	header
GS Options	OPTIONS	GS URI	header
Grant Options	OPTIONS	Grant URI	header
AuthZ Options	OPTIONS	AuthZ URI	header

Table 2

10.1.1. Authorization Header

For requests with the token in the header, the JWS payload MUST contain the following attributes:

iat - the time the token was created as a NumericDate.

jti - a unique identifier for the token per [RFC7519] section 4.1.7.

uri - the value of the URI being called (GS URI, Grant URI, or AuthZ URI).

verb - the HTTP verb being used in the call ("GET", "DELETE", "OPTIONS")

The HTTP authorization header is set to the "jose" parameter followed by one or more white space characters, followed by the resulting token.

A non-normative example of a JWS payload and the HTTP request follows:

```
{
    "iat"   : 15790460234,
    "jti"   : "f6d72254-4f23-417f-b55e-14ad323b1dc1",
    "uri"   : "https://as.example/endpoint/grant/example6",
    "verb"  : "GET"
}
```

```
GET /endpoint/example.grant HTTP/2
Host: as.example
Authorization: jose eyJhbGciOiJIUzI1NiIsIn ...
```

[Editor: make a real example token]

GS Verification

The GS MUST verify the token by:

- *TBD

10.1.2. Signed Body

For requests with the token in the body, the Client uses the Request JSON as the payload in a JWS. The resulting token is sent with the content-type set to "application/jose".

A non-normative example (line breaks added to the body for readability):

```
POST /endpoint HTTP/2
Host: as.example
Content-Type: application/jose
Content-Length: 155

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyzdWIiOiIxMjM0NTY3ODkwIiwibmF
tZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMe
Jf36POk6yJV_adQssw5c
```

[Editor: make a real example token]

GS Verification

The GS MUST verify the token by:

- *TBD

10.1.3. Public Key Resolution

- ***Registered Clients** can use any of the JWS header values to direct the GS to resolve the public key matching the private key used to

the Client ID. The GS MAY restrict with JWS headers a Client can use.

[Editor: would examples help here so that implementors understand the full range of options, and how an instance can have its own asymetric key pair]

A non-normative example of a JOSE header for a Registered Client with a key identifier of "12":

```
{
    "alg"   : "ES256",
    "typ"   : "JOSE",
    "kid"   : "12"
}
```

*__Dynamic Clients__ include their public key in the "jwk" JWS header.

A non-normative example of a JOSE header for a Dynamic Client:

```
{
    "alg"   : "ES256",
    "typ"   : "JOSE",
    "jwk"   : {
        "kty"   : "EC",
        "crv"   : "P-256",
        "x"     : "Kgl5DJSgLyV-G32osmLhFKxJ97FoMW0dZVEqDG-Cwo4",
        "y"     : "GsL4mOM4x2e6iON8BHvRDQ6AgXAPnw0m0SfdlREV7i4"
    }
}
```

10.2. RS Access

In the "jose" mechanism Section 10.2.2, all Client requests to the RS include a proof-of-possession token in the HTTP authorization header. In the "jose+body" mechanism Section 10.2.3, the Client signs the JSON document in the request if the POST or PUT verbs are used, otherwise it is the same as the "jose" mechanism.

10.2.1. JOSE header

The GS provides the Client one or more JWS header parameters and values for a a certificate, or a reference to a certificate or certificate chain, that the RS can use to resolve the public key matching the private key being used by the Client.

A non-normative examples JOSE header:

```
{
    "alg"   : "ES256",
    "typ"   : "JOSE",
    "x5u"   : "https://as.example/cert/example2"
}
```

[Editor: this enables Dynamic Clients to make proof-of-possession API calls the same as Registered Clients.]

10.2.2. "jose" Mechanism

The JWS payload MUST contain the following attributes:

iat - the time the token was created as a NumericDate.

jti - a unique identifier for the token per [RFC7519] section 4.1.7.

uri - the value of the RS URI being called.

verb - the HTTP verb being used in the call

token - the access token provided by the GS to the Client

The HTTP authorization header is set to the "jose" parameter followed by one or more white space characters, followed by the resulting token.

A non-normative example of a JWS payload and the HTTP request follows:

```
{
    "iat"   : 15790460234,
    "jti"   : "f6d72254-4f23-417f-b55e-14ad323b1dc1",
    "uri"   : "https://calendar.example/calendar",
    "verb"  : "GET",
    "token" : "eyJJ2D6.example.access.token.mZf9pTSpA"
}
```

```
GET /calendar HTTP/2
Host: calendar.example
Authorization: jose eyJhbG.example.jose.token.adks
```

[Editor: make a real example token]

RS Verification

The RS MUST verify the token by:

- *verify access token is bound to the public key - include key fingerprint in access token?

*TBD

10.2.3. "jose+body" Mechanism

The "jose+body" mechanism can only be used if the content being sent to the RS is a JSON document.

Any requests not sending a message body will use the "jose" mechanism Section 10.2.2.

Requests sending a message body MUST have the following JWS payload:

iat - the time the token was created as a NumericDate.

jti - a unique identifier for the token per [RFC7519] section 4.1.7.

uri - the value of the RS URI being called.

verb - the HTTP verb being used in the call

token - the access token provided by the GS to the Client

body - the message body being sent to the RS

A non-normative example of a JWS payload and the HTTP request follows:

```
{
    "iat"   : 15790460234,
    "jti"   : "f6d72254-4f23-417f-b55e-14ad323b1dc1",
    "uri"   : "https://calendar.example/calendar",
    "verb"  : "POST",
    "token" : "eyJJ2D6.example.access.token.mZf9pTSpA",
    "payload" : {
        "event" : {
            "title"             : "meeting with joe",
            "start_date_utc"    : "2020-02-21 11:00:00",
            "end_date_utc"      : "2020-02-21 11:00:00"
        }
    }
}
```

```
POST /calendar HTTP/2
Host: calendar.example
Content-Type: application/jose
Content-Length: 155

eyJhbGciOi.example.jose+body.adasdQssw5c
```

[Editor: make a real example token]

RS Verification

The RS MUST verify the token by:

* TBD

10.2.4. Public Key Resolution

The RS has a public key for the GS that it uses to verify the certificate or certificate chain the Client includes in the JWS header.

10.3. Request Encryption

[Editor: to be fleshed out]

The Client encrypts a request when ??? using the GS public key returned as the ??? attribute in GS Options Section 5.9.

10.4. Response Signing

[Editor: to be fleshed out]

The Client verifies a signed response ??? using the GS public key returned as the ??? attribute in GS Options Section 5.9.

10.5. Response Encryption

[Editor: to be fleshed out]

The Client decrypts a response when ??? using the private key matching the public key included in the request as the ??? attribute in Section 5.5.

11. Extensibility

This standard can be extended in a number of areas:

* **Client Authentication Mechanisms**

  - An extension could define other mechanisms for the Client to authenticate to the GS and/or RS such as Mutual TLS or HTTP Signing. Constrained environments could use CBOR [RFC7049] instead of JSON, and COSE [RFC8152] instead of JOSE, and CoAP [RFC8323] instead of HTTP/2.

* **Grant**

  - An extension can define new objects in the Grant Request and Grant Response JSON.

* **Top Level**

  - Top level objects SHOULD only be defined to represent functionality other the existing top level objects and attributes.

* **"client" Object**

  - Additional information about the Client that the GS would require related to an extension.

* **"user" Object**

  - Additional information about the User that the GS would require related to an extension.

* **"authorization" Object**

  - Additional types of authorizations in addition to OAuth 2.0 scopes and RAR.

* **"claims" Object**

  - Additional types of identity claims in addition to OpenID Connect claims and Verified Credentials.

* **Interaction types**

  - Additional types of interactions a Client can start with the User.

* **Continuous Authentication**

  - An extension could define a mechanism for the Client to regularly provide continuous authentication signals and receive responses.

[Editor: do we specify access token / handle introspection in this document, or leave that to an extension?]

[Editor: do we specify access token / handle revocation in this document, or leave that to an extension?]

12. Rational

1. **Why is there only one mechanism for the Client to authenticate with the GS? Why not support other mechanisms?**

   Having choices requires implementers to understand which choice is preferable for them. Having one default mechanism in the

document for the Client to authenticate simplifies most implementations. Deployments that have unique characteristics can select other mechanisms that are preferable in specific environments.

2. **Why is the default Client authentication JOSE rather than MTLS?**

   MTLS cannot be used today by a Dynamic Client. MTLS requires the application to have access below what is typically the application layer, that is often not available on some platforms. JOSE is done at the application layer. Many GS deployments will be an application behind a proxy performing TLS, and there are risks in the proxy passing on the results of MTLS.

3. **Why is the default Client authentication JOSE rather than HTTP signing?**

   There is currently no widely deployed open standard for HTTP signing. Additionally, HTTP signing requires passing all the relevant parts of the HTTP request to downstream services within an GS that may need to independently verify the Client identity.

4. **What are the advantages of using JOSE for the Client to authenticate to the GS and a resource?**

   Both Registered Clients and Dynamic Clients can have a private key, eliminating the public Client issues in OAuth 2.0, as a Dynamic Client can create an ephemeral key pair. Using asymetric cryptography also allows each instance of a Registered Client to have its own private key if it can obtain a certificate binding its public key to the public key the GS has for the Client. Signed tokens can be passed to downstream components in a GS or RS to enable independent verification of the Client and its request. The GS Initiated Sequence Section 6 requires a URL safe parameter, and JOSE is URL safe.

5. **Why does the GS not return parameters to the Client in the redirect url?**

   Passing parameters via a browser redirection is the source of many of the security risks in OAuth 2.0. It also presents a challenge for smart devices. In this protocol, the redirection from the Client to the GS is to enable the GS to interact with the User, and the redirection back to the Client is to hand back interaction control to the Client if the redirection was a full browser redirect. Unlike OAuth 2.0, the identity of the Client is independent of the URI the GS redirects to.

6. **Why is there not a UserInfo endpoint as there is with OpenID Connect?**

   Since the Client can Read Grant at any time, it get the same functionality as the UserInfo endpoint, without the Client having to manage a separate access token and refresh token. If the Client would like additional claims, it can Update Grant, and the GS will let the Client know if an interaction is required to get any of the additional claims, which the Client can then start.

   [Editor: is there some other reason to have the UserInfo endpoint?]

7. **Why is there still a Client ID?**

   The GS needs an identifier to fetch the meta data associated with a Client such as the name and image to display to the User, and the policies on what a Client is allowed to do. The Client ID was used in OAuth 2.0 for the same purpose, which simplifies migration. Dynamic Clients do not have a Client ID.

8. **Why have both claims and authorizations?**

   There are use cases for each that are independent: authenticating a user vs granting access to a resource. A request for an authorization returns an access token or handle, while a request for a claim returns the claim.

9. **Why specify HTTP/2 or later and TLS 1.3 or later for Client and GS communication in ?**Section 10

   Knowing the GS supports HTTP/2 enables a Client to set up a connection faster. HTTP/2 will be more efficient when Clients have large numbers of access tokens and are frequently refreshing them at the GS as there will be less network traffic. Mandating TLS 1.3 similarly improves the performance and security of Clients and GS when setting up a TLS connection.

10. **Why do some of the JSON objects only have one child, such as the identifiers object in the user object in the Grant Request?**

    It is difficult to forecast future use cases. Having more resolution may mean the difference between a simple extension, and a convoluted extension.

11. **Why is the "iss" included in the "oidc" identifier object? Would the "sub" not be enough for the GS to identify the User?**

This decouples the GS from the OpenID Provider (OP). The GS identifier is the GS URI, which is the endpoint at the GS. The OP issuer identifier will likely not be the same as the GS URI. The GS may also provide claims from multiple OPs.

12. **Why complicate the sequence with "interaction"."keep"?**

    A common pattern is to use an GS to authenticate the User at the Client. The Client does not know a priori if the User is a new User, or a returning User. Asking a returning User to consent releasing identity claims and/or authorizations they have already provided is a poor User experience, as is sending the User back to the GS. The Client requesting identity first enables the Client to get a response from the GS while the GS is still interacting with the User, so that the Client can request any identity claims/or authorizations required or desired. Additionally, the claims a Client may want about a User may be dependent on some initial Claims. For example, if a User is in a particular country, additional or different Claims my be required by the Client

13. **Why is there a "jose+body" RS access mechanism method for the Client?**Section 10.2.3

    There are numerous use cases where the RS wants non-repudiation and providence of the contents of an API call. For example, the UGS Service Supplier Framework for Authentication and Authorization [UTM].

14. **Why use URIs to instead of handles for the Grant and Authorization?**

    A URI is an identifier just like a handle that can contain GS information that is opaque to the Client - so it has all the features of a handle, plus it can be the URL that is resolved to manipulate a Grant or an Authorization. As the Grant URI and AuthZ URI are defined to start with the GS URI, the Client (and GS) can easily determine which GS a Grant or Authorization belong to. URIs also enable a RESTful interface to the GS functionality.

15. **Why have an OPTIONS verb on the GS URI? Why not use a .well-known mechanism?**

    Having the GS URI endpoint respond to the metadata allows the GS to provide Client specific results using the same Client authentication used for other requests to the GS. It also reduces the risk of a mismatch between what the advertised metadata, and the actual metadata. A .well-known discovery

mechanism may be defined to resolve from a hostname to the GS URI.

16. **Why have an UPDATE, DELETE, and OPTIONS verbs on the AuthZ URI?**

    Maybe there are no use cases for them [that the editor knows of], but the GS can not implement, and they are available if use cases come up.

17. **Why list explicit interactions, instead of the Client and GS negotiating interaction capabilities?**

    The Client knows what interactions it is capable of, and prefers. Telling the GS the interaction allows the GS to know what interaction the User will have. Knowing how the Client will transition the interaction will enable the GS to provider a better User experience. For example, the GS may want to provide a short URL if it knows the Client will be showing a QR code vs a redirect, or have a different layout if it is a popup vs a redirect.

13. Acknowledgments

This draft derives many of its concepts from Justin Richer's Transactional Authorization draft [TxAuth].

Additional thanks to Justin Richer for his strong critique of earlier drafts.

14. IANA Considerations

[JOSE parameter for Authorization HTTP header]

TBD

15. Security Considerations

TBD

16. References

16.1. Normative References

[RFC3966] Schulzrinne, H., "The tel URI for Telephone Numbers", RFC 3966, DOI 10.17487/RFC3966, December 2004, <https://www.rfc-editor.org/info/rfc3966>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/info/rfc5322>.

[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>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/info/rfc7516>.

[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>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[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>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[OIDC] Sakimora, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", 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>.

16.2. Informative References

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

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

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

[RFC8323] Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., and B. Raymor, Ed., "CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets", RFC 8323, DOI 10.17487/RFC8323, February 2018, <https://www.rfc-editor.org/info/rfc8323>.

[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>.

[browser_based_apps] Parecki, A. and D. Waite, "OAuth 2.0 for Browser-Based Apps", September 2019, <https://tools.ietf.org/html/draft-ietf-oauth-browser-based-apps-04>.

[RAR] Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", January 2020, <https://tools.ietf.org/html/draft-ietf-oauth-rar-00>.

[W3C_VC] Sporny, M., Noble, G., and D. Chadwick, "Verifiable Credentials Data Model 1.0", November 2019, <https://w3c.github.io/vc-data-model/>.

[QR_Code] "ISO/IEC 18004:2015 - Information technology - Automatic identification and data capture techniques - QR Code bar code symbology specification", February 2015, <https://www.iso.org/standard/62021.html>.

[TxAuth] Richer, J., "Transactional AuthN", December 2019, <https://tools.ietf.org/html/draft-richer-transactional-authz-04>.

[UTM] Rios, J., Smith, I., and P. Venkatesen, "UGS Service Supplier Framework for Authentication and AuthN",

September 2019, <https://utm.arc.nasa.gov/docs/2019-UTM_Framework-NGSA-TM220364.pdf>.

Appendix A. Document History

A.1. draft-hardt-xauth-protocol-00

*Initial version

A.2. draft-hardt-xauth-protocol-01

*text clean up

*added OIDC4IA claims

*added "jws" method for accessing a resource.

*renamed Initiation Request -> Grant Request

*renamed Initiation Response -> Interaction Response

*renamed Completion Request -> Authorization Request

*renamed Completion Response -> Grant Request

*renamed completion handle -> authorization handle

*added Authentication Request, Authentication Response, authentication handle

A.3. draft-hardt-xauth-protocol-02

*handles are now URIs

*the

*the collection of claims and authorizations are a Grant

*

Appendix B. Comparison with OAuth 2.0 and OpenID Connect

Changed Features

The major changes between this protocol and OAuth 2.0 and OpenID Connect are:

*The Client uses a private key to authenticate in this protocol instead of the client secret in OAuth 2.0 and OpenID Connect.

*The Client initiates the protocol by making a signed request directly to the GS instead of redirecting the User to the GS.

*The Client does not pass any parameters in redirecting the User to the GS, nor receive any parameters in the redirection back from the GS.

*The refresh_token has been replaced with a AuthZ URI that both represents the access, and is the URI to call to refresh access.

*The Client can request identity claims to be returned independent of the ID Token. There is no UserInfo endpoint to query claims as there is in OpenID Connect.

*The GS URI is the token endpoint. CHECK!!!s

Preserved Features

*This protocol reuses the OAuth 2.0 scopes, Client IDs, and access tokens of OAuth 2.0.

*This protocol reuses the Client IDs, Claims and ID Token of OpenID Connect.

*No change is required by the Client or the RS for existing bearer token protected APIs.

New Features

*A Grant represents the user identity claims and RS access granted to the Client.

*The Client can update, retrieve, and delete a Grant.

*The GS can initiate a flow by creating a Grant and redirecting the User to the Client with the Grant URI.

*The Client can discovery if an GS has a User with an identifier before the GS interacts with the User.

*The Client can request the GS to first authenticate the User and return User identity claims, and then the Client can update Grant request based on the User identity.

*Support for QR Code initiated interactions.

*Each Client instance can have its own private / public key pair.

*More Extensibility dimensions.

Appendix C. Open Questions

1.

Author's Address

Dick Hardt (editor)
SignIn.Org
United States

Email: dick.hardt@gmail.com