Draft

N. Sakimura
NRI
J. Bradley
Ping Identity
M. Jones
Microsoft
B. de Medeiros
Google
C. Mortimore
Salesforce
October 15, 2013

TOC

OpenID Connect Core 1.0 - draft 14

Abstract

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It enables Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.

This specification defines the core OpenID Connect functionality: authentication built on top of OAuth 2.0 and the use of Claims to communicate information about the End-User. It also describes the security and privacy considerations for using OpenID Connect.

Table of Contents

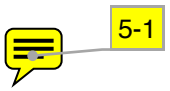
---

1. Introduction

TOC

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It enables Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.

The OpenID Connect Core 1.0 specification defines the core OpenID Connect functionality: authentication built on top of OAuth 2.0 and the use of Claims to communicate information about the End-User. It also describes the security and privacy considerations for using OpenID Connect.



As background, the **OAuth 2.0 Authorization Framework** [RFC6749] and **OAuth 2.0 Bearer Token Usage** [RFC6750] specifications provide a general framework for third-party applications to obtain and use limited access to HTTP resources. They define mechanisms to obtain and use Access Tokens to access resources but do not define standard methods to provide identity information. Notably, without profiling OAuth 2.0, it is incapable of providing information about the authentication of an End-User.

This specification assumes that the Client has already obtained the locations of the OpenID Provider's endpoints, including its Authorization Endpoint and Token Endpoint. These URLs are normally obtained via Discovery, as described in **OpenID Connect Discovery 1.0** [OpenID.Discovery], or MAY be obtained via other mechanisms.

Likewise, this specification assumes that the Client has already obtained sufficient credentials to interact with the OpenID Provider. These credentials are normally obtained via Dynamic Registration, as described in **OpenID Connect Dynamic Client Registration 1.0** [OpenID.Registration], or MAY be obtained via other mechanisms.

1.1. Requirements Notation and Conventions

TOC

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in **RFC 2119** [RFC2119].

Throughout this document, values are quoted to indicate that they are to be taken literally. When using these values in protocol messages, the quotes MUST NOT be used as part of the value.

All uses of **JSON Web Signature (JWS)** [JWS] and **JSON Web Encryption (JWE)** [JWE] data structures in this specification utilize the JWS Compact Serialization or the JWE Compact Serialization; the JWS JSON Serialization and the JWE JSON Serialization are not used.

1.2. Terminology

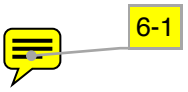
TOC

This section defines the terminology used by this specification. This section is a normative portion of this specification, imposing requirements upon implementations. All the capitalized words in the text of this specification, such as "Issuer Identifier", reference these defined terms. Whenever the reader encounters them, their definitions found in this section must be followed.

This specification uses the terms "Access Token", "Refresh Token", "Authorization Code", "Authorization Grant", "Authorization Server", "Authorization Endpoint", "Client", "Client Identifier", "Client Secret", "Protected Resource", "Resource Owner", "Resource Server", and "Token Endpoint" defined by **OAuth 2.0** [RFC6749], and the terms "Claim Names" and "Claim Values" defined by **JSON Web Token (JWT)** [JWT].

This specification also defines the following terms:

Authentication
: Process of verifying that an Entity is the owner of an Identity. Typically this involves the verification of the current or past possession of particular credentials, including what the entity knows, possesses, has as physical features, or behaviors, or combinations of these utilizing heuristics. The entity is often an End-User or a Client.



Authentication Request
: An OAuth 2.0 Authorization Request that requests that the End-User be authenticated by the Authorization Server.

Authentication Context
: Information that the Relying Party can require before it makes an entitlement decision with respect to an authentication response. Such context can include, but is not limited to, the actual authentication method used or level of assurance such as **ISO/IEC 29115** [ISO29115] entity authentication assurance level.

Authentication Context Class
: Set of authentication methods or procedures that are considered to be equivalent to each other in a particular context.

Authentication Context Class Reference
: Identifier for an Authentication Context Class.

Authorization Code Flow
: OAuth 2.0 flow in which all tokens are returned from the Token Endpoint.

Claim
: Piece of information asserted about an Entity.

Claim Type
: Syntax used for representing a Claim Value. This specification defines Normal, Aggregated, and Distributed Claim Types.

Claims Provider
: Server that can return Claims about an Entity.

Credential

Data presented as evidence of the right to use an identity or other resources.

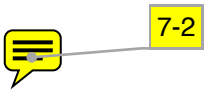
End-User
: Human Resource Owner.

Entity
: Something that has a separate and distinct existence and that can be identified in a context. An End-User is one example of an Entity.

Essential Claim
: Claim specified by the Client as being necessary to ensure a smooth authorization experience for the specific task requested by the End-User.

Hybrid Flow
: OAuth 2.0 flow in which some tokens are returned from the Authorization Endpoint and others are returned from the Token Endpoint.

ID Token
: **JSON Web Token (JWT)** [JWT] that contains Claims about the authentication event. It MAY contain other Claims.



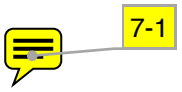
Identifier
: Value that uniquely characterizes an Entity in a specific context.

Identity
: Set of attributes related to an Entity.

Implicit Code Flow
: OAuth 2.0 flow in which all tokens are returned from the Authorization Endpoint.

Issuer
: Entity that issues a set of Claims.

Issuer Identifier
: Verifiable Identifier for an Issuer. An Issuer Identifier is a case sensitive URL using the `https` scheme that contains scheme, host, and OPTIONALLY, port number and path components and no query or fragment components.



Message
: Request or a response between an OpenID Relying Party and an OpenID Provider.

OpenID Provider (OP)
: OAuth 2.0 Authorization Server that is capable of providing Claims to a Relying Party about the authentication event and the End-User in an ID Token and/or a UserInfo Endpoint response.

OP Endpoints
: Authorization Endpoint, Token Endpoint, and UserInfo Endpoint.

Request Object
: JWT that contains a set of request parameters as its Claims.

Request URI
: URL that references a resource containing a Request Object. The Request URI contents MUST be retrievable by the Authorization Server.

Pairwise Pseudonymous Identifier (PPID)
: Identifier that identifies the Entity to a Relying Party that cannot be correlated with the Entity's PPID at another Relying Party.

Personally Identifiable Information (PII)
: Information that (a) can be used to identify the natural person to whom such information relates, or (b) is or might be directly or indirectly linked to a natural person to whom such information relates.

Relying Party (RP)
: OAuth 2.0 Client application requiring Claims from an OpenID Provider.

Self-Issued OpenID Provider

Personal OpenID Provider that issues self-signed ID Tokens.

UserInfo Endpoint
: Protected resource that, when presented with an Access Token by the Client, returns authorized information about the End-User represented by the corresponding Authorization Grant.

Validation
: Process intended to establish the soundness or correctness of a construct.

Verification
: Process intended to test or prove the truth or accuracy of a fact or value.

Voluntary Claim
: Claim specified by the Client as being useful but not Essential for the specific task requested by the End-User.

For more background on some of the terminology used, see **Internet Security Glossary, Version 2** [RFC4949], **ISO/IEC 29115 Entity Authentication Assurance** [ISO29115], and **ITU-T X.1252** [X.1252].

1.3. Overview

TOC

The OpenID Connect protocol, in abstract, follows the following steps.

1. The RP (Client) sends a request to the OpenID Provider.
2. The OP authenticates the End-User and obtains appropriate authorization.
3. The OP responds with an Access Token and an ID Token.
4. The RP can send a request with the Access Token to the UserInfo Endpoint, per **Section 4.3**.
5. The UserInfo Endpoint returns Claims about the End-User.

---

2. Authentication

TOC

Authentication is performed to log in the End-User or to determine that the End-User is already logged in. Authentication Requests can follow one of three paths: the Authorization Code Flow, the Implicit Flow, or the Hybrid Flow. The Authorization Code Flow is suitable for Clients that can securely maintain a Client Secret between themselves and the Authorization Server whereas, the Implicit Flow is suitable for Clients that cannot. The Hybrid Flow combines aspects of the Authorization Code Flow and the Implicit Flow. The flows determine how the ID Token and Access Token are returned to the Client. The flow used is determined by the `response_type` value contained in the Authorization Request.

2.1. Authentication using the Authorization Code Flow

TOC

This section describes how to perform authentication using the Authorization Code Flow. When using the Authorization Code Flow, all tokens are returned from the Token Endpoint.

The Authorization Code Flow returns an Authorization Code to the Client, which can then

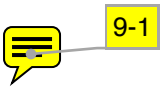
exchange it for an Access Token directly. This provides the benefit of not exposing the Access Token to the Resource Owner and possibly other malicious applications with access to the Resource Owner's User-Agent. The Authorization Server can also authenticate the Client before exchanging the Authorization Code for an Access Token. The Authorization Code flow is suitable for Clients that can securely maintain a Client Secret between themselves and the Authorization Server.

2.1.1. Authorization Code Flow Steps

TOC

The Authorization Code Flow goes through the following steps.

1. Client prepares an Authorization Request containing the desired request parameters.
2. Client sends a request to the Authorization Server.
3. Authorization Server Authenticates the End-User.
4. Authorization Server obtains End-User Consent/Authorization.
5. Authorization Server sends the End-User back to the Client with an Authorization Code.
6. Client requests a response using the Authorization Code at the Token Endpoint.
7. Client receives a response that contains an ID Token and Access Token in the response body.
8. Client validates the tokens and retrieves the End-User's subject identifier.



2.1.2. Authorization Endpoint

TOC

The Authorization Endpoint performs authentication of the End-User and requests authorization from the End-User to release information to an OpenID Connect Relying Party (Client). When an End-User accesses a Relying Party application that requires the End-User's identity and other information, it sends the User-Agent to the Authorization Server's Authorization Endpoint for authentication and authorization.

2.1.2.1. Authorization Request

TOC

When the Client wishes to access a Protected Resource and the End-User Authorization has not yet been obtained, the Client prepares an Authorization Request to the Authorization Endpoint.

An Authorization Request is a message sent from an RP to the OP's Authorization Endpoint. It is an extended **OAuth 2.0** [RFC6749] Authorization Request. Section 4.1.1 and 4.2.1 of **OAuth 2.0** [RFC6749] define the OAuth 2.0 Authorization Request parameters.

Communication with the Authorization Endpoint MUST utilize TLS. See **Section 15.17** for more information on using TLS.

Authorization Servers MUST support the use of the HTTP `GET` and `POST` methods defined

in **RFC 2616** [RFC2616] at the Authorization Endpoint.

Clients MAY use the HTTP `GET` or `POST` methods to send the Authorization Request to the Authorization Server. If using the HTTP `GET` method, the request parameters are serialized using URI Query String Serialization, per **Section 12.1**. If using the HTTP `POST` method, the request parameters are serialized using Form Serialization, per **Section 12.2**.

OpenID Connect uses the following OAuth 2.0 request parameters with the Authorization Code Flow:

scope
: REQUIRED. OAuth 2.0 scope values. OpenID Connect requests MUST contain the `openid` scope value. Other scope values MAY be present. See Sections **4.1** and **10** for additional scope values defined by this specification.

response_type
: REQUIRED. OAuth 2.0 registered response type value that determines how the Authorization Response is returned to the Client. When using the Authorization Code Flow, this value MUST be `code`.

client_id
: REQUIRED. OAuth 2.0 Client Identifier.

redirect_uri
: REQUIRED. Redirection URI to which the response will be sent. This MUST be pre-registered with the OpenID Provider. This URI MUST exactly match one of the `redirect_uris` registered for the Client, with the matching performed as described in Section 6.2.1 of **[RFC3986]** (Simple String Comparison). When using this flow, the redirection URI MAY use the `http` scheme, provided that the Client Type is `confidential`, as defined in Section 2.1 of OAuth 2.0; otherwise, it MUST use the `https` scheme.

10-2

state
: RECOMMENDED. Opaque value used to maintain state between the request and the callback. Typically, Cross-Site Request Forgery (CSRF, XSRF) mitigation is done by cryptographically binding the value of this parameter with the browser cookie.

10-1

This specification also defines the following request parameters for use with the Authorization Code Flow:

nonce
: OPTIONAL. String value used to associate a Client session with an ID Token, and to mitigate replay attacks. The value is passed through unmodified from the Authorization Request to the ID Token. Sufficient entropy MUST be present in the `nonce` values used to prevent attackers from guessing values. One method to achieve this is to store a random value as a signed session cookie, and pass the value in the `nonce` parameter. In that case, the `nonce` in the returned ID Token can be compared to the signed session cookie to detect ID Token replay by third parties.

10-3

display
: OPTIONAL. ASCII string value that specifies how the Authorization Server displays the authentication and consent user interface pages to the End-User. The defined values are:

    page
    : The Authorization Server SHOULD display authentication and

consent UI consistent with a full User-Agent page view. If the display parameter is not specified this is the default display mode.

popup
: The Authorization Server SHOULD display authentication and consent UI consistent with a popup User-Agent window. The popup User-Agent window SHOULD be 450 pixels wide and 500 pixels tall.

touch
: The Authorization Server SHOULD display authentication and consent UI consistent with a device that leverages a touch interface. The Authorization Server MAY attempt to detect the touch device and further customize the interface.

wap
: The Authorization Server SHOULD display authentication and consent UI consistent with a "feature phone" type display.

prompt
: OPTIONAL. Space delimited, case sensitive list of ASCII string values that specifies whether the Authorization Server prompts the End-User for reauthentication and consent. The defined values are:

none
: The Authorization Server MUST NOT display any authentication or consent user interface pages. A `login_required` error is returned if the End-User is not already authenticated or the Client does not have pre-configured consent for the requested Claims or does not fulfill other conditions for processing. This can be used as a method to check for existing authentication and/or consent.

login
: The Authorization Server SHOULD prompt the End-User for reauthentication. If it cannot prompt the End-User, it MUST return an error.

consent
: The Authorization Server SHOULD prompt the End-User for consent before returning information to the Client.

select_account
: The Authorization Server SHOULD prompt the End-User to select a user account. This enables an End-User who has multiple accounts at the Authorization Server to select amongst the multiple accounts that they might have current sessions for. If it cannot prompt the End-User, it MUST return an error.

The `prompt` parameter can be used by the Client to make sure that the End-User is still present for the current session or to bring attention to the request. If this parameter contains `none` with any other value, an error is returned.

max_age
: OPTIONAL. Maximum Authentication Age. Specifies the allowable elapsed time in seconds since the last time the End-User was actively authenticated. If the elapsed time is greater than this value, the OP MUST attempt to actively

re-authenticate the End-User. (The `max_age` request parameter corresponds to the OpenID 2.0 **PAPE** [OpenID.PAPE] `max_auth_age` request parameter.) When `max_age` is used, the ID Token returned MUST include an `auth_time` Claim Value.

ui_locales
: OPTIONAL. End-User's preferred languages and scripts for the user interface, represented as a space-separated list of **BCP47** [RFC5646] language tag values, ordered by preference. For instance, the value "fr-CA fr en" represents a preference for French as spoken in Canada, then French (without a region designation), followed by English (without a region designation). An error SHOULD NOT result if some or all of the requested locales are not supported by the OpenID Provider.

id_token_hint
: OPTIONAL. Previously issued ID Token passed to the Authorization Server as a hint about the End-User's current or past authenticated session with the Client. If the End-User identified by the ID Token is logged in or is logged in by the request, then the Authorization Server returns a positive response; otherwise, it SHOULD return a `login_required` error. When possible, an `id_token_hint` SHOULD be present when `prompt=none` is used and an `invalid_request` error MAY be returned if it is not; however, the server SHOULD respond successfully when possible, even if it is not present. The Authorization Server need not be listed as an audience of the ID Token when it is used as an `id_token_hint` value.
  If the ID Token received by the RP is encrypted, the Client MUST decrypt the signed ID Token contained within the encrypted ID Token. The Client MAY re-encrypt the signed ID token to the Authentication Server using a key that enables the server to decrypt the ID Token.

12-1

login_hint
: OPTIONAL. Hint to the Authorization Server about the login identifier the End-User might use to log in (if necessary). This hint can be used by an RP if it first asks the End-User for their e-mail address (or other identifier) and then wants to pass that value as a hint to the discovered authorization service. It is RECOMMENDED that the hint value match the value used for discovery. This value MAY also be a phone number in the format specified for the `phone_number` Claim. The use of this parameter is left to the OP's discretion.

acr_values
: OPTIONAL. Requested Authentication Context Class Reference values. Space-separated string that specifies the `acr` values that the Authorization Server is being requested to use for processing this Authentication Request, with the values appearing in order of preference. The Authentication Context Class satisfied by the authentication performed is returned as the `acr` Claim Value, as specified in **Section 2.1.3.6**. The `acr` Claim is requested as a Voluntary Claim by this parameter.

Other parameters MAY be sent. See Sections **2.2.2**, **2.3.2**, **4.4**, **4.5**, **5**, and **6.2.1** for additional Authorization Request parameters and parameter values defined by this specification.

The following is a non-normative example request using this flow (with line wraps within values for display purposes only):

```
GET /authorize?
```

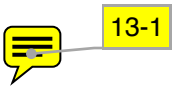
```
  response_type=code
  &scope=openid%20profile%20email
  &client_id=s6BhdRkqt3
  &state=af0ifjsldkj
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb HTTP/1.1
Host: server.example.com
```

2.1.2.2. Authorization Request Validation

TOC

The Authorization Server MUST validate the request received as follows:

1. The Authorization Server MUST validate all the OAuth 2.0 parameters according to the OAuth 2.0 specification.
2. The Authorization Server MUST verify that all the REQUIRED parameters are present.
3. If the `sub` (subject) Claim is requested with a specific value for the ID Token, the Authorization Server MUST only send a positive response if that user has an active session with the Authorization Server. The Authorization Server MUST NOT reply with an ID Token or Access Token for a different user, even if they have an active session with the Authorization Server. Such a request can be made either using an `id_token_hint` parameter or by requesting a specific Claim value as described in **Section 4.5.1**, if the `claims` parameter is supported by the implementation.



As specified in **OAuth 2.0** [RFC6749], Authorization Servers SHOULD ignore unrecognized request parameters.

If the Authorization Server encounters any error, it MUST return an error response.

2.1.2.3. Authorization Server Authenticates End-User

TOC

The Authorization Server validates the request to ensure all REQUIRED parameters are present and all parameters are valid. If the request is valid, the Authorization Server attempts to log in the End-User or determines whether he is logged in, depending upon the request parameter values used. The methods used by the Authorization Server to log in the End-User (e.g. username and password, session cookies, etc.) are beyond the scope of this specification. An authentication user interface MAY be displayed by the Authorization Server, depending upon the request parameter values used and the authentication methods used.

The Authorization Server MUST attempt to log in the End-User in the following cases:

- The End-User is not already logged in.
- The Authorization Request contains the `prompt` parameter with the value `login`. In this case, the Authorization Server MUST reauthenticate the End-User even if the End-User is already authenticated.

The Authorization Server MUST NOT interact with the End-User in the following case:

- The Authorization Request contains the `prompt` parameter with the value

`none`. In this case, the Authorization Server MUST return a `login_required` error if the End-User is not already logged in or could not be silently logged in.

The Authorization Server MUST employ appropriate measures against Cross-Site Request Forgery and Clickjacking as, described in Sections 10.12 and 10.13 of **OAuth 2.0** [RFC6749].

2.1.2.4. Authorization Server Obtains End-User Consent/Authorization

TOC

Once the End-User is authenticated, the Authorization Server MUST obtain an authorization decision. When permitted by the request parameters used, this MAY be done through an interactive dialogue with the End-User that makes it clear what is being consented to or by establishing consent via conditions for processing or other means (for example, via previous administrative consent).

The Authorization Server MUST employ countermeasures against Cross-Site Request Forgery and Clickjacking when interacting with the End-User.

2.1.2.5. Authorization Successful Response

TOC

Once the authorization is determined, the Authorization Server returns a successful or error response. This section describes the successful response. **Section 2.1.2.6** describes the error response.

An Authorization Response is a message returned from the OP's Authorization Endpoint in response to the Authorization Request by the RP. Response parameters are returned to the Client by adding them to the `redirect_uri` specified in the Authorization Request using the "application/x-www-form-urlencoded" format.

This specification only describes **OAuth 2.0 Bearer Token Usage** [RFC6750]. The OAuth 2.0 response parameter `token_type` MUST be set to `Bearer` unless another Token Type has been negotiated with the Client.

When using the Authorization Code Flow, the Authorization Response MUST return the parameters defined in Section 4.1.2 of **OAuth 2.0** [RFC6749].

The following is a non-normative example successful response using this flow (with line wraps within values for display purposes only):

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb?
  code=SplxlOBeZQQYbYS6WxSbIA
  &state=af0ifjsldkj
```

2.1.2.6. Authorization Error Response

TOC

If the End-User denies the request or the End-User authentication fails, the OP (Authorization Server) informs the RP (Client) by using the Error Response parameters defined in Sections 4.1.2.1 of **OAuth 2.0** [RFC6749].

The Authorization Server returns the Client to the redirection URI specified in the Authorization Request with the appropriate error parameters. No other parameters SHOULD be returned.

In addition to the error codes defined in Sections 4.1.2.1 and 4.2.2.1 of OAuth 2.0, this specification also defines the following error codes:

interaction_required
: The Authorization Server requires End-User interaction of some form to proceed. This error MAY be returned when the `prompt` parameter in the Authorization Request is set to `none` to request that the Authorization Server MUST NOT display any user interfaces to the End-User, but the Authorization Request cannot be completed without displaying a user interface for End-User interaction.

login_required
: The Authorization Server requires End-User authentication. This error MAY be returned when the `prompt` parameter in the Authorization Request is set to `none` to request that the Authorization Server MUST NOT display any user interfaces to the End-User, but the Authorization Request cannot be completed without displaying a user interface for user authentication.

session_selection_required
: The End-User is REQUIRED to select a session at the Authorization Server. The End-User MAY be authenticated at the Authorization Server with different associated accounts, but the End-User did not select a session. This error MAY be returned when the `prompt` parameter in the Authorization Request is set to `none` to request that the Authorization Server MUST NOT display any user interfaces to the End-User, but the Authorization Request cannot be completed without displaying a user interface to prompt for a session to use.

consent_required
: The Authorization Server requires End-User consent. This error MAY be returned when the `prompt` parameter in the Authorization Request is set to `none` to request that the Authorization Server MUST NOT display any user interfaces to the End-User, but the Authorization Request cannot be completed without displaying a user interface for End-User consent.

invalid_request_uri
: The `request_uri` in the Authorization Request returns an error or contains invalid data.

invalid_request_object
: The `request` parameter contains an invalid Request Object.

request_not_supported
: The OP does not support use of the `request` parameter.

request_uri_not_supported
: The OP does not support use of the `request_uri` parameter.

registration_not_supported
: The OP does not support use of the `registration` parameter.

The error response parameters are the following:

error

REQUIRED. Error code.

error_description
: OPTIONAL. Human-readable ASCII encoded text description of the error.

error_uri
: OPTIONAL. URI of a web page that includes additional information about the error.

state
: OAuth 2.0 state value. REQUIRED if the Authorization Request included the `state` parameter. Set to the value received from the Client.

When using the Authorization Code Flow, the response parameters are added to the query component of the redirection URI.

The following is a non-normative example error response using this flow (with line wraps within values for display purposes only):

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb?
  error=invalid_request
  &error_description=
    the%20request%20is%20not%20valid%20or%20malformed
  &state=af0ifjsldkj
```

2.1.2.7. Authorization Response Validation

TOC

When using the Authorization Code Flow, the Client MUST validate the response according to RFC 6749, especially Sections 4.1.2 and 10.12.

2.1.3. Token Endpoint

TOC

When using the Authorization Code Flow, the RP (Client) sends a Token Request to the Token Endpoint to obtain a Token Response, which includes an ID Token and an Access Token and MAY include a Refresh Token and other results.

Clients MUST use the HTTP `POST` method to make requests to the Token Endpoint. Request parameters are added using Form Serialization, per **Section 12.2**. The Token Endpoint MUST support the use of the HTTP `POST` method defined in **RFC 2616** [RFC2616] at the Token Endpoint.

Communication with the Token Endpoint MUST utilize TLS. See **Section 15.17** for more information on using TLS.

All Token Endpoint responses that contain tokens, secrets, or other sensitive information MUST include the following HTTP response header fields and values:

Header Name	Header Value
Cache-Control	no-store

Pragma	no-cache

HTTP Response Headers and Values

Clients MAY provide authentication parameters in the request to the Token Endpoint, as described in **Section 8**.

2.1.3.1. Token Request

TOC

A Client using the Authorization Code Flow obtains an ID Token and an Access Token by authenticating with the Authorization Server and presenting its Authorization Grant (in the form of an Authorization Code) to the Token Endpoint.

To obtain an ID Token, Access Token, or Refresh Token, the Client MUST authenticate to the Token Endpoint using the authentication method registered for its `client_id`, as described in **Section 8**. The Client sends the parameters via HTTPS `POST` to the Token Endpoint using Form Serialization, per **Section 12.2**, as described in Section 4.1.3 of **OAuth 2.0** [RFC6749].

The following is a non-normative example of a Token Request (with line wraps within values for display purposes only):

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

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
```

2.1.3.2. Token Request Validation

TOC

Upon receipt of the request, the Authorization Server MUST:

- Authenticate any Clients that were issued Client Credentials (or for which other Client Authentication methods can be used),
- Ensure the Authorization Code was issued to the authenticated Client,
- Verify that the Authorization Code is valid, and
- Ensure that the `redirect_uri` parameter value is identical to the `redirect_uri` parameter value that was included in the initial Authorization Request. If the `redirect_uri` parameter value is not present when there is only one registered `redirect_uri` value, the Authorization Server MAY return an error (since the Client should have included the parameter) or MAY proceed without an error (since OAuth 2.0 permits the parameter to be omitted in this case).

2.1.3.3. Token Successful Response

TOC

Upon receipt of the Token Request, the Authorization Server MUST return either a successful response or an error response that corresponds to the received Authorization Code.

After receiving and validating a valid and authorized Token Request from the Client, the Authorization Server returns a successful response that includes an ID Token and an Access Token. The parameters in the successful response are defined in Section 4.1.4 of **OAuth 2.0** [RFC6749].

This specification only describes **OAuth 2.0 Bearer Token Usage** [RFC6750]. The OAuth 2.0 response parameter `token_type` MUST be set to `Bearer` unless another Token Type has been negotiated with the Client. Servers SHOULD support **OAuth 2.0 Bearer Token Usage** [RFC6750] for interoperability. For security reasons, Servers MAY only allow Clients to register specific `token_type` values. Clients MUST support **OAuth 2.0 Bearer Token Usage** [RFC6750] and MAY support other `token_type`.

In addition to the OAuth 2.0 response parameters, the following parameters MUST be included in the response if the `grant_type` value is `authorization_code` and the Authorization Request `scope` parameter contains `openid`:

id_token
: ID Token value associated with the authenticated session.

An `id_token` MUST be returned when the `grant_type` value is `authorization_code` and MAY be returned when other grant types are used.

A successful response uses the `application/json` media type.

The following is a non-normative example of a successful Token Response:

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

{
 "access_token": "SlAV32hkKG",
 "token_type": "Bearer",
 "refresh_token": "8xLOxBtZp8",
 "expires_in": 3600,
 "id_token": "eyJhbGciOiJSUzI1NiJ9.ew0KICAgICJpc3MiOiAiaHR0cDovL
   3NlcnZlci5leGFtcGxlLmNvbSIsDQogICAgInVzZXJfaWQiOiAiMjQ4Mjg5NzYxM
   DAxIiwNCiAgICAiYXVkIjogInM2QmhkUmtxdDMiLA0KICAgICJub25jZSI6ICJuL
   TBTNl9XekEyTWoiLA0KICAgICJleHAiOiAxMzExMjgxOTcwLA0KICAgICJpYXQiO
   iAxMzExMjgwOTcwDQp9.lsQI_KNHpl58YY24G9tUHXr3Yp7OKYnEaVpRL0KI4szT
   D6GXpZcgxIpkOCcajyDiIv62R9rBWASV191Akk1BM36gUMm8H5s8xyxNdRfBViCa
   xTqHA7X_vV3U-tSWl6McR5qaSJaNQBpg1oGPjZdPG7zWCG-yEJC4-Fbx2FPOS7-h
   5V0k33O5Okd-OoDUKoFPMd6ur5cIwsNyBazcsHdFHqWlCby5nl_HZdW-PHq0gjzy
   JydB5eYIvOfOHYBRVML9fKwdOLM2xVxJsPwvy3BqlVKc593p2WwItIg52ILWrc6A
   tqkqHxKsAXLVyAoVInYkl_NDBkCqYe2KgNJFzfEC8g"
}
```

As specified in **OAuth 2.0** [RFC6749], Clients SHOULD ignore unrecognized response parameters.

2.1.3.4. Token Error Response

TOC

If the Token Request is invalid or unauthorized, the Authorization Server constructs the error response. The parameters of the Token Error Response are defined as in Section 5.2 of **OAuth 2.0** [RFC6749]. The HTTP response body uses the `application/json` media type with HTTP response code of 400.

The following is a non-normative example Token Error Response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
 "error": "invalid_request"
}
```

2.1.3.5. Token Response Validation

TOC

The Client MUST validate the Token Response as follows:

1. Follow the validation rules in RFC 6749, especially those in Sections 5.1 and 10.12.
2. Follow the ID Token validation rules in **Section 2.1.3.7**.
3. Follow the Access Token validation rules in **Section 2.1.3.8**.

2.1.3.6. ID Token

TOC

The ID Token is a security token that contains Claims about the authentication event and other requested Claims. The ID Token is represented as a **JSON Web Token (JWT)** [JWT].

The ID Token is used to manage the authentication event and user identifier and is scoped to a particular Client via the `aud` (audience) and `nonce` Claims.

The following Claims are used within the ID Token when using this flow:

iss
: REQUIRED. Issuer Identifier for the Issuer of the response. The `iss` value is a case sensitive URL using the `https` scheme that contains scheme, host, and OPTIONALLY, port number and path components and no query or fragment components.

sub

REQUIRED. Subject identifier. A locally unique and never reassigned identifier within the Issuer for the End-User, which is intended to be consumed by the Client, e.g., `24400320` or `AItOawmwtWwcT0k51BayewNvutrJUqsvl6qs7A4`. It MUST NOT exceed 255 ASCII characters in length. The `sub` value is a case sensitive string.

aud

REQUIRED. Audience(s) that this ID Token is intended for. It MUST contain the OAuth 2.0 `client_id` of the Relying Party as an audience value. It MAY also contain identifiers for other audiences. In the general case, the `aud` value is an array of case sensitive strings. In the special case when there is one audience, the `aud` value MAY be a single case sensitive string.

exp

REQUIRED. Expiration time on or after which the ID Token MUST NOT be accepted for processing. The processing of this parameter requires that the current date/time MUST be before the expiration date/time listed in the value. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. The time is represented as the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time. See **RFC 3339** [RFC3339] for details regarding date/times in general and UTC in particular. The `exp` value is a number.

iat

REQUIRED. Time at which the JWT was issued. The time is represented as the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time. The `iat` value is a number.

auth_time

OPTIONAL or REQUIRED. Time when the End-User authentication occurred. The time is represented as the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time. When a `max_age` request is made or when `auth_time` is requested as an Essential Claim, then this Claim is REQUIRED. (The `auth_time` Claim semantically corresponds to the OpenID 2.0 **PAPE** [OpenID.PAPE] `auth_time` response parameter.) The `auth_time` value is a number.

nonce

OPTIONAL or REQUIRED. String value used to associate a Client session with an ID Token, and to mitigate replay attacks. The value is passed through unmodified from the Authorization Request to the ID Token. If present in the ID Token, Clients MUST verify that the `nonce` Claim Value is equal to the value of the `nonce` parameter sent in the Authorization Request. If present in the Authorization Request, Authorization Servers MUST include a `nonce` Claim in the ID Token with the Claim Value being the nonce value sent in the Authorization Request. Authorization Servers SHOULD perform no other processing on `nonce` values used. Use of the nonce is REQUIRED for all requests where an ID Token is returned directly from the Authorization Endpoint. It is OPTIONAL when the ID Token is returned from the Token Endpoint. The `nonce` value is a case sensitive string.

at_hash

OPTIONAL. 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` parameter of the ID Token's **JWS** [JWS] 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. The `at_hash` value is a case

sensitive string.

acr

OPTIONAL. Authentication Context Class Reference. String specifying an Authentication Context Class Reference value that identifies the Authentication Context Class that the authentication performed satisfied. The value "0" indicates the End-User authentication did not meet the requirements of **ISO/IEC 29115** [ISO29115] level 1. Authentication using a long-lived browser cookie, for instance, is one example where the use of "level 0" is appropriate. Authentications with level 0 SHOULD never be used to authorize access to any resource of any monetary value. (This corresponds to the OpenID 2.0 **PAPE** [OpenID.PAPE] `nist_auth_level` 0.) An absolute URI or a **registered name** [RFC6711] SHOULD be used as the `acr` value; registered names MUST NOT be used with a different meaning than that which is registered. Parties using this claim will need to agree upon the meanings of the values used, which may be context-specific. The `acr` value is a case sensitive string.

amr

OPTIONAL. Authentication Methods References. JSON array of strings that are identifiers for authentication methods used in the authentication. For instance, values might indicate that both password and OTP authentication methods were used. The definition of particular values to be used in the `amr` Claim is beyond the scope of this specification. Parties using this claim will need to agree upon the meanings of the values used, which may be context-specific. The `amr` value is an array of case sensitive strings.

azp

OPTIONAL or REQUIRED. Authorized Party - the party to which the ID Token was issued. If present, it MUST contain the OAuth 2.0 `client_id` of this party. This Claim is only REQUIRED when the ID Token has a single audience value and that audience is different than the Authorized Party. It MAY be included even when the Authorized Party is the same as the sole audience. The `azp` value is a case sensitive string containing a StringOrURI value.

ID Tokens MAY contain other Claims. Any Claims used that are not understood MUST be ignored. See Sections **2.3.2.11**, **4.2**, and **6.4** for additional Claims defined by this specification.

ID Tokens MUST be signed using **JWS** [JWS] and OPTIONALLY both signed and then encrypted using **JWS** [JWS] and **JWE** [JWE] respectively, thereby providing authentication, integrity, non-repudiation, and optionally, confidentiality, per **Section 15.14**. ID Tokens MUST NOT use `none` as the `alg` value unless the Authorization Request used the Authorization Code Flow and the Client explicitly requested the use of `none` at registration time.

ID Tokens SHOULD NOT use the JWS or JWE `x5u`, `x5c`, `jku`, or `jwk` header parameter fields. Instead, key values and key references used for ID Tokens are communicated in advance using Discovery and Registration parameters.

The following is a non-normative example of an ID Token claims set:

```
{
 "iss": "https://server.example.com",
 "sub": "24400320",
```

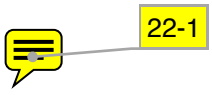
```
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "auth_time": 1311280969,
 "acr": "urn:mace:incommon:iap:silver",
 "at_hash": "MTIzNDU2Nzg5MDEyMzQ1Ng"
}
```

2.1.3.7. ID Token Validation

TOC

Clients MUST validate the ID Token in the Token Response, in the following manner:

1. If the Client has provided an `id_token_encrypted_response_alg` parameter during Registration, decrypt the ID Token using the key pair specified during Registration.
2. The Client MUST validate that the `aud` (audience) Claim contains its `client_id` value registered at the Issuer identified by the `iss` (issuer) Claim as an audience. The `aud` (audience) Claim MAY contain an array with more than one element. The ID Token MUST be rejected if the ID Token does not list the Client as a valid audience, or if it contains additional audiences not trusted by the Client.
3. If the ID Token contains multiple audiences, the Client SHOULD verify that an `azp` Claim is present.
4. If an `azp` (authorized party) Claim is present, the Client SHOULD verify and that its `client_id` is the Claim value.
5. If the `id_token` is received via direct communication between the Client and the Token Endpoint, the TLS server validation MAY be used to validate the issuer in place of checking the token signature. The Client MUST validate the signature of all other ID Tokens according to **JWS** [JWS] using the algorithm specified in the `alg` parameter of the JWT header.
6. The `alg` value SHOULD be the default of `RS256` or the algorithm sent by the Client in the `id_token_signed_response_alg` parameter during Registration.
7. If the `alg` parameter of the JWT header is a MAC based algorithm such as `HS256`, `HS384`, or `HS512`, the octets of the UTF-8 representation of the `client_secret` corresponding to the `client_id` contained in the `aud` (audience) Claim are used as the key to validate the signature. Multiple audiences are not supported for MAC based algorithms.
8. For other Signing algorithms, the Client MUST use the signing key provided in Discovery by the Issuer. The issuer MUST exactly match the value of the `iss` (issuer) Claim.
9. The current time MUST be less than the value of the `exp` Claim.
10. The `iat` Claim can be used to reject tokens that were issued too far away from the current time, limiting the amount of time that nonces need to be stored to prevent attacks. The acceptable range is Client specific.
11. If a nonce value was sent in the Authorization Request, a `nonce` Claim MUST be present and its value checked to verify that it is the same value as the one that was sent in the Authorization Request. The Client SHOULD check the `nonce` value for replay attacks. The precise method for detecting replay attacks is Client specific.
12. If the `acr` Claim was requested, the Client SHOULD check that the asserted



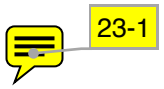
Claim Value is appropriate. The meaning and processing of `acr` Claim Values is out of scope for this specification.

13. If the `auth_time` Claim was requested, either through a specific request for this Claim or by using the `max_age` parameter, the Client SHOULD check the `auth_time` Claim value and request re-authentication if it determines too much time has elapsed since the last End-User authentication.

2.1.3.8. Access Token Validation

TOC

When using the Authorization Code Flow, if the ID Token contains an `at_hash` Claim, the Client MAY use it to validate the Access Token the same manner as for the Implicit Flow, as defined in **Section 2.2.2.9**.



2.2. Authentication using the Implicit Flow

TOC

This section describes how to perform authentication using the Implicit Flow. When using the Implicit Flow, all tokens are returned from the Authorization Endpoint; the Token Endpoint is not used.

The Implicit Flow is mainly used by Clients implemented in a browser using a scripting language. The Access Token and ID Token are returned directly to the Client, which may expose them to the Resource Owner and other applications that have access to the Resource Owner's User-Agent. The Authorization Server does not perform Client Authentication before issuing the Access Token.

2.2.1. Implicit Flow Steps

TOC

The Implicit Flow follows the following steps:

1. Client prepares an Authorization Request containing the desired request parameters.
2. Client sends a request to the Authorization Server.
3. Authorization Server Authenticates the End-User.
4. Authorization Server obtains End-User Consent/Authorization.
5. Authorization Server sends the End-User back to the Client with an ID Token and, if requested, an Access Token.
6. Client validates the tokens and retrieves the End-User's subject identifier.

2.2.2. Authorization Endpoint

TOC

When using the Implicit Flow, the Authorization Endpoint is used in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2**, with the exception of the differences specified in this section.

2.2.2.1. Authorization Request

TOC

When using the Implicit Flow, Authorization Requests are made in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.1**, with the exception of the differences specified in this section.

When using the Implicit Flow, the same Authorization Request parameters are used as for the Code Flow, as defined in **Section 2.1.2**, with the exception of the differences specified in this section.

24-1

response_type
: REQUIRED. OAuth 2.0 registered response type value that determines how the Authorization Response is returned to the Client. When using the Implicit Flow, this value MUST be `id_token token` or `id_token`. The meanings of both of these values are defined in **OAuth 2.0 Multiple Response Type Encoding Practices** [OAuth.Responses]. No Access Token is returned when the value is `id_token`.
(Note that while OAuth 2.0 also defines the `token` response type value for the Implicit Flow, OpenID Connect does not use this response type, since no ID Token is returned.)

redirect_uri
: REQUIRED. Redirection URI to which the response will be sent. This MUST be pre-registered with the OpenID Provider. This URI MUST exactly match one of the `redirect_uris` registered for the Client, with the matching performed as described in Section 6.2.1 of **[RFC3986]** (Simple String Comparison). When using this flow, the redirection URI MUST NOT use the `http` scheme unless the Client is a native application, in which case it MAY use the `http:` scheme with `localhost` as the hostname.

24-2

nonce
: REQUIRED. String value used to associate a Client session with an ID Token, and to mitigate replay attacks. The value is passed through unmodified from the Authorization Request to the ID Token. Sufficient entropy MUST be present in the `nonce` values used to prevent attackers from guessing values. One method to achieve this is to store a random value as a signed session cookie, and pass the value in the `nonce` parameter. In that case, the `nonce` in the returned ID Token can be compared to the signed session cookie to detect ID Token replay by third parties.

24-3

The following is a non-normative example request using the Implicit Flow (with line wraps within values for display purposes only):

```
GET /authorize?
  response_type=id_token%20token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile
  &state=af0ifjsldkj
  &nonce=n-0S6_WzA2Mj HTTP/1.1
Host: server.example.com
```

2.2.2.2. Authorization Request Validation

TOC

When using the Implicit Flow, the Authorization Request is validated in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.2**.

2.2.2.3. Authorization Server Authenticates End-User

TOC

When using the Implicit Flow, End-User Authentication is performed in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.3**.

2.2.2.4. Authorization Server Obtains End-User Consent/Authorization

TOC

When using the Implicit Flow, End-User Consent is obtained in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.4**.

2.2.2.5. Authorization Successful Response

TOC

When using the Implicit Flow, Authorization Responses are made in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.5**, with the exception of the differences specified in this section.

When using the Implicit Flow, all response parameters are added to the fragment component of the redirection URI, as specified in **OAuth 2.0 Multiple Response Type Encoding Practices** [OAuth.Responses]. These parameters are returned:

access_token
: Access Token for the UserInfo Endpoint. This is returned unless the `response_type` value used is `id_token`.

token_type
: OAuth 2.0 Token Type value. The value MUST be `Bearer` or another `token_type` value that the Client has negotiated with the Authorization Server. Clients implementing this profile MUST support the **OAuth 2.0 Bearer Token Usage** [RFC6750] specification. This profile only describes the use of bearer tokens. This is returned in the same cases as `access_token` is.

id_token
: REQUIRED. ID Token.

state
: OAuth 2.0 state value. REQUIRED if the `state` parameter is present in the Client Authorization Request. Clients MUST verify that the `state` value is equal to the value of `state` parameter in the Authorization Request.

expires_in
: OPTIONAL. Expiration time of the Access Token in seconds since the response was generated.

The following is a non-normative example of a successful response using the Implicit Flow (with line wraps for the display purposes only):

```
HTTP/1.1 302 Found
```

```
Location: https://client.example.org/cb#
  access_token=SlAV32hkKG
  &token_type=bearer
  &id_token=eyJ0 ... NiJ9.eyJ1c ... I6IjIifX0.DeWt4Qu ... ZXso
  &expires_in=3600
  &state=af0ifjsldkj
```

2.2.2.6. Authorization Error Response

TOC

When using the Implicit Flow, Authorization Error Responses are made in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.6**, with the exception of the differences specified in this section.

If the End-User denies the request or the End-User authentication fails, the Authorization Server MUST return the error Authorization Response as defined in 4.2.2.1 of **OAuth 2.0** [RFC6749] and **OAuth 2.0 Multiple Response Type Encoding Practices** [OAuth.Responses]. In particular, error parameters are returned in the fragment component of the redirection URI.

2.2.2.7. Redirect URI Fragment Handling

TOC

The Client MUST provide a way for the User-Agent to parse the fragment encoded response and post it to the Web Server Client for validation.

The following is an example of a JavaScript file that a Client might host at its `redirect_uri`. This is loaded by the redirect from the Authorization Server. The fragment component is parsed and then sent by `POST` to a URI that will validate the information received.

Following is a non-normative example of a Redirect URI response:

```
GET /cb HTTP/1.1
Host: client.example.org

HTTP/1.1 200 OK
Content-Type: text/html

<script type="text/javascript">

// First, parse the query string
var params = {}, postBody = location.hash.substring(1),
    regex = /([^&=]+)=([^&]*)/g, m;
while (m = regex.exec(postBody)) {
  params[decodeURIComponent(m[1])] = decodeURIComponent(m[2]);
}

// And send the token over to the server
var req = new XMLHttpRequest();
// using POST so query isn't logged
req.open('POST', 'https://' + window.location.host +
```

```
                  '/catch_response', true);
req.setRequestHeader('Content-Type',
                     'application/x-www-form-urlencoded');

req.onreadystatechange = function (e) {
  if (req.readyState == 4) {
    if (req.status == 200) {
// If the response from the POST is 200 OK, perform a redirect
      window.location = 'https://'
        + window.location.host + '/redirect_after_login'
    }
// if the OAuth response is invalid, generate an error message
    else if (req.status == 400) {
      alert('There was an error processing the token')
    } else {
      alert('Something other than 200 was returned')
    }
  }
};
req.send(postBody);
```

2.2.2.8. Authorization Response Validation

TOC

When using the Implicit Flow, the Client MUST validate the response as follows:

1. Verify that the response conforms to Section 5 of **[OAuth.Responses]**.
2. Follow the validation rules in RFC 6749, especially those in Sections 4.2.2 and 10.12.
3. Follow the ID Token validation rules in **Section 2.2.2.11**.
4. Follow the Access Token validation rules in **Section 2.2.2.9**, unless the `response_type` value used is `id_token`.

2.2.2.9. Access Token Validation

TOC

To validate an Access Token issued from the Authorization Endpoint with an ID Token, the Client SHOULD do the following:

1. Hash the octets of the ASCII representation of the `access_token` with the hash algorithm specified in **JWA** [JWA] for the `alg` parameter in the ID Token's **JWS** [JWS] header.
2. Take the left-most half of the hash and base64url encode it.
3. The value of `at_hash` in the ID Token MUST match the value produced in the previous step if `at_hash` is present in the ID Token.

2.2.2.10. ID Token

TOC

When using the Implicit Flow, the contents of the ID Token are the same as for the Authorization Code Flow, as defined in **Section 2.1.3.6**, with the exception of the

differences specified in this section.

The requirements for using the following Claims are as follows when using the Implicit Flow:

nonce
: REQUIRED. String value used to associate a Client session with an ID Token, and to mitigate replay attacks. The value is passed through unmodified from the Authorization Request to the ID Token. Clients MUST verify that the `nonce` Claim Value is equal to the value of the `nonce` parameter sent in the Authorization Request. Authorization Servers MUST include a `nonce` Claim in the ID Token with the Claim Value being the nonce value sent in the Authorization Request. Use of the nonce is REQUIRED when using the Implicit Flow. The `nonce` value is a case sensitive string.

at_hash
: OPTIONAL or REQUIRED. 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` parameter of the ID Token's **JWS** [JWS] 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. The `at_hash` value is a case sensitive string.
If the ID Token is issued from the Authorization Endpoint with an `access_token`, which is the case with the `response_type` value `id_token token`, this is REQUIRED.

2.2.2.11. ID Token Validation

TOC

When using the Implicit Flow, the contents of the ID Token MUST be validated in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.3.7**, with the exception of the differences specified in this section.

1. The Client MUST validate the signature of the ID Token according to **JWS** [JWS] using the algorithm specified in the `alg` parameter of the JWT header.
2. The value of the `nonce` Claim MUST be checked to verify that it is the same value as the one that was sent in the Authorization Request. The Client SHOULD check the `nonce` value for replay attacks. The precise method for detecting replay attacks is Client specific.

2.3. Authentication using the Hybrid Flow

TOC

This section describes how to perform authentication using the Hybrid Flow. When using the Hybrid Flow, some tokens are returned from the Authorization Endpoint and others are returned from the Token Endpoint. The mechanisms for returning tokens in the Hybrid Flow are specified in **OAuth 2.0 Multiple Response Type Encoding Practices** [OAuth.Responses].

2.3.1. Hybrid Flow Steps

TOC

The Hybrid Flow follows the following steps:

1. Client prepares an Authorization Request containing the desired request parameters.
2. Client sends a request to the Authorization Server.
3. Authorization Server Authenticates the End-User.
4. Authorization Server obtains End-User Consent/Authorization.
5. Authorization Server Sends the End-User back to the Client with an ID Token and, if requested, an Authorization Code and/or Access Token.
6. Client requests a response using the Authorization Code at the Token Endpoint.
7. Client receives a response that contains an ID Token and Access Token in the response body.
8. Client validates the tokens and retrieves the End-User's subject identifier.

2.3.2. Authorization Endpoint

TOC

When using the Hybrid Flow, the Authorization Endpoint is used in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2**, with the exception of the differences specified in this section.

2.3.2.1. Authorization Request

TOC

When using the Hybrid Flow, Authorization Requests are made in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.1**, with the exception of the differences specified in this section.

When using the Hybrid Flow, the same Authorization Request parameters are used as for the Implicit Flow, as defined in **Section 2.2.2**, with the exception of the differences specified in this section.

response_type
: REQUIRED. OAuth 2.0 registered response type value that determines how the Authorization Response is returned to the Client. When using the Hybrid Flow, this value MUST be `code id_token`, `code token`, or `code id_token token`. The meanings of these values are defined in **OAuth 2.0 Multiple Response Type Encoding Practices** [OAuth.Responses]. No Access Token is returned when the value is `id_token`.

29-1

The following is a non-normative example request using the Hybrid Flow (with line wraps within values for display purposes only):

```
GET /authorize?
  response_type=code%20id_token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile%20email
```

```
    &nonce=n-0S6_WzA2Mj
    &state=af0ifjsldkj HTTP/1.1
Host: server.example.com
```

2.3.2.2. Authorization Request Validation

TOC

When using the Hybrid Flow, the Authorization Request is validated in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.2**.

2.3.2.3. Authorization Server Authenticates End-User

TOC

When using the Hybrid Flow, End-User Authentication is performed in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.3**.

2.3.2.4. Authorization Server Obtains End-User Consent/Authorization

TOC

When using the Hybrid Flow, End-User Consent is obtained in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.4**.

2.3.2.5. Authorization Successful Response

TOC

When using the Hybrid Flow, Authorization Responses are made in the same manner as for the Implicit Flow, as defined in **Section 2.2.2.5**, with the exception of the differences specified in this section.

access_token
: Access Token for the UserInfo Endpoint. This is returned when the `response_type` value used is `code token`, or `code id_token token`. (A `token_type` value is also returned in the same cases.)

id_token
: ID Token. This is returned when the `response_type` value used is `code id_token` or `code id_token token`.

The following is a non-normative example of a successful response using the Hybrid Flow (with line wraps for the display purposes only):

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  code=SplxlOBeZQQYbYS6WxSbIA
  &id_token=eyJ0 ... NiJ9.eyJ1c ... I6IjIifX0.DeWt4Qu ... ZXso
  &state=af0ifjsldkj
```

TOC

2.3.2.6. Authorization Error Response

When using the Hybrid Flow, Authorization Error Responses are made in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.2.6**, with the exception of the differences specified in this section.

If the End-User denies the request or the End-User authentication fails, the Authorization Server MUST return the error Authorization Response as defined in 4.2.2.1 of **OAuth 2.0** [RFC6749] and **OAuth 2.0 Multiple Response Type Encoding Practices** [OAuth.Responses]. In particular, error parameters are returned in the fragment component of the redirection URI.

2.3.2.7. Redirect URI Fragment Handling

TOC

When using the Hybrid Flow, Redirect URI handling is done in the same manner as for the Implicit Flow, as defined in **Section 2.2.2.7**, with the exception of the differences specified in this section.

2.3.2.8. Authorization Response Validation

TOC

When using the Hybrid Flow, the Client MUST validate the response as follows:

1. Verify that the response conforms to Section 5 of **[OAuth.Responses]**.
2. Follow the validation rules in RFC 6749, especially those in Sections 4.2.2 and 10.12.
3. Follow the ID Token validation rules in **Section 2.3.2.12** when the `response_type` value used is `code id_token` or `code id_token token`.
4. Follow the Access Token validation rules in **Section 2.3.2.9** when the `response_type` value used is `code id_token token`.
5. Follow the Authorization Code validation rules in **Section 2.3.2.10** when the `response_type` value used is `code id_token` or `code id_token token`.

2.3.2.9. Access Token Validation

TOC

When using the Hybrid Flow, Access Tokens returned from the Authorization Endpoint are validated in the same manner as for the Implicit Flow, as defined in **Section 2.2.2.9**.

2.3.2.10. Code Validation

TOC

To validate an Authorization Code issued from the Authorization Endpoint with an ID Token, the Client SHOULD do the following:

1. Hash the octets of the ASCII representation of the `code` with the hash algorithm specified in **JWA** [JWA] for the `alg` parameter in the ID Token's **JWS** [JWS] header.

2. Take the left-most half of the hash and base64url encode it.
3. The value of `c_hash` in the ID Token MUST match the value produced in the previous step if `c_hash` is present in the ID Token.

2.3.2.11. ID Token

TOC

When using the Hybrid Flow, the contents of the ID Token returned from the Authorization Endpoint are the same as for the Implicit Flow, as defined in **Section 2.2.2.10**, with the exception of the differences specified in this section.

The requirements for using the following Claims are as follows when using the Hybrid Flow:

nonce
: REQUIRED. String value used to associate a Client session with an ID Token, and to mitigate replay attacks. The value is passed through unmodified from the Authorization Request to the ID Token. Clients MUST verify that the `nonce` Claim Value is equal to the value of the `nonce` parameter sent in the Authorization Request. Authorization Servers MUST include a `nonce` Claim in the ID Token with the Claim Value being the nonce value sent in the Authorization Request. Use of the nonce is REQUIRED when using the Hybrid Flow. The `nonce` value is a case sensitive string.

at_hash
: OPTIONAL or REQUIRED. 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` parameter of the ID Token's **JWS** [JWS] 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. The `at_hash` value is a case sensitive string.
If the ID Token is issued from the Authorization Endpoint with an `access_token`, which is the case with the `response_type` value `code id_token token`, this is REQUIRED.

32-1

c_hash
: OPTIONAL or REQUIRED. Code 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 `code` value, where the hash algorithm used is the hash algorithm used in the `alg` parameter of the ID Token's **JWS** [JWS] header. For instance, if the `alg` is `HS512`, hash the `code` value with SHA-512, then take the left-most 256 bits and base64url encode them. The `c_hash` value is a case sensitive string.
If the ID Token is issued from the Authorization Endpoint with a `code`, which is the case with the `response_type` values `code id_token` and `code id_token token`, this is REQUIRED.

2.3.2.12. ID Token Validation

TOC

When using the Hybrid Flow, the contents of the ID Token returned from the Authorization Endpoint MUST be validated in the same manner as for the Implicit Flow, as defined in **Section 2.2.2.11**.

2.3.3. Token Endpoint

TOC

When using the Hybrid Flow, the Token Endpoint is used in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.3**, with the exception of the differences specified in this section.

2.3.3.1. Token Request

TOC

When using the Hybrid Flow, Token Requests are made in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.3.1**.

2.3.3.2. Token Request Validation

TOC

When using the Hybrid Flow, Token Requests are validated in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.3.2**.

2.3.3.3. Token Successful Response

TOC

When using the Hybrid Flow, Token Responses are made in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.3.3**.

2.3.3.4. Token Error Response

TOC

When using the Hybrid Flow, Token Error Responses are made in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.3.4**.

2.3.3.5. Token Response Validation

TOC

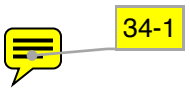
When using the Hybrid Flow, Token Responses are validated in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.3.5**.

2.3.3.6. ID Token

TOC

When using the Hybrid Flow, the contents of the ID Token returned from the Token Endpoint are the same as for the ID Token returned from the Authorization Endpoint, as defined in **Section 2.3.2.11**, with the exception of the differences specified in this section.

If an ID Token is returned from both the Authorization Endpoint and from the Token Endpoint, which is the case with the `response_type` values `code id_token` and `code id_token token`, it is RECOMMENDED that their values be the same. However, it is acceptable for `at_hash` and `c_hash` values that may have been present in the ID Token returned from the Authorization Endpoint to be omitted from the ID Token returned from the Token Endpoint. At a minimum, the `iss` and `sub` Claim values MUST be the identical. Furthermore, any other Claims about the subject that are present in both ID Tokens SHOULD have the same values in both.



2.3.3.7. ID Token Validation

TOC

When using the Hybrid Flow, the contents of the ID Token returned from the Token Endpoint MUST be validated in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.3.7**.

2.3.3.8. Access Token

TOC

If an Access Token is returned from both the Authorization Endpoint and from the Token Endpoint, which is the case with the `response_type` values `code token` and `code id_token token`, it is RECOMMENDED that their values be the same.

2.3.3.9. Access Token Validation

TOC

When using the Hybrid Code Flow, the Access Token returned from the Token Endpoint is validated in the same manner as for the Authorization Code Flow, as defined in **Section 2.1.3.8**.

3. Initiating Login from a Third Party

TOC

In some cases, the login flow can start at the Authorization Server or another party by contacting the Client via a stored link. The target resource at the Client can be a deep link, rather than a default landing page.

The Client MAY optionally **register** [OpenID.Registration] an `initiate_login_uri` that can be used by the Authorization Server or another party to initiate a login for an End-User at the Client.

The Authorization Server or a third party sends a Login Initiation Request to the Client Initiation URI with the following parameters:

login_hint
: OPTIONAL. Hint to the Authorization Server about the login identifier the End-User might use to log in. If the client receives a value for this string-valued parameter, it MUST include it in the subsequent authorization request as the `login_hint` parameter value.

iss
: REQUIRED. Issuer Identifier for the Issuer that the Client is to send the Authentication Request to. Its value MUST be a URL using the `https` scheme.

target_link_uri
: OPTIONAL. URI that the Client is requested to redirect to after authentication. Clients MUST verify the value of the `target_link_uri` to prevent being used as an open redirector to external sites.

Other parameters MAY be sent, if defined by extensions. Any parameters used that are not understood MUST be ignored by the Client.

Clients SHOULD employ frame busting and other techniques to prevent End-Users from being logged in by third party sites without their knowledge through attacks such as Clickjacking. Refer to Section 4.4.1.9 of **[RFC6819]** for more details.

---

4. Claims

TOC

This section specifies how the Client can obtain Claims about the End-User and defines a standard set of basic profile Claims. Pre-defined sets of claims can be requested using specific scope values or individual claims can be requested using the `claims` request parameter. The claims can come directly from the OpenID Provider or from distributed sources as well.

4.1. Requesting Claims using Scope Values

TOC

OpenID Connect Clients use `scope` values as defined in Section 3.3 of **OAuth 2.0** [RFC6749] to specify what access privileges are being requested for Access Tokens. The scopes associated with Access Tokens determine what resources will be available when they are used to access OAuth 2.0 protected endpoints. Protected Resource endpoints MAY perform different actions and return different information based on the scope values and other parameters used when requesting the presented Access Token.

For OpenID Connect, scopes can be used to request that specific sets of information be made available as Claim Values. This specification describes only the scope values used by OpenID Connect.

OpenID Connect allows additional scope values to be defined and used. Scope values used that are not understood by an implementation SHOULD be ignored.

Claims requested by the following scopes are treated by Authorization Servers as Voluntary Claims.

OpenID Connect defines the following `scope` values:

openid
: REQUIRED. Informs the Authorization Server that the Client is making an OpenID Connect request. If the `openid` scope value is not present, the behavior is entirely unspecified.

profile
: OPTIONAL. This scope value requests access to the End-User's default profile

Claims, which are: `name`, `family_name`, `given_name`, `middle_name`, `nickname`, `preferred_username`, `profile`, `picture`, `website`, `gender`, `birthdate`, `zoneinfo`, `locale`, and `updated_at`.

email
: OPTIONAL. This scope value requests access to the `email` and `email_verified` Claims.

address
: OPTIONAL. This scope value requests access to the `address` Claim.

phone
: OPTIONAL. This scope value requests access to the `phone_number` and `phone_number_verified` Claims.

Multiple scope values MAY be used by creating a space delimited, case sensitive list of ASCII scope values.

The Claims requested by the `profile`, `email`, `address`, and `phone` scope values are returned from the UserInfo Endpoint, as described in **Section 4.3.2**, when a `response_type` value is used that results in an Access Token being issued. However, when the `response_type` value used is `id_token` (which issues no Access Token), the resulting Claims are returned in the ID Token.

In some cases, the End-User will be given the option to have the OpenID Provider decline to provide some or all information requested by Clients. To minimize the amount of information that the End-User is being asked to disclose, a Client can elect to only request a subset of the information available from the UserInfo Endpoint.

The following is a non-normative example of a `scope` Request:

```
scope=openid profile email phone
```

4.2. Standard Claims

TOC

This specification defines a set of standard Claims. They can be requested to be returned either in the UserInfo Response or in the ID Token.

Member	Type	Description
sub	string	Subject - Identifier for the End-User at the Issuer.
name	string	End-User's full name in displayable form including all name parts, possibly including titles and suffixes, ordered according to the End-User's locale and preferences.
given_name	string	Given name(s) or first name(s) of the End-User. Note that in some cultures, people can have multiple given names; all can be present, with the names being separated by space characters.
family_name	string	Surname(s) or last name(s) of the End-User. Note that in some cultures, people can have multiple family names or no family name; all can be present, with the names being separated by

		space characters.
middle_name	string	Middle name(s) of the End-User. Note that in some cultures, people can have multiple middle names; all can be present, with the names being separated by space characters. Also note that in some cultures, middle names are not used.
nickname	string	Casual name of the End-User that may or may not be the same as the `given_name`. For instance, a `nickname` value of `Mike` might be returned alongside a `given_name` value of `Michael`.
preferred_username	string	Shorthand name that the End-User wishes to be referred to at the RP, such as `janedoe` or `j.doe`. This value MAY be any valid JSON string including special characters such as `@`, `/`, or whitespace. This value MUST NOT be relied upon to be unique by the RP. (See **Section 4.2.3**.)
profile	string	URL of the End-User's profile page. The contents of this Web page SHOULD be about the End-User.
picture	string	URL of the End-User's profile picture. This URL MUST refer to an image file (for example, a PNG, JPEG, or GIF image file), rather than to a Web page containing an image. Note that this URL SHOULD specifically reference a profile photo of the End-User suitable for displaying when describing the End-User, rather than an arbitrary photo taken by the End-User.
website	string	URL of the End-User's Web page or blog. This Web page SHOULD contain information published by the End-User or an organization that the End-User is affiliated with.
email	string	End-User's preferred e-mail address. Its value MUST conform to the **RFC 5322** [RFC5322] addr-spec syntax. This value MUST NOT be relied upon to be unique by the RP, as discussed in **Section 4.2.3**.
email_verified	boolean	True if the End-User's e-mail address has been verified; otherwise false. When this Claim Value is `true`, this means that the OP took affirmative steps to ensure that this e-mail address was controlled by the End-User at the time the verification was performed. The means by which an e-mail address is verified is context-specific, and dependent upon the trust framework or contractual agreements within which the parties are operating.
gender	string	End-User's gender. Values defined by this specification are `female` and `male`. Other values MAY be used when neither of the defined values are applicable.
birthdate	string	End-User's birthday, represented as an **ISO 8601:2004** [ISO8601-2004] `YYYY-MM-DD` format. The year MAY be `0000`, indicating that it is omitted. To represent only the year, `YYYY` format is allowed. Note that depending on the underlying platform's date related function, providing just year can result in varying month and day, so the implementers need to take this factor into account to correctly process the dates.
zoneinfo	string	String from zoneinfo **[zoneinfo]** time zone database representing the End-User's time zone. For example, `Europe/Paris` or `America/Los_Angeles`.

locale	string	End-User's locale, represented as a **BCP47** [RFC5646] language tag. This is typically an **ISO 639-1 Alpha-2** [ISO639-1] language code in lowercase and an **ISO 3166-1 Alpha-2** [ISO3166-1] country code in uppercase, separated by a dash. For example, `en-US` or `fr-CA`. As a compatibility note, some implementations have used an underscore as the separator rather than a dash, for example, `en_US`; Implementations MAY choose to accept this locale syntax as well.
phone_number	string	End-User's preferred telephone number. **E.164** [E.164] is RECOMMENDED as the format of this Claim, for example, `+1 (425) 555-1212` or `+56 (2) 687 2400`. If the phone number contains an extension, it is RECOMMENDED that the extension be represented using the **RFC 3966** [RFC3966] extension syntax, for example, `+1 (604) 555-1234;ext=5678`.
phone_number_verified	boolean	True if the End-User's phone number has been verified; otherwise false. When this Claim Value is `true`, this means that the OP took affirmative steps to ensure that this phone number was controlled by the End-User at the time the verification was performed. The means by which a phone number is verified is context-specific, and dependent upon the trust framework or contractual agreements within which the parties are operating. When true, the `phone_number` Claim MUST be in E.164 format and any extensions MUST be represented in RFC 3966 format.
address	JSON object	End-User's preferred address. The value of the `address` member is a **JSON** [RFC4627] structure containing some or all of the members defined in **Section 4.2.1**.
updated_at	number	Time the End-User's information was last updated. The time is represented as the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.

Table 1: Reserved Member Definitions

For privacy reasons, OpenID Providers MAY elect to not return values for some requested Claims.

The `sub` (subject) Claim in the UserInfo Endpoint response MUST exactly match the `sub` Claim in the ID Token, before using additional UserInfo Endpoint Claims.

The UserInfo Endpoint MUST return Claims in JSON format unless a different format was specified during Registration **[OpenID.Registration]**. The UserInfo Endpoint MAY return Claims in JWT format, which can be signed and/or encrypted. The UserInfo Endpoint MUST return a content-type header to indicate which format is being returned. The following are accepted content types:

Content-Type	Format Returned
application/json	plain text JSON object
application/jwt	JSON Web Token (JWT)

The following is a non-normative example of a UserInfo Response body:

```
{
 "sub": "248289761001",
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "preferred_username": "j.doe",
 "email": "janedoe@example.com",
 "picture": "http://example.com/janedoe/me.jpg"
}
```

4.2.1. Address Claim

TOC

The Address Claim represents a physical mailing address. Implementations MAY return only a subset of the fields of an `address`, depending upon the information available and the End-User's privacy preferences. For example, the `country` and `region` might be returned without returning more fine-grained address information.

Implementations MAY return just the full address as a single string in the formatted sub-field, or they MAY return just the individual component fields using the other sub-fields, or they MAY return both. If both variants are returned, they SHOULD be describing the same address, with the formatted address indicating how the component fields are combined.

formatted
: Full mailing address, formatted for display or use on a mailing label. This field MAY contain multiple lines, separated by newline characters.

street_address
: Full street address component, which MAY include house number, street name, Post Office Box, and multi-line extended street address information. This field MAY contain multiple lines, separated by newline characters.

locality
: City or locality component.

region
: State, province, prefecture or region component.

postal_code
: Zip code or postal code component.

country
: Country name component.

4.2.2. Claims Languages and Scripts

TOC

Human-readable Claim Values and Claim Values that reference human-readable values MAY be represented in multiple languages and scripts. To specify the languages and scripts, **BCP47** [RFC5646] language tags are added to member names, delimited by a `#` character. For example, `family_name#ja-Kana-JP` expresses the Family Name in Katakana in Japanese, which is commonly used to index and represent the phonetics of the Kanji representation of the same represented as `family_name#ja-Hani-JP`. As another example, both `website` and `website#de` Claim Values might be returned,

referencing a Web site in an unspecified language and a Web site in German.

Since Claim Names are case sensitive, it is strongly RECOMMENDED that language tag values used in Claim Names be spelled using the character case with which they are registered in the **IANA Language Subtag Registry** [IANA.Language]. In particular, normally language names are spelled with lowercase characters, region names are spelled with uppercase characters, and scripts are spelled with mixed case characters. However, since BCP47 language tag values are case insensitive, implementations SHOULD interpret the language tag values supplied in a case insensitive manner.

Per the recommendations in BCP47, language tag values for Claims SHOULD only be as specific as necessary. For instance, using `fr` might be sufficient in many contexts, rather than `fr-CA` or `fr-FR`. Where possible, OPs SHOULD try to match requested Claim locales with Claims it has. For instance, if the Client asks for a Claim with a `de` (German) language tag and the OP has a value tagged with `de-CH` (Swiss German) and no generic German value, it would be appropriate for the OP to return the Swiss German value to the Client. (This intentionally moves as much of the complexity of language tag matching to the OP as possible, to simplify Clients.)

A `claims_locales` request can be used to specify the preferred languages and scripts to use for the returned Claims. **Section 4.5.2** describes how to request that specific Claims use particular languages and scripts.

When the OP determines, either through the `claims_locales` parameter, or by other means, that the End-User and Client are requesting Claims in only one set of languages and scripts, it is RECOMMENDED that OPs return Claims without language tags when they employ this language and script. It is also RECOMMENDED that Clients be written in a manner that they can handle and utilize Claims using language tags.

4.2.3. Claim Stability and Uniqueness

TOC

The `sub` (subject) and `iss` (issuer) Claims are the only Claims that a Client can rely upon as a stable identifier for the End-User, since the `sub` Claim MUST be locally unique and never reassigned within the Issuer for a particular End-User, as described in **Section 2.1.3.6**. Therefore, the only guaranteed unique identifier for a given End-User is the combination of the `iss` Claim and the `sub` Claim.

All other Claims carry no such guarantees across different issuers in terms of stability over time or uniqueness across users, and Issuers are permitted to apply local restrictions and policies. For instance, an Issuer MAY re-use an `email` Claim value across different End-Users at different points in time, and the claimed `email` address for a given End-User MAY change over time. Therefore, other Claims such as `email`, `phone_number`, and `preferred_username` and MUST NOT be used as unique identifiers for the End-User.

4.2.4. Additional Claims

TOC

While this specification defines only small set of Claims as standard Claims, other Claims MAY be used in conjunction with the standard Claims. When using such Claims, it is RECOMMENDED that collision resistant names be used for the Claim Names, as described

in Section 4.2 (Public Claim Names) of the **JSON Web Token (JWT)** [JWT] specification. Alternatively, Private Claim Names can be safely used when naming conflicts are unlikely to arise, as described in 4.3 of the JWT specification. Or, if specific additional Claims will have broad and general applicability, they can be registered with Reserved Claim Names, per Sections 4.1 and 9.1 of the JWT specification.

4.3. UserInfo Endpoint

TOC

The UserInfo Endpoint is an OAuth 2.0 Protected Resource that returns Claims about the authenticated End-User. To obtain the requested Claims about the End-User, the Client makes a `GET` or `POST` request to the UserInfo Endpoint. These Claims are represented by a JSON object that contains a collection of name and value pairs for the Claims.

Communication with the UserInfo Endpoint MUST utilize TLS. See **Section 15.17** for more information on using TLS.

The UserInfo Endpoint MUST support the use of the HTTP `GET` and HTTP `POST` methods defined in **RFC 2616** [RFC2616] at the UserInfo Endpoint.

The UserInfo Endpoint MUST accept Access Tokens as **OAuth 2.0 Bearer Token Usage** [RFC6750].

The UserInfo Endpoint SHOULD support the use of **Cross Origin Resource Sharing (CORS)** [CORS] and or other methods as appropriate to enable Java Script Clients to access the endpoint.

4.3.1. UserInfo Request

TOC

Clients send requests with the following parameters to the UserInfo Endpoint to obtain Claims about the End-User:

access_token
: REQUIRED. Access Token obtained from an OpenID Connect Authorization Server.

The Client sends the UserInfo Request using either HTTPS `GET` or HTTPS `POST`. It is RECOMMENDED that the Client use the `Authorization` header field method for all requests and that they use the `GET` method.

The Access Token obtained from an OpenID Connect Authorization Request MUST be sent as a Bearer Token. Section 2 of the **OAuth 2.0 Bearer Token Usage** [RFC6750] specification documents the permissible methods of sending the Access Token.

The following is a non-normative example of a UserInfo request:

```
GET /userinfo HTTP/1.1
Host: server.example.com
Authorization: Bearer SlAV32hkKG
```

4.3.2. UserInfo Successful Response

TOC

The UserInfo Claims MUST be returned as the members of a JSON object unless a signed or encrypted response was requested during Client Registration. The Claims defined in **Section 4.2** can be returned, as can additional Claims not specified there.

If a Claim is not returned, that Claim Name SHOULD be omitted from the JSON object representing the Claims; it SHOULD NOT be present with a null or empty string value.

The `sub` (subject) Claim MUST always be returned in the UserInfo Response.

NOTE: The UserInfo Endpoint response is not guaranteed to be about the End-User identified by the `sub` (subject) element of the ID Token. The `sub` Claim in the UserInfo Endpoint response MUST be verified to exactly match the `sub` Claim in the ID Token before using additional UserInfo Endpoint Claims.

Upon receipt of the UserInfo request, the UserInfo Endpoint MUST return the JSON Serialization of the UserInfo response as in **Section 12.3** in the HTTP response body unless a different format was specified during Registration **[OpenID.Registration]**. The content-type of the HTTP response MUST be set to `application/json` if the response body is a text JSON structure; the response body SHOULD be encoded using UTF-8. If the JSON response is signed or encrypted, then the content-type MUST be set to `application/jwt`.

The following is a non-normative example of a UserInfo Response:

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

{
 "sub": "248289761001",
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "email": "janedoe@example.com",
 "picture": "http://example.com/janedoe/me.jpg"
}
```

4.3.3. UserInfo Error Response

TOC

When an error condition occurs, the UserInfo Endpoint returns an Error Response as defined in Section 3 of **OAuth 2.0 Bearer Token Usage** [RFC6750].

The following is a non-normative example of a UserInfo Error Response:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example.com",
                    error="invalid_token",
                    error_description="The Access Token expired"
```

4.3.4. UserInfo Response Validation

TOC

The Client MUST validate the UserInfo Response as follows:

1. If the Client has provided a `userinfo_encrypted_response_alg` parameter during Registration, decrypt the UserInfo Response using the key pair specified during Registration.
2. If the response was signed, the Client SHOULD validate the signature according to **JWS** [JWS].
3. Check that the OP that responded was the intended OP through a TLS server certificate check, per **RFC 6125** [RFC6125].

4.4. Requesting Claims Locales with the "claims_locales" Request Parameter

TOC

OpenID Connect defines the following Authorization Request parameter to enable requesting that Claims be returned for specific locales:

claims_locales
: OPTIONAL. End-User's preferred languages and scripts for Claims being returned, represented as a space-separated list of **BCP47** [RFC5646] language tag values, ordered by preference. An error SHOULD NOT result if some or all of the requested locales are not supported by the OpenID Provider.

4.5. Requesting Claims using the "claims" Request Parameter

TOC

OpenID Connect defines the following Authorization Request parameter to enable requesting individual Claims and specifying parameters that apply to the requested Claims:

claims
: OPTIONAL. This parameter is used to request that specific Claims be returned. The value is a JSON object listing the requested Claims.

The `claims` Authorization Request parameter requests that specific Claims be returned from the UserInfo Endpoint and/or in the ID Token. It is represented as a JSON object containing lists of Claims being requested from these locations. Properties of the Claims being requested MAY also be specified.

Support for the `claims` parameter is OPTIONAL. Should an OP not support this parameter and an RP uses it, the OP SHOULD return a set of Claims to the RP that it believes would be useful to the RP and the End-User using whatever heuristics it believes are appropriate. The `claims_parameter_supported` Discovery result indicates whether the OP supports this parameter.

The `claims` parameter value is represented in an OAuth 2.0 request as UTF-8 encoded JSON (which ends up being form-urlencoded when passed as an OAuth parameter). When used in a Request Object value, per **Section 5.1**, the JSON is used as the value of the `claims` member.

The top-level members of the Claims request JSON object are:

userinfo
: OPTIONAL. Requests that the listed individual Claims be returned from the UserInfo Endpoint. If present, the listed Claims are being requested to be added to any Claims that are being requested using `scope` values. If not present, the Claims being requested from the UserInfo Endpoint are only those requested using `scope` values.
When the `userinfo` member is used, the request MUST also use a `response_type` value that results in an Access Token being issued to use at the UserInfo Endpoint.

id_token
: OPTIONAL. Requests that the listed individual Claims be returned in the ID Token. If present, the listed Claims are being requested to be added to the default Claims in the ID Token. If not present, the default ID Token Claims are requested.

Other members MAY be present. Any members used that are not understood MUST be ignored.

An example Claims request is as follows:

```
{
 "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},
   "acr": {"values": ["urn:mace:incommon:iap:silver"] }
  }
}
```

Note that a Claim that is not in the standard set defined in **Section 4.2**, the (example) `http://example.info/claims/groups` Claim, is being requested. Using the `claims` parameter is the only way to request Claims outside the standard set. It is also the only way to request specific combinations of the standard Claims that cannot be specified using scope values.

4.5.1. Individual Claims Requests

TOC

The `userinfo` and `id_token` members of the `claims` request both are JSON objects with the names of the individual Claims being requested as the member names. The member values MUST be one of the following:

null
: Indicates that this Claim is being requested in the default manner. In particular, this is a Voluntary Claim. For instance, the Claim request:

```
"given_name": null
```

requests the `given_name` Claim in the default manner.

JSON Object
: Used to provide additional information about the Claim being requested. This specification defines the following members:

  essential
  : OPTIONAL. Indicates whether the Claim being requested is an Essential Claim. If the value is `true`, this indicates that the Claim is an Essential Claim. For instance, the Claim request:

    ```
    "auth_time": {"essential": true}
    ```

    can be used to specify that it is Essential to return an `auth_time` Claim Value.
    If the value is `false`, it indicates that it is a Voluntary Claim. The default is `false`.
    By requesting Claims as Essential Claims the Client indicates to the End-User that releasing these Claims will ensure a smooth authorization for the specific task requested by the End-User. Note that even if the Claims are not available because the End-User did not authorize their release or they are not present, the Authorization Server MUST NOT generate an error when Claims are not returned, whether they are Essential or Voluntary, unless otherwise specified in the description of the specific claim.

  value
  : OPTIONAL. Requests that the Claim be returned with a particular value. For instance the Claim request:

    ```
    "sub": {"value": "248289761001"}
    ```

    can be used to specify that the request apply to the End-User with subject identifier `248289761001`.
    The value of the `value` member MUST be a valid value for the Claim being requested. Definitions of individual Claims can include requirements on how and whether the `value` qualifier is to be used when requesting that Claim.

  values
  : OPTIONAL. Requests that the Claim be returned with one of a set of values, with the values appearing in order of preference. For instance the Claim request:

    ```
    "acr": {"essential": true,
            "values": ["urn:mace:incommon:iap:silver",
                       "urn:mace:incommon:iap:bronze"]}
    ```

    specifies that it is Essential that the `acr` Claim be returned with either the value `urn:mace:incommon:iap:silver` or `urn:mace:incommon:iap:bronze`.

The values in the `values` member array MUST be valid values for the Claim being requested. Definitions of individual Claims can include requirements on how and whether the `values` qualifier is to be used when requesting that Claim.

Other members MAY be defined to provide additional information about the requested Claims. Any members used that are not understood MUST be ignored.

Note that when the `claims` request parameter is supported, the scope values that request Claims, as defined in **Section 4.1**, are effectively shorthand methods for requesting sets of individual Claims. For example, using the scope value `openid email` and a `response_type` that returns an Access Token is equivalent to using the scope value `openid` and the following request for individual Claims.

Equivalent of using the `email` scope value:

```
{
 "userinfo":
  {
   "email": null,
   "email_verified": null
  }
}
```

4.5.1.1. Requesting the "acr" Claim

TOC

If the `acr` Claim is requested as an Essential Claim for the ID Token with a `values` parameter requesting specific Authentication Context Class Reference values and the implementation supports the `claims` parameter, the Authorization Server MUST return an `acr` Claim Value that matches one of the requested values. The Authorization Server MAY ask the End-User to re-authenticate with additional factors to meet this requirement. If this is an Essential Claim and the requirement cannot be met, then the Authorization Server MUST treat that outcome as a failed authentication attempt.

Note that the Client MAY request the `acr` Claim as a Voluntary Claim by using the `acr_values` request parameter or by not including "essential": true in an individual `acr` Claim request. If the Claim is not Essential and a requested value cannot be provided, the Authorization Server SHOULD return the session's current `acr` as the value of the `acr` Claim. If the Claim is not Essential, the Authorization Server is not required to provide this Claim in its response.

If the client requests the `acr` Claim using both the `acr_values` request parameter and an individual `acr` Claim request for the ID Token listing specific requested values, the resulting behavior is unspecified.

4.5.2. Languages and Scripts for Individual Claims

TOC

As described in **Section 4.2.2**, human-readable Claims values and Claim Values that reference human-readable values MAY be represented in multiple languages and scripts. Within a request for individual Claims, requested languages and scripts for particular Claims MAY be requested by including Claim Names that contain `#`-separated **BCP47** [RFC5646] language tags in the Claims request, using the Claim Name syntax specified in **Section 4.2.2**. For example, a Family Name in Katakana in Japanese can be requested using the Claim Name `family_name#ja-Kana-JP` and a Kanji representation of the Family Name in Japanese can be requested using the Claim Name `family_name#ja-Hani-JP`. A German-language Web site can be requested with the Claim Name `website#de`.

If an OP receives a request for human-readable Claims in a language and script that it doesn't have, any versions of those Claims returned that don't use the requested language and script SHOULD use a language tag in the Claim Name.

4.6. Claim Types

TOC

The UserInfo Endpoint MAY return the following three types of Claims:

Normal Claims
: Claims that are directly asserted by the OpenID Provider.

Aggregated Claims
: Claims that are asserted by a Claims Provider other than the OpenID Provider but are returned by OpenID Provider.

Distributed Claims
: Claims that are asserted by a Claims Provider other than the OpenID Provider but are returned as references by the OpenID Provider.

The UserInfo Endpoint MUST support Normal Claims.

Aggregated and Distributed Claims support is OPTIONAL.

4.6.1. Normal Claims

TOC

Normal Claims are represented as members in a JSON object. The Claim Name is the member name and the Claim Value is the member value.

The following is a non-normative response containing Normal Claims:

```
{
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "email": "janedoe@example.com",
 "picture": "http://example.com/janedoe/me.jpg"
}
```

4.6.2. Aggregated and Distributed Claims

TOC

Aggregated and distributed Claims are represented by using special `_claim_names` and `_claim_sources` members of the JSON object containing the Claims.

_claim_names
: JSON object whose member names are the Claim Names for the Aggregated and Distributed Claims. The member values are references to the member names in the `_claim_sources` member from which the actual Claim Values can be retrieved.

_claim_sources
: JSON object whose member names are referenced by the member values of the `_claim_names` member. The member values contain sets of Aggregated Claims or reference locations for Distributed Claims. The member values can have one of the following formats depending on whether it is providing Aggregated or Distributed Claims:

  Aggregated Claims
  : JSON object that MUST contain the `JWT` member whose value is a **JWT** [JWT] that MUST contain all the Claims in the `_claim_names` object that references the corresponding `_claim_sources` member. Other members MAY be present. Any members used that are not understood MUST be ignored.

    JWT
    : REQUIRED. JWT containing Claim Values.

    The JWT SHOULD NOT contain a `sub` (subject) Claim unless its value is an identifier for the End-User at the Claims Provider (and not for the OpenID Provider or another party); this typically means that a `sub` Claim SHOULD NOT be provided.

  Distributed Claims
  : JSON object that contains the following members and values:

    endpoint
    : REQUIRED. OAuth 2.0 resource endpoint from which the associated Claim can be retrieved. The endpoint URL MUST return the Claim as a JWT.

    access_token
    : OPTIONAL. Access Token enabling retrieval of the Claims from the endpoint URL by using the **OAuth 2.0 Bearer Token Usage** [RFC6750] protocol. Claims SHOULD be requested using the Authorization Request header field and Claims Providers MUST support this method. If the Access Token is not available, Clients MAY need to retrieve the Access Token out of band or use an a priori Access Token that was negotiated between the Claims Provider and Client, or

the Claims Provider MAY reauthenticate the End-User and/or reauthorize the Client.

A `sub` (subject) Claim SHOULD NOT be returned from the Claims Provider unless its value is an identifier for the End-User at the Claims Provider (and not for the OpenID Provider or another party); this typically means that a `sub` Claim SHOULD NOT be provided.

4.6.2.1. Example of Aggregated Claims

TOC

In this non-normative example, Claims from Claims Provider A are combined with other Claims held by the OpenID provider, with the Claims from Claims Provider A being returned as Aggregated Claims.

In this example, these Claims about Jane Doe have been issued by Claims Provider A:

```
{
 "address": {
   "street_address": "1234 Hollywood Blvd.",
   "locality": "Los Angeles",
   "region": "CA",
   "postal_code": "90210",
   "country": "US"},
 "phone_number": "+1 (310) 123-4567"
}
```

Claims Provider A signs the JSON Claims, representing them in a signed JWT: jwt_header.jwt_part2.jwt_part3. It is this JWT that is used by the OpenID Provider.

In this example, this JWT containing Jane Doe's Aggregated Claims from Claims Provider A is combined with other Normal Claims, and returned as the following set of Claims:

```
{
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "birthdate": "0000-03-22",
 "eye_color": "blue",
 "email": "janedoe@example.com",
 "_claim_names": {
   "address": "src1",
   "phone_number": "src1"
 },
 "_claim_sources": {
   "src1": {"JWT": "jwt_header.jwt_part2.jwt_part3"}
 }
}
```

4.6.2.2. Example of Distributed Claims

TOC

In this non-normative example, the OpenID Provider combines Normal Claims that it holds with references to Claims held by two different Claims Providers, B and C, incorporating references to some of the Claims held by B and C as Distributed Claims.

In this example, these Claims about Jane Doe are held by Claims Provider B (Jane Doe's bank):

```
{
 "shipping_address": {
   "street_address": "1234 Hollywood Blvd.",
   "locality": "Los Angeles",
   "region": "CA",
   "postal_code": "90210",
   "country": "US"},
 "payment_info": "Some_Card 1234 5678 9012 3456",
 "phone_number": "+1 (310) 123-4567"
}
```

Also in this example, this Claim about Jane Doe is held by Claims Provider C (a credit agency):

```
{
 "credit_score": 650
}
```

The OpenID Provider returns Jane Doe's Claims along with references to the Distributed Claims from Claims Provider B and Claims Provider C by sending the Access Tokens and URLs of locations from which the Distributed Claims can be retrieved:

```
{
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "email": "janedoe@example.com",
 "birthdate": "0000-03-22",
 "eye_color": "blue",
 "_claim_names": {
   "payment_info": "src1",
   "shipping_address": "src1",
   "credit_score": "src2"
  },
 "_claim_sources": {
   "src1": {"endpoint":
              "https://bank.example.com/claim_source"},
   "src2": {"endpoint":
              "https://creditagency.example.com/claims_here",
            "access_token": "ksj3n283dke"}
 }
```

```
}
```

5. Passing Request Parameters as JWTs

TOC

OpenID Connect defines the following Authorization Request parameters to enable Authentication Requests to be signed and optionally encrypted:

request
: OPTIONAL. This parameter enables OpenID Connect requests to be passed in a single, self-contained parameter and to be signed and optionally encrypted. The parameter value is a Request Object value, as specified in **Section 5.1**. It represents the request as a JWT whose Claims are the request parameters above.

request_uri
: OPTIONAL. This parameter enables OpenID Connect requests to be passed by reference, rather than by value. The `request_uri` value is a URL using the `https` scheme referencing a resource containing a Request Object value, which is a JWT containing the request parameters.

Requests using these parameters are represented as JWTs, which are respectively passed by value or by reference. The ability to pass requests by reference is particularly useful for large requests. If one of these parameters is used, the other MUST NOT be used in the same request.

5.1. Passing a Request Object by Value

TOC

The `request` Authorization Request parameter enables OpenID Connect requests to be passed in a single, self-contained parameter and to be signed and optionally encrypted. It represents the request as a JWT whose Claims are the request parameters specified in **Section 2.1.2**. This JWT is called a Request Object.

Support for the `request` parameter is OPTIONAL. The `request_parameter_supported` Discovery result indicates whether the OP supports this parameter. Should an OP not support this parameter and an RP uses it, the OP MUST return the `request_not_supported` error.

When the `request` parameter is used, the OpenID Connect request parameter values contained in the JWT supersede those passed using the OAuth 2.0 request syntax. However, some parameters MAY be passed using the OAuth 2.0 request syntax even when a Request Object is used; this would typically be done to enable a cached, pre-signed (and possibly pre-encrypted) Request Object value to be used containing the fixed request parameters, while parameters that can vary with each request, such as `state` and `nonce`, are passed as OAuth 2.0 parameters.

Even if a `scope` parameter is present in the Request Object value, a `scope` parameter MUST always be passed using the OAuth 2.0 request syntax containing the `openid` scope value to indicate to the underlying OAuth 2.0 logic that this is an OpenID Connect request.

The Request Object MAY be signed or unsigned (plaintext). When it is plaintext, this is

indicated by use of the `none` algorithm **[JWA]** in the JWS header. If signed, the Request Object SHOULD contain the Claims `iss` (issuer) and `aud` (audience) as members, with their semantics being as defined in the **JWT** [JWT] specification.

The Request Object MAY also be encrypted using **JWE** [JWE], with nested signing and encryption performed as described in the JWT **[JWT]** specification.

`request` and `request_uri` parameters MUST NOT be included in Request Objects.

The following is a non-normative example of the Claims in a Request Object before base64url encoding and signing:

```
{
 "response_type": "code id_token",
 "client_id": "s6BhdRkqt3",
 "redirect_uri": "https://client.example.org/cb",
 "scope": "openid",
 "state": "af0ifjsldkj",
 "nonce": "n-0S6_WzA2Mj",
 "max_age": 86400,
 "claims":
  {
   "userinfo":
    {
     "given_name": {"essential": true},
     "nickname": null,
     "email": {"essential": true},
     "email_verified": {"essential": true},
     "picture": null
    },
   "id_token":
    {
     "gender": null,
     "birthdate": {"essential": true},
     "acr": {"values": ["urn:mace:incommon:iap:silver"]}
    }
  }
}
```

Signing it with the `RS256` algorithm results in this Request Object value (with line wraps within values for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9.ew0KICJyZXNwb25zZV90eXBlIjogImNvZGUgaWRfdG9rZW
4iLA0KICJjbGllbnRfaWQiOiAiczZCaGRSa3F0MyIsDQogInJlZGlyZWN0X3VyaSI6I
CJodHRwczovL2NsaWVudC5leGFtcGxlLm9yZy9jYiIsDQogInNjb3BlIjogIm9wZW5p
ZCIsDQogInN0YXRlIjogImFmMGlmanNsZGtqIiwNCiAibm9uY2UiOiAibi0wUzZfV3p
BMk1qIiwNCiAibWF4X2FnZSI6IDg2NDAwLA0KICJjbGFpbXMiOiANCiAgew0KICAgIn
VzZXJpbmZvIjogDQogICAgew0KICAgICAiZ2l2ZW5fbmFtZSI6IHsiZXNzZW50aWFsI
jogdHJ1ZX0sDQogICAgICJuaWNrbmFtZSI6IG51bGwsDQogICAgICJlbWFpbCI6IHsi
ZXNzZW50aWFsIjogdHJ1ZX0sDQogICAgICJlbWFpbF92ZXJpZmllZCI6IHsiZXNzZW5
0aWFsIjogdHJ1ZX0sDQogICAgICJwaWN0dXJlIjogbnVsbA0KICAgIH0sDQogICAiaW
RfdG9rZW4iOiANCiAgICB7DQogICAgICJnZW5kZXIiOiBudWxsLA0KICAgICAiYmlyd
GhkYXRlIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAgICAgImFjciI6IHsidmFsdWVz
```

```
IjogWyIyIl19DQogICAgfQ0KICB9DQp9.bOD4rUiQfzh4QPIs_f_R2GVBhNHcc1p2cQ
TgixB1tsYRs52xW4TO74USgb-nii3RPsLdfoPlsEbJLmtbxG8-TQBHqGAyZxMDPWy3p
hjeRt9ApDRnLQrjYuvsCj6byu9TVaKX9r1KDFGT-HLqUNlUTpYtCyM2B2rLkWM08ufB
q9JBCEzzaLRzjevYEPMaoLAOjb8LPuYOYTBqshRMUxy4Z380-FJ2Lc7VSfSu6HcB2nL
SjiKrrfI35xkRJsaSSmjasMYeDZarYCl7r4o17rFclk5KacYMYgAs-JYFkwab6Dd56Z
rAzakHt9cExMpg04lQIux56C-Qk6dAsB6W6W91AQ
```

The following is the RSA public key in JWK format that can be used to validate the Request Object signature in this and subsequent Request Object examples (with line wraps within values for display purposes only):

```
{
 "kty":"RSA",
 "n":"y9Lqv4fCp6Ei-u2-ZCKq83YvbFEk6JMs_pSj76eMkddWRuWX2aBKGHAtKlE5P
      7_vn__PCKZWePt3vGkB6ePgzAFu08NmKemwE5bQI0e6kIChtt_6KzT5OaaXDF
      I6qCLJmk51Cc4VYFaxgqevMncYrzaW_50mZ1yGSFIQzLYP8bijAHGVjdEFgZa
      ZEN9lsn_GdWLaJpHrB3ROlS50E45wxrlg9xMncVb8qDPuXZarvghLL0HzOuYR
      adBJVoWZowDNTpKpk2RklZ7QaBO7XDv3uR7s_sf2g-bAjSYxYUGsqkNA9b3xV
      W53am_UZZ3tZbFTIh557JICWKHlWj5uzeJXaw",
 "e":"AQAB"
}
```

5.1.1. Request using the "request" Request Parameter

TOC

The Client sends the Authorization Request to the Authorization Endpoint.

The following is a non-normative example of an Authorization Request using the `request` parameter (with line wraps within values for display purposes only):

```
https://server.example.com/authorize?
  response_type=code%20id_token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid
  &state=af0ifjsldkj
  &nonce=n-0S6_WzA2Mj
  &request=eyJhbGciOiJSUzI1NiJ9.ew0KICJyZXNwb25zZV90eXBlIjogImNvZG
  UgaWRfdG9rZW4iLA0KICJjbGllbnRfaWQiOiAiczZCaGRSa3F0MyIsDQogInJlZG
  lyZWN0X3VyaSI6ICJodHRwczovL2NsaWVudC5leGFtcGxlLm9yZy9jYiIsDQogIn
  Njb3BlIjogIm9wZW5pZCIsDQogInN0YXRlIjogImFmMGlmanNsZGtqIiwNCiAibm
  9uY2UiOiAibi0wUzZfV3pBMk1qIiwNCiAibWF4X2FnZSI6IDg2NDAwLA0KICJjbG
  FpbXMiOiANCiAgew0KICAgInVzZXJpbmZvIjogDQogICAgew0KICAgICAiZ2l2ZW
  5fbmFtZSI6IHsiZXNzZW50aWFsIjogdHJ1ZX0sDQogICAgICJuaWNrbmFtZSI6IG
  51bGwsDQogICAgICJlbWFpbCI6IHsiZXNzZW50aWFsIjogdHJ1ZX0sDQogICAgIC
  JlbWFpbF92ZXJpZmllZCI6IHsiZXNzZW50aWFsIjogdHJ1ZX0sDQogICAgICJwaW
  N0dXJlIjogbnVsbA0KICAgIH0sDQogICAiaWRfdG9rZW4iOiANCiAgICB7DQogIC
  AgICJnZW5kZXIiOiBudWxsLA0KICAgICAiYmlydGhkYXRlIjogeyJlc3NlbnRpYW
  wiOiB0cnVlfSwNCiAgICAgImFjciI6IHsidmFsdWVzIjogWyIyIl19DQogICAgfQ
  0KICB9DQp9.bOD4rUiQfzh4QPIs_f_R2GVBhNHcc1p2cQTgixB1tsYRs52xW4TO7
  4USgb-nii3RPsLdfoPlsEbJLmtbxG8-TQBHqGAyZxMDPWy3phjeRt9ApDRnLQrjY
  uvsCj6byu9TVaKX9r1KDFGT-HLqUNlUTpYtCyM2B2rLkWM08ufBq9JBCEzzaLRzj
```

```
evYEPMaoLAOjb8LPuYOYTBqshRMUxy4Z380-FJ2Lc7VSfSu6HcB2nLSjiKrrfI35
xkRJsaSSmjasMYeDZarYCl7r4o17rFclk5KacYMYgAs-JYFkwab6Dd56ZrAzakHt
9cExMpg04lQIux56C-Qk6dAsB6W6W91AQ
```

5.2. Passing a Request Object by Reference

TOC

The `request_uri` Authorization Request parameter enables OpenID Connect requests to be passed by reference, rather than by value. This parameter is used identically to the `request` parameter, other than that the Request Object value is retrieved from the resource at the specified URL, rather than passed by value.

The `request_uri_parameter_supported` Discovery result indicates whether the OP supports this parameter. Should an OP not support this parameter and an RP uses it, the OP MUST return the `request_uri_not_supported` error.

When the `request_uri` parameter is used, the OpenID Connect request parameter values contained in the referenced JWT supersede those passed using the OAuth 2.0 request syntax. However, some parameters MAY be passed using the OAuth 2.0 request syntax even when a `request_uri` is used; this would typically be done to enable a cached, pre-signed (and possibly pre-encrypted) Request Object value to be used containing the fixed request parameters, while parameters that can vary with each request, such as `state` and `nonce`, are passed as OAuth 2.0 parameters.

Even if a `scope` parameter is present in the referenced Request Object, a `scope` parameter MUST always be passed using the OAuth 2.0 request syntax containing the `openid` scope value to indicate to the underlying OAuth 2.0 logic that this is an OpenID Connect request.

Servers MAY cache the contents of the resources referenced by Request URIs. If the contents of the referenced resource could ever change, the URI SHOULD include the base64url encoded SHA-256 hash of the referenced resource contents as the fragment component of the URI. If the fragment value used for a URI changes, that signals the server that any cached value for that URI with the old fragment value is no longer valid.

Note that Clients MAY pre-register `request_uri` values using the `request_uris` parameter defined in Section 2.1 of the **OpenID Connect Dynamic Client Registration 1.0** [OpenID.Registration] specification. OPs can require that `request_uri` values used be pre-registered with the `require_request_uri_registration` discovery parameter.

The entire Request URI MUST NOT exceed 512 ASCII characters.

The contents of the resource referenced by the URL MUST be a Request Object. The scheme used in the `request_uri` value MUST be `https`, unless the target Request Object is signed in a way that is verifiable by the Authorization Server. The `request_uri` value MUST be reachable by the Authorization Server, and SHOULD be reachable by the Client.

The following is a non-normative example of the contents of a Request Object resource that can be referenced by a `request_uri` (with line wraps within values for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9.ew0KICJyZXNwb25zZV90eXBlIjogImNvZGUgaWRfdG9rZ
```

```
W4iLA0KICJjbGllbnRfaWQiOiAiczZCaGRSa3F0MyIsDQogInJlZGlyZWN0X3VyaSI
6ICJodHRwczovL2NsaWVudC5leGFtcGxlLm9yZy9jYiIsDQogInNjb3BlIjogIm9wZ
W5pZCIsDQogInN0YXRlIjogImFmMGlmanNsZGtqIiwNCiAibm9uY2UiOiAibi0wUzZ
fV3pBMk1qIiwNCiAibWF4X2FnZSI6IDg2NDAwLA0KICJjbGFpbXMiOiANCiAgew0KI
CAgInVzZXJpbmZvIjogDQogICAgew0KICAgICAiZ2l2ZW5fbmFtZSI6IHsiZXNzZW5
0aWFsIjogdHJ1ZX0sDQogICAgICJuaWNrbmFtZSI6IG51bGwsDQogICAgICJlbWFpb
CI6IHsiZXNzZW50aWFsIjogdHJ1ZX0sDQogICAgICJlbWFpbF92ZXJpZmllZCI6IHs
iZXNzZW50aWFsIjogdHJ1ZX0sDQogICAgICJwaWN0dXJlIjogbnVsbA0KICAgIH0sD
QogICAiaWRfdG9rZW4iOiANCiAgICB7DQogICAgICJnZW5kZXIiOiBudWxsLA0KICA
gICAiYmlydGhkYXRlIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAgICAgImFjciI6I
HsidmFsdWVzIjogWyIyIl19DQogICAgfQ0KICB9DQp9.bOD4rUiQfzh4QPIs_f_R2G
VBhNHcc1p2cQTgixB1tsYRs52xW4TO74USgb-nii3RPsLdfoPlsEbJLmtbxG8-TQBH
qGAyZxMDPWy3phjeRt9ApDRnLQrjYuvsCj6byu9TVaKX9r1KDFGT-HLqUNlUTpYtCy
M2B2rLkWM08ufBq9JBCEzzaLRzjevYEPMaoLAOjb8LPuYOYTBqshRMUxy4Z380-FJ2
Lc7VSfSu6HcB2nLSjiKrrfI35xkRJsaSSmjasMYeDZarYCl7r4o17rFclk5KacYMYg
As-JYFkwab6Dd56ZrAzakHt9cExMpg04lQIux56C-Qk6dAsB6W6W91AQ
```

5.2.1. URL Referencing the Request Object

TOC

The Client stores the Request Object resource either locally or remotely at a URL the Server can access. This is the Request URI, `request_uri`.

If the Request Object includes attribute values, it MUST NOT be revealed to anybody but the Authorization Server. As such, the `request_uri` MUST have appropriate entropy for its lifetime. It is RECOMMENDED that it be removed if it is known that it will not be used again or after a reasonable timeout unless access control measures are taken.

The following is a non-normative example of a Request URI value (with line wraps within values for display purposes only):

```
https://client.example.org/request.jwt#
  GkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM
```

5.2.2. Request using the "request_uri" Request Parameter

TOC

The Client sends the Authorization Request to the Authorization Endpoint.

The following is a non-normative example of an Authorization Request using the `request_uri` parameter (with line wraps within values for display purposes only):

```
https://server.example.com/authorize?
  response_type=code%20id_token
  &client_id=s6BhdRkqt3
  &request_uri=https%3A%2F%2Fclient.example.org%2Frequest.jwt
  %23GkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM
  &state=af0ifjsldkj&nonce=n-0S6_WzA2Mj
  &scope=openid
```

5.2.3. Authorization Server Fetches Request Object

TOC

Upon receipt of the Request, the Authorization Server MUST send a `GET` request to the `request_uri` to retrieve the content unless it is already cached and parse it to recreate the Authorization Request parameters.

Note that the RP SHOULD use a unique URI for each request utilizing distinct parameters, or otherwise prevent the Authorization Server from caching the `request_uri`.

The following is a non-normative example of this fetch process:

```
GET /request.jwt HTTP/1.1
Host: client.example.org
```

5.2.4. "request_uri" Rationale

TOC

There are several reasons that one might choose to use the `request_uri` parameter:

1. The set of request parameters can become large, and can exceed browser URI size limitations. Passing the request parameters by reference can solve this problem.
2. Passing a `request_uri` value, rather than a complete request by value, can reduce request latency.
3. Most requests for Claims from an RP are constant. The `request_uri` is a way of creating and sometimes also signing and encrypting a constant set of request parameters in advance. (The `request_uri` value becomes an "artifact" representing a particular fixed set of request parameters.)
4. Pre-registering a fixed set of request parameters at registration time enables OPs to cache and pre-validate the request parameters at registration time, meaning they need not be retrieved at request time.
5. Pre-registering a fixed set of request parameters at registration time enables OPs to vet the contents of the request from consumer protection and other points of views, either itself or by utilizing a third party.

5.3. Validating JWT-Based Requests

TOC

When the `request` or `request_uri` Authorization Request parameters are used, additional steps must be performed to validate the Authentication Request beyond those specified in Sections **2.1.2.2**, **2.2.2.2**, or **2.3.2.2**. These steps are to validate the JWT containing the Request Object and to validate the Request Object itself.

5.3.1. Encrypted Request Object

TOC

If the Authorization Server has advertised JWE encryption algorithms in the `request_object_encryption_alg_values_supported` and

request_object_encryption_enc_values_supported elements of its Discovery document **[OpenID.Discovery]**, or has supplied encryption algorithms by other means, these are used by the Client to encrypt the JWT.

The Authorization Server MUST decrypt the JWT in accordance with the **JSON Web Encryption** [JWE] specification. The result MAY be either a signed or unsigned (plaintext) Request Object. In the former case, signature validation MUST be performed as defined in **Section 5.3.2**.

The Authorization Server MUST return an error if decryption fails.

5.3.2. Signed Request Object

TOC

To perform Signature Validation, the alg parameter in the JWS header MUST match the value of the request_object_signing_alg set during Client Registration **[OpenID.Registration]** or a value that was pre-registered by other means. The signature MUST be validated against the key registered for that client_id and algorithm, in accordance with the **JSON Web Signature** [JWS] specification.

The Authorization Server MUST return an error if signature validation fails.

5.3.3. Request Parameter Assembly and Validation

TOC

The Authorization Server MUST assemble the set of Authorization Request parameters to be used from the Request Object value and the OAuth 2.0 Authorization Request parameters (minus the request or request_uri parameters). If the same parameter exists both in the Request Object and the OAuth Authorization Request parameters, the parameter in the Request Object is used. Using this set of Authorization Request parameters, the Authorization Server then validates the request the normal manner for the flow being used, as specified in Sections **2.1.2.2**, **2.2.2.2**, or **2.3.2.2**.

---

6. Self-Issued OpenID Provider

TOC

OpenID Connect supports Self-Issued OpenID Providers - personal OPs that issue self-signed ID Tokens. Self-Issued OPs use the special Issuer Identifier https://self-issued.me.

The messages used to communicate with Self-Issued OPs are mostly the same as those used to communicate with other OPs. Specifications for the few additional parameters used and for the values of some parameters in the Self-Issued case are defined in this section.

6.1. Self-Issued OpenID Provider Discovery

TOC

If the input identifier for the discovery process contains the domain self-issued.me,

dynamic discovery is not performed. Instead, then the following static configuration values are used:

```
{
 "authorization_endpoint":
   "openid:",
 "issuer":
   "https://self-issued.me",
 "scopes_supported":
   ["openid", "profile", "email", "address", "phone"],
 "response_types_supported":
   ["id_token"],
 "subject_types_supported":
   ["pairwise"],
 "id_token_signing_alg_values_supported":
   ["RS256"],
 "request_object_signing_alg_values_supported":
   ["none", "RS256"]
}
```

Note: The OpenID Foundation plans to host the OpenID Provider site `https://self-issued.me/`, including its WebFinger service, so that performing discovery on it returns the above static discovery information, enabling Clients to not need any special processing for discovery of the Self-Issued OP. This site will be hosted on an experimental basis. Production implementations should not take a dependency upon it without a subsequent commitment by the OpenID Foundation to host the site in a manner intended for production use.

6.2. Self-Issued OpenID Provider Registration

TOC

When using a Self-Issued OP, the Client is deemed to have registered with the OP and obtained following Client Registration Response.

client_id
: `redirect_uri` value of the Client.

client_secret_expires_at
: 0

Note: The OpenID Foundation plans to host the (stateless) endpoint `https://self-issued.me/registration/1.0/` that returns the response above, enabling Clients to not need any special processing for registration with the Self-Issued OP. This site will be hosted on an experimental basis. Production implementations should not take a dependency upon it without a subsequent commitment by the OpenID Foundation to host the site in a manner intended for production use.

6.2.1. Providing Information with the "registration" Request Parameter

TOC

OpenID Connect defines the following Authorization Request parameter to enable Clients to provide additional registration information to Self-Issued OpenID Providers:

registration
: OPTIONAL. This parameter is used by the Client to provide information about itself to a Self-Issued OP that would normally be provided to an OP during Dynamic Client Registration. The value is a JSON object containing Client metadata values, as defined in Section 2.1 of the **OpenID Connect Dynamic Client Registration 1.0** [OpenID.Registration] specification. The `registration` parameter SHOULD NOT be used when the OP is not a Self-Issued OP.

None of this information is REQUIRED by Self-Issued OPs, so the use of this parameter is OPTIONAL.

The `registration` parameter value is represented in an OAuth 2.0 request as UTF-8 encoded JSON (which ends up being form-urlencoded when passed as an OAuth parameter). When used in a Request Object value, per **Section 5.1**, the JSON is used as the value of the `registration` member.

The Registration parameters that would typically be used in requests to Self-Issued OPs are `policy_uri`, `tos_uri`, and `logo_uri`. If the Client uses more than one redirection URI, the `redirect_uris` parameter would be used to register them. Finally, if the Client is requesting encrypted responses, it would use the `jwks_uri`, `id_token_encrypted_response_alg` and `id_token_encrypted_response_enc` parameters.

6.3. Self-Issued OpenID Provider Request

TOC

The Client sends the Authorization Request to the Authorization Endpoint with the following parameters:

scope
: REQUIRED. `scope` parameter value, as specified in **Section 2.1.2**.

response_type
: REQUIRED. Constant string value `id_token`.

client_id
: REQUIRED. Client ID value for the Client, which in this case contains the `redirect_uri` value of the Client. Since the Client's `redirect_uri` URI value is communicated as the Client ID, a `redirect_uri` parameter is NOT REQUIRED to also be included in the request.

id_token_hint
: OPTIONAL. `id_token_hint` parameter value, as specified in **Section 2.1.2**. If the ID Token is encrypted to the Self-Issued OP, the `sub` (subject) of the signed ID Token MUST be sent as the `kid` (Key ID) of the JWE. Encrypting content to Self-Issued OPs is currently only supported when the OP's JWK key type is `RSA` and the encryption algorithm used is `RSA1_5`.

claims
: OPTIONAL. `claims` parameter value, as specified in **Section 4.5**.

registration
: OPTIONAL. This parameter is used by the Client to provide information about itself to a Self-Issued OP that would normally be provided to an OP during Dynamic Client Registration, as specified in **Section 6.2.1**.

request

OPTIONAL. Request Object value, as specified in **Section 5.1**. The Request Object MAY be encrypted to the Self-Issued OP by the Client. In this case, the `sub` (subject) of a previously issued ID Token for this Client MUST be sent as the `kid` (Key ID) of the JWE. Encrypting content to Self-Issued OPs is currently only supported when the OP's JWK key type is `RSA` and the encryption algorithm used is `RSA1_5`.

Other parameters MAY be sent. Note that all Claims are returned in the ID Token.

The entire URL MUST NOT exceed 2048 ASCII characters.

The following is a non-normative example response (with line wraps within values for display purposes only):

```
HTTP/1.1 302 Found
Location: openid://?
  response_type=id_token
  &client_id=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile
  &state=af0ifjsldkj
  &nonce=n-0S6_WzA2Mj
  registration=&%7B%22logo_uri%22%3A%22https%3A%2F%2F
    client.example.org%2Flogo.png%22%7D
```

6.4. Self-Issued OpenID Provider Response

TOC

OpenID Connect defines the following Claim for use in Self-Issued OpenID Provider Responses:

sub_jwk
: REQUIRED. Public key value used to check the signature of an ID Token issued by a Self-Issued OpenID Provider, as specified in **Section 6**. The key is a bare key in JWK **[JWK]** format (not an X.509 certificate value). The `sub_jwk` value is a JSON object. Use of the `sub_jwk` Claim is NOT RECOMMENDED when the OP is not Self-Issued.

The Self-Issued OpenID Provider response is the same as the normal implicit flow response with the following refinements. Since it is an implicit flow response, the response parameters will be returned in the URL fragment component.

1. The `iss` (issuer) Claim Value is `https://self-issued.me`.
2. A `sub_jwk` Claim is present, with its value being the public key value used to check the signature of the ID Token.
3. The `sub` (subject) Claim value is the base64url encoded SHA-256 hash of the concatenation of the octets of the UTF-8 representations of the base64url encoded key values in the `sub_jwk` Claim. When the `kty` value is `RSA`, the key values `n` and `e` are concatenated in that order. When the `kty` value is `EC`, the key values `crv`, `x`, and `y` are concatenated in that order.
4. No Access Token is returned for accessing a UserInfo Endpoint, so all Claims returned MUST be in the ID Token.

6.5. Self-Issued ID Token Validation

TOC

If any of the validation procedures defined in this specification fail, any operations requiring the information that failed to correctly validate MUST be aborted and the information that failed to validate MUST NOT be used.

To validate the ID Token in the Authorization or Token Endpoint Response, the Client MUST do the following:

1. The Client MUST validate that the value of the `iss` (issuer) Claim is `https://self-isued.me`. If `iss` contains a different value, the ID Token is not Self-Issued, and instead it MUST be validated according to **Section 2.1.3.7**.
2. The Client MUST validate that the `aud` (audience) Claim contains the value of the `redirect_uri` that the Client sent in the Authentication Request as an audience.
3. The Client MUST validate the signature of the ID Token according to **JWS** [JWS] using the algorithm specified in the `alg` parameter of the JWT header **[JWT]**, using the key in the `sub_jwk` Claim; the key is a bare key in JWK format (not an X.509 certificate value).
4. The `alg` value SHOULD be the default of `RS256`. It MAY also be `ES256`.
5. The Client MUST validate that the `sub` (subject) Claim value is the base64url encoded SHA-256 hash of the concatenation of the octets of the UTF-8 representations of the base64url encoded key values in the `sub_jwk` Claim. When the `kty` value is `RSA`, the key values `n` and `e` are concatenated in that order. When the `kty` value is `EC`, the key values `crv`, `x`, and `y` are concatenated in that order.
6. The current time MUST be less than the value of the `exp` Claim (possibly allowing for some small leeway to account for clock skew).
7. The `iat` Claim can be used to reject tokens that were issued too far away from the current time, limiting the amount of time that nonces need to be stored to prevent attacks. The acceptable range is Client specific.
8. If a nonce value was sent in the Authorization Request, a `nonce` Claim MUST be present and its value of the checked to verify that it is the same value as the one that was sent in the Authorization Request. The Client SHOULD check the `nonce` value for replay attacks. The precise method for detecting replay attacks is Client specific.

The following is a non-normative example of a base64url decoded Self-Issued ID Token (with line wraps within values for display purposes only):

```
{
 "iss": "https://self-issued.me",
 "sub": "wBy8QvHbPzUnL0x63h13QqvUYcOur1X0cbQpPVRqX5k",
 "aud": "https://client.example.org/cb",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "sub_jwk": {
   "kty":"RSA",
   "n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
   4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMs
```

```
      tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
      QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbI
      SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
      w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",
      "e":"AQAB"
    }
  }
```


7. Subject Identifier Types

TOC

The OpenID Provider's Discovery document SHOULD list its supported identifier types in the `subject_types_supported` element. If there is more than one type listed in the array, the Client MAY elect to provide its preferred identifier type using the `subject_type` parameter during Registration. The types supported by this specification are:

public
: This provides the same `sub` (subject) value to all Clients. It is the default if the provider has no `subject_types_supported` element in its discovery document.

pairwise
: This provides a different `sub` value to each Client, to prevent correlation of the End-User's activities by Clients without his permission.

7.1. Pairwise Identifier Algorithm

TOC

The OpenID Provider MUST calculate a unique `sub` (subject) value for each Sector Identifier. The subject value MUST NOT be reversible by any party other than the OpenID Provider.

Providers who use pairwise `sub` values SHOULD support the `sector_identifier_uri` in **Dynamic Client Registration** [OpenID.Registration]. It provides a way for a group of websites under common administrative control to have consistent pairwise `sub` values independent of the individual domain names. It also provides a way for Clients to change `redirect_uri` domains without having to reregister all of their users.

If the Client has not provided a value for `sector_identifier_uri` in **Dynamic Client Registration** [OpenID.Registration], the Sector Identifier used for pairwise identifier calculation is the host component of the registered `redirect_uri`. If there are multiple hostnames in the registered `redirect_uris`, the Client MUST register a `sector_identifier_uri`.

When a `sector_identifier_uri` is provided, the host component of that URL is used as the Sector Identifier for the pairwise identifier calculation. The value of the `sector_identifier_uri` MUST be a URL using the `https` scheme that points to a JSON file containing an array of `redirect_uri` values. The values of the registered `redirect_uris` MUST be included in the elements of the array, or the registration MUST fail.

A number of algorithms can be used by OpenID Providers to calculate pairwise identifiers. Three example methods are:

1. The Sector Identifier can be concatenated with a local account ID and a salt value that is kept secret by the Provider. The concatenated string is then hashed using an appropriate algorithm.

   Calculate `sub` = SHA-256 (sector_identifier | local_account_id | salt).

2. The Sector Identifier can be concatenated with a local account ID and a salt value that is kept secret by the Provider. The concatenated string is then encrypted using an appropriate algorithm.

   Calculate `sub` = AES-128 (sector_identifier | local_account_id | salt).

3. The Issuer creates a Globally Unique Identifier (GUID) for the pair of Sector Identifier and local account ID and stores this value.

---

8. Client Authentication

TOC

During Client Registration, the RP (Client) MAY register an authentication method. If no method is registered, the default method of `client_secret_basic` MUST be used.

The Supported options are:

client_secret_basic
: Clients that have received a `client_secret` value from the Authorization Server, authenticate with the Authorization Server in accordance with Section 3.2.1 of **OAuth 2.0** [RFC6749] using HTTP Basic authentication scheme.

client_secret_post
: Clients that have received a `client_secret` value from the Authorization Server, authenticate with the Authorization Server in accordance with Section 3.2.1 of **OAuth 2.0** [RFC6749] by including the Client Credentials in the request body.

client_secret_jwt
: Clients that have received a `client_secret` value from the Authorization Server create a JWT using an HMAC SHA algorithm, such as HMAC SHA-256. The HMAC (Hash-based Message Authentication Code) is calculated using the octets of the UTF-8 representation of the `client_secret` as the shared key. The Client authenticates in accordance with Section 2.2 of **JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants** [OAuth.JWT] and **Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants** [OAuth.Assertions]. The JWT MUST contain the following REQUIRED Claim Values and MAY contain the following OPTIONAL Claim Values:

    iss
    : REQUIRED. Issuer. This MUST contain the `client_id` of the OAuth Client.

    sub
    : REQUIRED. Subject. This MUST contain the `client_id` of the OAuth Client.

    aud
    : REQUIRED. Audience. The `aud` (audience) Claim. Value that

identifies the Authorization Server as an intended audience. The Authorization Server MUST verify that it is an intended audience for the token. The Audience SHOULD be the URL of the Authorization Server's Token Endpoint.

jti
: REQUIRED. JWT ID. A unique identifier for the token. The JWT ID MAY be used by implementations requiring message de-duplication for one-time use assertions.

exp
: REQUIRED. Expiration time on or after which the ID Token MUST NOT be accepted for processing.

iat
: OPTIONAL. Time at which the JWT was issued.

ID Tokens MAY contain other Claims. Any Claims used that are not understood MUST be ignored.

The authentication token MUST be sent as the value of the **[OAuth.Assertions]** `client_assertion` parameter.

The value of the **[OAuth.Assertions]** `client_assertion_type` parameter MUST be "urn:ietf:params:oauth:client-assertion-type:jwt-bearer", per **[OAuth.JWT]**.

private_key_jwt
: Clients that have registered a public key sign a JWT using that key. The Client authenticates in accordance with Section 2.2 of **JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants** [OAuth.JWT] and **Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants** [OAuth.Assertions]. The JWT MUST contain the following REQUIRED Claim Values and MAY contain the following OPTIONAL Claim Values:

iss
: REQUIRED. Issuer. This MUST contain the `client_id` of the OAuth Client.

sub
: REQUIRED. Subject. This MUST contain the `client_id` of the OAuth Client.

aud
: REQUIRED. Audience. The `aud` (audience) Claim. Value that identifies the Authorization Server as an intended audience. The Authorization Server MUST verify that it is an intended audience for the token. The Audience SHOULD be the URL of the Authorization Server's Token Endpoint.

jti
: REQUIRED. JWT ID. A unique identifier for the token. The JWT ID MAY be used by implementations requiring message de-duplication for one-time use assertions.

exp
: REQUIRED. Expiration time on or after which the ID Token MUST NOT be accepted for processing.

iat
: OPTIONAL. Time at which the JWT was issued.

The JWT MAY contain other Claims. Any Claims used that are not understood

MUST be ignored.
The authentication token MUST be sent as the value of the **[OAuth.Assertions]** `client_assertion` parameter.
The value of the **[OAuth.Assertions]** `client_assertion_type` parameter MUST be "urn:ietf:params:oauth:client-assertion-type:jwt-bearer", per **[OAuth.JWT]**.

For example (with line wraps within values for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
  code=i1WsRn1uB1&
  client_id=s6BhdRkqt3&
  client_assertion_type=
  urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer&
  client_assertion=PHNhbWxwOl ... ZT
```

---

9. Signatures and Encryption

TOC

Depending on the transport through which the messages are sent, the integrity of the message might not be guaranteed and the originator of the message might not be authenticated. To mitigate these risks, ID Token, UserInfo Response, Request Object, and Token Endpoint client authentication JWT values can utilize **JSON Web Signature (JWS)** [JWS] to sign their contents. To achieve message confidentiality, these values can also use **JSON Web Encryption (JWE)** [JWE] to encrypt their contents.

When the message is both signed and encrypted, it MUST be signed first and then encrypted, per **Section 15.14**, with nesting performed in the same manner as specified for JWTs **[JWT]**. Note that all JWE encryption methods perform integrity checking.

9.1. Supported Algorithms

TOC

The server advertises its supported signing and encryption algorithms in its discovery document. The algorithm identifiers are specified in **JWA** [JWA]. The related elements are:

userinfo_signing_alg_values_supported
: JSON array containing a list of the JWS **[JWS]** signing algorithms (`alg` values) **[JWA]** supported by the UserInfo Endpoint to encode the Claims in a JWT **[JWT]**.

userinfo_encryption_alg_values_supported
: JSON array containing a list of the JWE **[JWE]** encryption algorithms (`alg` values) **[JWA]** supported by the UserInfo Endpoint to encode the Claims in a JWT **[JWT]**.

userinfo_encryption_enc_values_supported

JSON array containing a list of the JWE encryption algorithms (`enc` values) **[JWA]** supported by the UserInfo Endpoint to encode the Claims in a JWT **[JWT]**.

id_token_signing_alg_values_supported
: JSON array containing a list of the JWS signing algorithms (`alg` values) supported by the Authorization Server for the ID Token to encode the Claims in a JWT **[JWT]**.

id_token_encryption_alg_values_supported
: JSON array containing a list of the JWE encryption algorithms (`alg` values) supported by the Authorization Server for the ID Token to encode the Claims in a JWT **[JWT]**.

id_token_encryption_enc_values_supported
: JSON array containing a list of the JWE encryption algorithms (`enc` values) supported by the Authorization Server for the ID Token to encode the Claims in a JWT **[JWT]**.

request_object_signing_alg_values_supported
: JSON array containing a list of the JWS signing algorithms (`alg` values) supported by the Authorization Server for Request Object values. Servers SHOULD support `none` and `RS256`.

request_object_encryption_alg_values_supported
: JSON array containing a list of the JWE encryption algorithms (`alg` values) supported by the Authorization Server for Request Object values.

request_object_encryption_enc_values_supported
: JSON array containing a list of the JWE encryption algorithms (`enc` values) supported by the Authorization Server for Request Object values.

token_endpoint_auth_signing_alg_values_supported
: JSON array containing a list of the JWS signing algorithms (`alg` values) supported by the Token Endpoint for the `private_key_jwt` and `client_secret_jwt` methods to encode the JWT **[JWT]**. Servers SHOULD support `RS256`.

The Client registers its REQUIRED algorithms for Signing and Encryption using the following Registration parameters:

request_object_signing_alg
: OPTIONAL. JWS signature algorithm **[JWA]** REQUIRED for Request Objects by the Authorization Server. All Request Objects from this `client_id` MUST be rejected if not signed by this algorithm. Servers SHOULD support `RS256`.

userinfo_signed_response_alg
: OPTIONAL. JWS signature algorithm **[JWA]** REQUIRED for UserInfo Responses. If this is specified the response will be **JWT** [JWT] serialized.

userinfo_encrypted_response_alg
: OPTIONAL. JWE `alg` algorithm **[JWA]** REQUIRED for UserInfo Responses. If this is requested in combination with signing, the response MUST be signed first then encrypted, per **Section 15.14**. If this is specified, the response will be **JWT** [JWT] serialized.

userinfo_encrypted_response_enc
: OPTIONAL. JWE `enc` algorithm **[JWA]** REQUIRED for UserInfo Responses. If `userinfo_encrypted_response_alg` is specified the default for this value is `A128CBC-HS256`.

id_token_signed_response_alg
: OPTIONAL. JWS signature algorithm **[JWA]** REQUIRED for ID Tokens issued to this `client_id`. The default if not specified is `RS256`. The public key for

validating the signature is provided by retrieving the JWK Set referenced by the `jwks_uri` element from Discovery.

id_token_encrypted_response_alg
: OPTIONAL. JWE `alg` algorithm **[JWA]** REQUIRED for ID Tokens issued to this `client_id`. If this is requested, the response MUST be signed then encrypted. The default if not specified is no encryption.

id_token_encrypted_response_enc
: OPTIONAL. JWE `enc` algorithm **[JWA]** REQUIRED for ID Tokens issued to this `client_id`. If `id_token_encrypted_response_alg` is specified the default for this value is `A128CBC-HS256`.

9.2. Keys

TOC

The OpenID Provider provides its public keys during Discovery using the following element:

jwks_uri
: REQUIRED. URL of the OP's JSON Web Key Set **[JWK]** document. This contains the signing key(s) the Client uses to validate signatures from the OP. The JWK Set MAY also contain the Server's encryption key(s), which are used by Clients to encrypt requests to the Server. When both signing and encryption keys are made available, a `use` (Key Use) parameter value is REQUIRED for all keys in the document to indicate each key's intended usage.

Likewise, the Client can provide its public keys during Registration using the following element:

jwks_uri
: OPTIONAL. URL for the Client's JSON Web Key Set **[JWK]** document. If the Client signs requests to the Server, it contains the signing key(s) the Server uses to validate signatures from the Client. The JWK Set MAY also contain the Client's encryption keys(s), which are used by the Server to encrypt responses to the Client. When both signing and encryption keys are made available, a `use` (Key Use) parameter value is REQUIRED for all keys in the document to indicate each key's intended usage.

When both signing and encryption keys are made available, the `use` (Key Use) parameter value is REQUIRED for all keys in the JWK Set at the `jwks_uri` to indicate each key's intended usage. Although some algorithms allow the same key pair to be used for both signatures and encryption, doing so is NOT RECOMMENDED, as it is less secure.

In both cases, the JWK `x5c` parameter MAY be used to provide X.509 representations of keys provided. When used, the bare key values MUST still be present and MUST match those in the certificate.

9.3. Signing

TOC

The signing party MUST select a signature algorithm based on the supported algorithms of the recipient in **Section 9.1**.

Asymmetric Signatures
: When using RSA or ECDSA Signatures, the `alg` Claim of the JWS header MUST be set to the appropriate algorithm as defined in **JSON Web Algorithms** [JWA]. The private key MUST be the one associated with the Public Signing Key provided in **Section 9.2**. If there are multiple keys in the referenced JWK document, a `kid` value MUST be provided in the JWS header. The key usage of the respective keys MUST support signature.

Symmetric Signatures
: When using MAC-based signatures, the `alg` Claim of the JWS header MUST be set to a MAC algorithm, as defined in **JSON Web Algorithms** [JWA]. The MAC key used is the octets of the UTF-8 representation of the `client_secret` value. See **Section 15.19** for a discussion of entropy requirements for `client_secret` values. Symmetric signatures MUST never be used by public (non-confidential) Clients because of their inability to keep secrets.

See **Section 15.20** for Security Considerations about the need for signed requests.

9.3.1. Rotation of Asymmetric Signing Keys

TOC

Rotation of signing keys can be accomplished with the following approach. The signer publishes its keys in a JWK Set at the `jwks_uri` location and includes the `kid` of the signing key in the JWS header of each message to indicate to the verifier which key is to be used to validate the signature. Keys can be rolled over by periodically adding new keys to the JWK Set at `jwks_uri`. The signer can begin using a new key at its discretion and signals the change to the verifier using the `kid` value. The verifier knows to go back to the `jwks_uri` to re-retrieve the keys when it sees an unfamiliar `kid` value. The JWK Set document at the `jwks_uri` SHOULD retain recently decommissioned signing keys for a reasonable period of time to facilitate a smooth transition.

9.4. Encryption

TOC

The encrypting party MUST select an encryption algorithm based on the supported algorithms of the recipient in **Section 9.1**. All JWTs MUST be signed before encryption to enable verification of the Issuer.

Asymmetric Encryption: RSA
: Use the link registered/discovered in **Section 9.2** to retrieve the relevant key. If there are multiple keys in the referenced JWK document, a `kid` value MUST be provided in the JWE header. Use the supported RSA key wrapping algorithm to wrap a random `Content Master Key` to be used for encrypting the signed JWT. The key usage of the respective keys MUST include encryption.

Asymmetric Encryption: Elliptic Curve
: Create an ephemeral Elliptic Curve public key for the `epk` element of the JWE header. Use the link registered/discovered in **Section 9.2** to retrieve the relevant key. If there are multiple keys in the referenced JWK document, a `kid` value MUST be provided in the JWE header. Use the ECDH-ES algorithm to wrap a random `Content Master Key` to be used for encrypting the signed JWT. The key usage of the respective keys MUST support encryption.

Symmetric Encryption

The symmetric encryption key is derived from the `client_secret` value by using a left truncated SHA-256 hash of the octets of the UTF-8 representation of the `client_secret`. The SHA-256 value MUST be left truncated to the appropriate bit length for the AES key wrapping algorithm used, for instance, to 128 bits for `A128KW`. If a key wrapping key with greater than 256 bits is needed, a different method of deriving the key from the `client_secret` would have to be defined by an extension. Symmetric encryption MUST never be used by public (non-confidential) Clients because of their inability to keep secrets.

See **Section 15.21** for Security Considerations about the need for encrypted requests.

9.4.1. Rotation of Asymmetric Encryption Keys

TOC

Rotating encryption keys is necessarily a different process than for signing keys because the encrypting party starts the process and thus cannot rely on a change in kid as a signal to know that keys need to change. The encrypting party still uses the kid header in the JWE to tell the decrypting party which private key to use to decrypt, however, the encrypting party needs to first select the most appropriate key from those provided in the JWK Set at `jwks_uri`. To rotate keys, the decrypting party can publish new keys at `jwks_uri` and remove from the JWK Set those that are being decommissioned. The `jwks_uri` SHOULD include a `Cache-Control` header in the response that contains a `max-age` directive, as defined in **RFC 2616** [RFC2616], which enables the encrypting party to safely cache the JWK Set and not have to re-retrieve the document for every encryption event. The decrypting party SHOULD remove decommissioned keys from the JWK Set at `jwks_uri` but retain them internally for some reasonable period of time, coordinated with the cache duration, to facilitate a smooth transition between keys by allowing the encrypting party some time to obtain the new keys. The cache duration SHOULD also be coordinated with the issuance of new signing keys as described in **Section 9.3.1**.

---

10. Offline Access

TOC

OpenID Connect defines the following `scope` value to request offline access:

offline_access
: OPTIONAL. This scope value requests that an OAuth 2.0 Refresh Token be issued that can be used to obtain an Access Token that grants access to the End-User's UserInfo Endpoint even when the End-User is not present (not logged in).

When offline access is requested, a `prompt` parameter value of `consent` MUST be used unless other conditions for processing permitting offline access to the requested resources are in place. The OP MUST always obtain consent to returning a Refresh Token that enables offline access to the requested resources. A previously saved user consent is not always sufficient to grant offline access.

Upon receipt of a scope parameter containing the `offline_access` value, the Authorization Server:

- MUST ensure that the prompt parameter contains `consent` unless other conditions for processing permitting offline access to the requested resources are in place; unless one or both of these conditions are fulfilled, then it MUST ignore the `offline_access` request,
- MUST ignore the `offline_access` request unless the Client is using a `response_type` value that would result in an Authorization Code being returned,
- MUST explicitly receive or have consent for all Clients when the registered `application_type` is `web`,
- SHOULD explicitly receive or have consent for all Clients when the registered `application_type` is `native`.

The use of Refresh Tokens is not exclusive to the `offline_access` use case. The Authorization Server MAY grant Refresh Tokens in other contexts that are beyond the scope of this specification.

---

11. Using Refresh Tokens

TOC

A request to the Token Endpoint can also use a Refresh Token by using the `grant_type` value `refresh_token`, as described in Section 6 of **OAuth 2.0** [RFC6749]. This section defines the behaviors for OpenID Connect Authorization Servers when Refresh Tokens are used.

11.1. Refresh Request

TOC

To refresh an Access Token, the Client MUST authenticate to the Token Endpoint using the authentication method registered for its `client_id`, as documented in **Section 8**. The Client sends the parameters via HTTPS `POST` to the Token Endpoint using Form Serialization, per **Section 12.2**.

The following is a non-normative example of a Refresh Request (with line wraps within values for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

client_id=s6BhdRkqt3
  &client_secret=some_secret12345
  &grant_type=refresh_token
  &refresh_token=8xLOxBtZp8
  &scope=openid%20profile
```

The Authorization Server MUST validate the Refresh Token.

11.2. Refresh Successful Response

TOC

Upon receipt of the Refresh Request, the Authorization Server MUST return either a successful response or an error response that corresponds to the received Refresh Token.

Upon successful validation of the Refresh Token, the response body is the Token Response of **Section 2.1.3.3** except that it might not contain an `id_token`.

If an ID Token is returned as a result of a token refresh request, the following requirements apply:

- its `iss` Claim value MUST be the same as in the ID Token issued when the original authentication occurred,
- its `sub` Claim value MUST be the same as in the ID Token issued when the original authentication occurred,
- its `iat` Claim MUST represent the time that the new ID Token is issued,
- its `aud` Claim value MUST be the same as in the ID Token issued when the original authentication occurred,
- if the ID Token contains an `auth_time` Claim, its value MUST represent the time of the original authentication - not the time that the new ID token is issued,
- its `azp` Claim value MUST be the same as in the ID Token issued when the original authentication occurred; if no `azp` Claim was present in the original ID Token, one MUST NOT be present in the new ID Token, and
- otherwise, the same rules apply as apply when issuing an ID Token at the time of the original authentication.

The following is a non-normative example of a Refresh Response:

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

{
 "access_token": "TlBN45jURg",
 "token_type": "Bearer",
 "refresh_token": "9yNOxJtZa5",
 "expires_in": 3600
}
```

11.3. Refresh Error Response

TOC

If the Refresh Request is invalid or unauthorized, the Authorization Server returns the Token Error Response as defined in Section 5.2 of **OAuth 2.0** [RFC6749].

---

12. Serializations

TOC

Messages are serialized using one of the following methods:

1. Query String Serialization

2. Form Serialization
3. JSON Serialization

Not all methods can be used for all messages.

12.1. Query String Serialization

TOC

In order to serialize the parameters using the Query String Serialization, the Client constructs the string by adding the parameters and values to the query component of a URL using the `application/x-www-form-urlencoded` format as defined by **[W3C.REC-html401-19991224]**. Query String Serialization is typically used in HTTP `GET` requests. The same serialization method is also used when adding parameters to the fragment component of a URL.

The following is a non-normative example of this serialization (with line wraps within values for display purposes only):

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

12.2. Form Serialization

TOC

Parameters and their values are Form Serialized by adding the parameter names and values to the entity body of the HTTP request using the `application/x-www-form-urlencoded` format as defined by **[W3C.REC-html401-19991224]**. Form Serialization is typically used in HTTP `POST` requests.

The following is a non-normative example of this serialization (with line wraps within values for display purposes only):

```
POST /authorize HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

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

12.3. JSON Serialization

TOC

The parameters are serialized into a JSON structure by adding each parameter at the

highest structure level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. Each parameter MAY have a JSON structure as its value.

The following is a non-normative example of this serialization:

```
{
 "access_token": "SlAV32hkKG",
 "token_type": "Bearer",
 "expires_in": 3600,
 "refresh_token": "8xLOxBtZp8"
}
```

---

13. String Operations

TOC

Processing some OpenID Connect messages requires comparing values in the messages to known values. For example, the Claim Names returned by the UserInfo Endpoint might be compared to specific Claim Names such as `sub`. Comparing Unicode strings, however, has significant security implications.

Therefore, comparisons between JSON strings and other Unicode strings MUST be performed as specified below:

1. Remove any JSON applied escaping to produce an array of Unicode code points.
2. **Unicode Normalization** [USA15] MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.
3. Comparisons between the two strings MUST be performed as a Unicode code point to code point equality comparison.

In several places, this specification uses space delimited lists of strings. In all such cases, only the ASCII space character (0x20) MAY be used for this purpose.

---

14. Implementation Considerations

TOC

This specification defines features used by both Relying Parties and OpenID Providers. It is expected that some OpenID Providers will require static, out-of-band configuration of RPs using them, whereas others will support dynamic usage by RPs without a pre-established relationship between them. For that reason, the mandatory-to-implement features for OPs are listed below in two groups: the first for all OPs and the second for "Dynamic" OpenID Providers.

14.1. Mandatory to Implement Features for All OpenID Providers

TOC

All OpenID Providers MUST implement the following features defined in this specification. This list augments the set of features that are already listed elsewhere as being "REQUIRED" or are described with a "MUST", and so is not, by itself, a comprehensive set

of implementation requirements for OPs.

Signing ID Tokens with RSA SHA-256
: OPs MUST support signing ID Tokens with the RSA SHA-256 algorithm (an `alg` value of `RS256`).

Prompt Parameter
: OPs MUST support the `prompt` parameter, as defined in **Section 2.1.2**, including the specified user interface behaviors such as `none` and `login`.

Display Parameter
: OPs MUST support the `display` parameter, as defined in **Section 2.1.2**. (Note that the minimum level of support required for this parameter is simply to have its use not result in an error.)

Preferred Locales
: OPs MUST support requests for preferred languages and scripts for the user interface and for Claims via the `ui_locales` and `claims_locales` request parameters, as defined in **Section 2.1.2**. (Note that the minimum level of support required for these parameters is simply to have their use not result in errors.)

Authentication Time
: OPs MUST support returning the time at which the End-User authenticated via the `auth_time` Claim, as defined in **Section 2.1.3.6**.

Maximum Authentication Age
: OPs MUST support enforcing a maximum authentication age via the `max_age` parameter, as defined in **Section 2.1.2**.

Authentication Context Class Reference
: OPs MUST support requests for specific Authentication Context Class Reference values via the `acr_values` parameter, as defined in **Section 2.1.2**. (Note that the minimum level of support required for this parameter is simply to have its use not result in an error.)

14.2. Mandatory to Implement Features for Dynamic OpenID Providers

TOC

In addition to the features listed above, OpenID Providers supporting dynamic establishment of relationships with RPs that they do not have a pre-configured relationship with MUST also implement the following features defined in this and related specifications.

Response Types
: These OpenID Providers MUST support the `id_token` response type and all that are not Self-Issued OPs MUST also support the `id_token token` and `code` response types.

Discovery
: These OPs MUST support Discovery, as defined in **OpenID Connect Discovery 1.0** [OpenID.Discovery].

Dynamic Registration
: These OPs MUST support Dynamic Client Registration, as defined in **OpenID Connect Dynamic Client Registration 1.0** [OpenID.Registration].

UserInfo Endpoint
: All dynamic OPs that issue Access Tokens MUST support the UserInfo Endpoint, as defined in **Section 4.3**. (Self-Issued OPs do not issue Access Tokens.)

Public Keys Published as Bare Keys

These OPs MUST publish their public keys as bare keys, rather than in X.509 format.

Request URI
: These OPs MUST support requests made using a Request Object value that is retrieved from a Request URI that is provided with the `request_uri` parameter, as defined in **Section 2.1.2**.

14.3. Discovery and Registration

TOC

Some OpenID Connect installations can use a pre-configured set of OpenID Providers and/or Relying Parties. In those cases, it might not be necessary to support dynamic discovery of information about identities or services or dynamic registration of Clients.

However, if installations choose to support unanticipated interactions between Relying Parties and OpenID Providers that do not have pre-configured relationships, they SHOULD accomplish this by implementing the facilities defined in the **OpenID Connect Discovery 1.0** [OpenID.Discovery] and **OpenID Connect Dynamic Client Registration 1.0** [OpenID.Registration] specifications.

14.4. Mandatory to Implement Features for Relying Parties

TOC

In general, it is up to Relying Parties which features they use when interacting with OpenID Providers. However, some choices are dictated by the nature of their OAuth Client, such as whether it is a Confidential Client, capable of keeping secrets, in which case the Authorization Code Flow may be appropriate, or whether it is a Public Client, for instance, a User-Agent Based Application or a Native Application, in which case the Implicit Flow may be appropriate.

When using OpenID Connect features, those listed as being "REQUIRED" or are described with a "MUST" are mandatory to implement, when used by a Relying Party. Likewise, those features that are described as "OPTIONAL" need not be used or supported unless they provide value in the particular application context. Finally, when interacting with OpenID Providers that support Discovery, the OP's Discovery document can be used to dynamically determine which OP features are available for use by the RP.

14.5. Compatibility Notes

TOC

14.5.1. Pre-Final IETF Specifications

TOC

Implementers should be aware that the OpenID Connect specifications use several IETF specifications that are not yet final specifications. Those specifications are:

- **JSON Web Token (JWT) draft -12** [JWT]
- **JSON Web Signature (JWS) draft -17** [JWS]
- **JSON Web Encryption (JWE) draft -17** [JWE]

- **JSON Web Key (JWK) draft -17** [JWK]
- **JSON Web Algorithms draft -17** [JWA]
- **Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants draft -12** [OAuth.Assertions]
- **JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants draft -06** [OAuth.JWT]
- **The 'acct' URI Scheme draft -06** [I-D.ietf-appsawg-acct-uri]

While every effort will be made to prevent breaking changes to these specifications, should they occur, OpenID Connect implementations should continue to use the specifically referenced draft versions above in preference to the final versions, unless using a possible future OpenID Connect profile or specification that updates some or all of these references.

14.5.2. Google "iss" Value

TOC

Implementers may want to be aware that, as of the time of this writing, Google's deployed OpenID Connect implementation issues ID Tokens that omit the required `https://` scheme prefix from the `iss` (issuer) Claim value. Relying Party implementations wishing to work with Google will therefore need to have code to work around this, until such time as their implementation is updated. Any such workaround code should be written in a manner that will not break at such point Google adds the missing prefix to their issuer values.

14.6. Related Specifications and Implementer's Guides

TOC

These related OpenID Connect specifications MAY be used in combination with this specification to provide additional functionality:

- **OpenID Connect Discovery 1.0** [OpenID.Discovery] - Dynamic discovery for user and Authorization Server endpoints and information
- **OpenID Connect Dynamic Client Registration 1.0** [OpenID.Registration] - Dynamic registration of OpenID Connect Clients with OpenID Providers
- **OpenID Connect Session Management 1.0** [OpenID.Session] - Session management for OpenID Connect, including logout functionality

These implementer's guides are intended to serve as self-contained references for implementers of basic Web-based Relying Parties:

- **OpenID Connect Basic Client Implementer's Guide 1.0** [OpenID.Basic] - Implementer's guide containing a subset of this specification that is intended for use by basic Web-based Relying Parties using the OAuth Authorization Code Flow
- **OpenID Connect Implicit Client Implementer's Guide 1.0** [OpenID.Implicit] - Implementer's guide containing a subset of this specification that is intended for use by basic Web-based Relying Parties using the OAuth Implicit Flow

---

15. Security Considerations

TOC

This specification references the security considerations defined in Section 10 of **OAuth 2.0** [RFC6749], and Section 5 of **OAuth 2.0 Bearer Token Usage** [RFC6750]. Furthermore, the **OAuth 2.0 Threat Model and Security Considerations** [RFC6819] specification provides an extensive list of threats and controls that apply to this specification as well, given that it is based upon OAuth 2.0. **ISO/IEC 29115** [ISO29115] also provides threats and controls that implementers need to take into account. Implementers are highly advised to read these references in detail and apply the countermeasures described therein.

In addition, the following list of attack vectors and remedies are also considered.

15.1. Request Disclosure

TOC

If appropriate measures are not taken, a request might be disclosed to an attacker, posing security and privacy threats.

In addition to what is stated in Section 5.1.1 of **[RFC6819]**, this standard provides a way to provide the confidentiality of the request end to end through the use of `request` or `request_uri` parameters, where the content of the `request` is an encrypted JWT with the appropriate key and cipher. This protects even against a compromised User-Agent in the case of indirect request.

15.2. Server Masquerading

TOC

A malicious Server might masquerade as the legitimate server using various means. To detect such an attack, the Client needs to authenticate the server.

In addition to what is stated in Section 5.1.2 of **[RFC6819]**, this standard provides a way to authenticate the Server through either the use of Signed or Encrypted JWTs with an appropriate key and cipher.

15.3. Token Manufacture/Modification

TOC

An Attacker might generate a bogus token or modify the token content (such as the authentication or attribute statements) of an existing parseable token, causing the RP to grant inappropriate access to the Client. For example, an Attacker might modify the parseable token to extend the validity period; a Client might modify the parseable token to have access to information that they should not be able to view.

There are two ways to mitigate this attack:

1. The token can be digitally signed by the OP. The Relying Party SHOULD validate the digital signature to verify that it was issued by a legitimate OP.
2. The token can be sent over a protected channel such as TLS. See **Section 15.17** for more information on using TLS. In this specification, the token is always sent over a TLS protected channel. Note however, that this

measure is only a defense against third party attackers and is not applicable to the case where the Client is the attacker.

15.4. Access Token Disclosure

TOC

Access Tokens are credentials used to access Protected Resources, as defined in Section 1.4 of **OAuth 2.0** [RFC6749]. Access Tokens represent a Resource Owner's authorization and MUST NOT be exposed to unauthorized parties.

15.5. Server Response Disclosure

TOC

The server response might contain authentication and attribute statements that include sensitive Client information. Disclosure of the response contents can make the Client vulnerable to other types of attacks.

The server response disclosure can be mitigated in the following two ways:

1. Using the `code` response type. The response is sent over a TLS protected channel, where the Client is authenticated by the `client_id` and `client_secret`.
2. For other response types, the signed response can be encrypted with the Client's public key or a shared secret as an encrypted JWT with an appropriate key and cipher.

15.6. Server Response Repudiation

TOC

A response might be repudiated by the server if the proper mechanisms are not in place. For example, if a Server does not digitally sign a response, the Server can claim that it was not generated through the services of the Server.

To mitigate this threat, the response MAY be digitally signed by the Server using a key that supports non-repudiation. The Client SHOULD validate the digital signature to verify that it was issued by a legitimate Server and its integrity is intact.

15.7. Request Repudiation

TOC

Since it is possible for a compromised or malicious Client to send a request to the wrong party, a Client that was authenticated using only a bearer token can repudiate any transaction.

To mitigate this threat, the Server MAY require that the request be digitally signed by the Client using a key that supports non-repudiation. The Server SHOULD validate the digital signature to verify that it was issued by a legitimate Client and the integrity is intact.

15.8. Access Token Redirect

TOC

An Attacker uses the Access Token generated for one resource to obtain access to a second resource.

To mitigate this threat, the Access Token SHOULD be audience and scope restricted. One way of implementing it is to include the identifier of the resource for whom it was generated as audience. The resource verifies that incoming tokens include its identifier as the audience of the token.

15.9. Token Reuse

TOC

An Attacker attempts to use a one-time use token such as an Authorization Code that has already been used once with the intended Resource. To mitigate this threat, the token SHOULD include a timestamp and a short validity lifetime. The Relying Party then checks the timestamp and lifetime values to ensure that the token is currently valid.

Alternatively, the server MAY record the state of the use of the token and check the status for each request.

15.10. Eavesdropping or Leaking Authorization Codes (Secondary Authenticator Capture)

TOC

In addition to the attack patterns described in Section 4.4.1.1 of **[RFC6819]**, an Authorization Code can be captured in the User-Agent where the TLS session is terminated if the User-Agent is infected by malware. However, capturing it is not useful as long as the profile uses either Client authentication or an encrypted response.

15.11. Token Substitution

TOC

Token Substitution is a class of attacks in which a malicious user swaps various tokens, including swapping an Authorization Code for a legitimate user with another token that the attacker has. One means of accomplishing this is for the attacker to copy a token out one session and use it in an HTTP message for a different session, which is easy to do when the token is available to the browser; this is known as the "cut and paste" attack.

The implicit flow of **OAuth 2.0** [RFC6749] is not designed to mitigate this risk. In Section 10.16, it normatively requires that any use of the authorization process as a form of delegated End-User authentication to the Client MUST NOT use the implicit flow without employing additional security mechanisms that enable the Client to determine whether the Access Token was issued for its use.

In OpenID Connect, this is mitigated through mechanisms provided through the ID Token. The ID Token is a signed security token that provides Claims such as `iss` (issuer), `sub` (subject), `aud` (audience), `azp` (authorized party), `at_hash` (access token hash), and `c_hash` (code hash). Using the ID Token, the Client is capable of detecting the Token Substitution Attack.

The `c_hash` in the ID Token enables Clients to prevent `code` substitution.

Also, a malicious user may attempt to impersonate a more privileged user by subverting the communication channel between the Authorization Endpoint and Client, or the Token Endpoint and Client, for example by swapping the `code` or reordering the messages, to convince the Token Endpoint that the attacker's authorization grant corresponds to a grant sent on behalf of a more privileged user.

For the HTTP binding defined by this specification, the responses to Token Requests are bound to the corresponding requests by message order in HTTP, as both the response containing the token and requests are protected by TLS, which will detect and prevent packet reordering.

When designing another binding of this specification to a protocol incapable of strongly binding Token Endpoint requests to responses, additional mechanisms to address this issue MUST be utilized. One such mechanism could be to include an ID Token with a `c_hash` Claim in the token request and response.

15.12. Timing Attack

TOC

A timing attack enables the attacker to obtain an unnecessary large amount of information through the elapsed time differences in the code paths taken by successful and unsuccessful decryption operations or successful and unsuccessful signature validation of a message. It can be used to reduce the effective key length of the cipher used.

Implementations SHOULD NOT terminate the validation process at the instant of the finding an error but SHOULD continue running until all the octets have been processed to avoid this attack.

15.13. Other Crypto Related Attacks

TOC

There are various crypto related attacks possible depending on the method used for encryption and signature / integrity checking. Implementers need to consult the Security Considerations for the **JWT** [JWT] specification and specifications that it references to avoid the vulnerabilities identified in these specifications.

15.14. Signing and Encryption Order

TOC

Signatures over encrypted text are not considered valid in many jurisdictions. Therefore, for integrity and non-repudiation, this specification requires signing the plain text JSON Claims.

15.15. Issuer Identifier

TOC

OpenID Connect supports multiple issuers per Host and Port combination. The issuer returned by discovery MUST exactly match the value of `iss` in the ID Token.

OpenID Connect treats the path component of any URI as part of the user identifier. For instance, the subject "1234" with an issuer of "https://example.com" is not equivalent to the subject "1234" with an issuer of "https://example.com/sales".

It is RECOMMENDED that only a single issuer per host be used.

15.16. Implicit Grant Flow Threats

TOC

In the implicit grant flow, the Access Token is returned in the fragment component of the Client's `redirect_uri` through HTTPS, thus it is protected between the OP and the User-Agent, and User-Agent and the RP. The only the place it can be captured is the User-Agent where the TLS session is terminated, and is possible if the User-Agent is infested by malware.

15.17. TLS Requirements

TOC

Implementations MUST support TLS. Which version(s) ought to be implemented will vary over time, and depend on the widespread deployment and known security vulnerabilities at the time of implementation. At the time of this writing, TLS version 1.2 **[RFC5246]** is the most recent version, but has very limited actual deployment, and might not be readily available in implementation toolkits. TLS version 1.0 **[RFC2246]** is the most widely deployed version, and will give the broadest interoperability.

To protect against information disclosure and tampering, confidentiality protection MUST be applied using TLS with a ciphersuite that provides confidentiality and integrity protection.

Whenever TLS is used, a TLS server certificate check MUST be performed, per **RFC 6125** [RFC6125].

15.18. Lifetimes of Access Tokens and Refresh Tokens

TOC

Access Token grants are not revocable by the Authorization Server. Access Token grant lifetimes SHOULD be kept to single use or very short lifetimes.

If access to the UserInfo Endpoint or other protected resources is required, a Refresh Token SHOULD be used. The Client MAY then exchange the Refresh Token at the Token Endpoint for a fresh short-lived Access Token that can be used to access the resource.

The Authorization Server SHOULD clearly identify long-term grants to the User during Authorization. The Authorization Server SHOULD provide a mechanism for the End-User to revoke Refresh Tokens granted to a Client.

15.19. Symmetric Key Entropy

TOC

In **Section 9.3** and **Section 9.4**, keys are derived from the `client_secret` value. Thus, when used with symmetric signing or encryption operations, `client_secret` values MUST contain sufficient entropy to generate cryptographically strong keys. Also, `client_secret` values MUST also contain at least the minimum of number of octets required for MAC keys for the particular algorithm used. So for instance, for `HS256`, the `client_secret` value MUST contain at least 8 octets (and almost certainly SHOULD contain more, since `client_secret` values are likely to use a restricted alphabet.

15.20. Need for Signed Requests

TOC

In some situations, Clients might need to use signed requests to ensure that the desired request parameters are delivered to the OP without having been tampered with. For instance, the `max_age` and `acr_values` provide more assurance about the nature of the authentication performed when delivered in signed requests.

15.21. Need for Encrypted Requests

TOC

In some situations, knowing the contents of an OpenID Connect request can, in and of itself, reveal sensitive information about the End-User. For instance, knowing that the Client is requesting a particular Claim or that it is requesting that a particular authentication method be used can reveal sensitive information about the End-User. OpenID Connect enables requests to be encrypted to the OpenID Provider to prevent such potentially sensitive information from being revealed.

---

16. Privacy Considerations

TOC

16.1. Personally Identifiable Information

TOC

The UserInfo Response typically contains Personally Identifiable Information (PII). As such, End-User consent for the release of the information for the specified purpose SHOULD be obtained at or prior to the authorization time in accordance with relevant regulations. The purpose of use is typically registered in association with the `redirect_uris`.

Only necessary UserInfo data should be stored at the Client and the Client SHOULD associate the received data with the purpose of use statement.

16.2. Data Access Monitoring

TOC

The Resource Server SHOULD make the UserInfo access log available to the End-User so that the End-User can monitor who accessed his data.

16.3. Correlation

TOC

To protect the End-User from a possible correlation among Clients, the use of a Pairwise Pseudonymous Identifier (PPID) as the `sub` (subject) SHOULD be considered.

16.4. Offline Access

TOC

Offline access enables access to Claims when the user is not present, posing greater privacy risk than the Claims transfer when the user is present. Therefore, it is prudent to obtain explicit consent for offline access to resources. This specification mandates the use of the `prompt` parameter to obtain consent unless it is a priori known that the request complies with the conditions for processing in each jurisdiction.

When an Access Token is returned in the front channel, there is a greater risk of it being exposed to an attacker, who could later use it to access the UserInfo endpoint. If the Access Token does not enable offline access and the server can differentiate whether the Client request has been made offline or online, the risk will be substantially reduced. Therefore, this specification mandates ignoring the offline access request when the Access Token is transmitted in the front channel. Note that differentiating between online and offline access from the server can be difficult especially for native clients. The server may well have to rely on heuristics. Also, the risk of exposure for the Access Token delivered in the front channel for the response types of `code token` and `token` is the same. Thus, the implementations should be prepared to detect the channel from which the Access Token was issued and deny offline access if the token was issued in the front channel.

Note that although these provisions require an explicit consent dialogue through the `prompt` parameter, the mere fact that the user pressed an "accept" button etc., might not constitute a valid consent. Developers should be aware that for the act of consent to be valid, typically, the impact of the terms have to be understood by the End-User, the consent must be freely given and not forced (i.e., other options have to be available), and the terms must fair and equitable. In general, it is advisable for the service to follow the required privacy principles in each jurisdiction and rely on other conditions of processing than simply explicit consent, as online self-service "explicit consent" often does not form a valid consent in some jurisdictions.

17. IANA Considerations

TOC

17.1. JSON Web Token Claims Registry

TOC

This specification registers the Claims defined in **Section 4.2** and **Section 2.1.3.6** in the IANA JSON Web Token Claims registry defined in **[JWT]**.

17.1.1. Registry Contents

TOC

- Claim Name: `name`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `given_name`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `family_name`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `middle_name`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `nickname`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `preferred_username`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `profile`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `picture`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `website`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `email`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `email_verified`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net

- Specification Document(s): **Section 4.2** of this document

- Claim Name: `gender`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `birthdate`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `zoneinfo`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `locale`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `phone_number`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `address`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `updated_at`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 4.2** of this document

- Claim Name: `azp`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 2.1.3.6** of this document

- Claim Name: `nonce`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 2.1.3.6** of this document

- Claim Name: `auth_time`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 2.1.3.6** of this document

- Claim Name: `at_hash`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net

- Specification Document(s): **Section 2.1.3.6** of this document

- Claim Name: `c_hash`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 2.3.2.11** of this document

- Claim Name: `acr`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 2.1.3.6** of this document

- Claim Name: `amr`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 2.1.3.6** of this document

- Claim Name: `sub_jwk`
- Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification Document(s): **Section 6.4** of this document

---

17.2. OAuth Parameters Registry

TOC

This specification registers the following parameters in the IANA OAuth Parameters registry defined in **RFC 6749** [RFC6749].

17.2.1. Registry Contents

TOC

- Parameter name: `nonce`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2** of this document
- Related information: None

- Parameter name: `display`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2** of this document
- Related information: None

- Parameter name: `prompt`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2** of this document
- Related information: None

- Parameter name: `max_age`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2** of this document
- Related information: None

- Parameter name: `ui_locales`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2** of this document
- Related information: None

- Parameter name: `claims_locales`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 4.4** of this document
- Related information: None

- Parameter name: `id_token_hint`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2** of this document
- Related information: None

- Parameter name: `login_hint`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2** of this document
- Related information: None

- Parameter name: `acr_values`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2** of this document
- Related information: None

- Parameter name: `claims`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 4.5** of this document
- Related information: None

- Parameter name: `registration`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 6.2.1** of this document

- Related information: None

- Parameter name: `request`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 5** of this document
- Related information: None

- Parameter name: `request_uri`
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 5** of this document
- Related information: None

- Parameter name: `id_token`
- Parameter usage location: Authorization Response, Access Token Response
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.3.3** of this document
- Related information: None

17.3. OAuth Extensions Error Registry

TOC

This specification registers the following errors in the IANA OAuth Extensions Error registry defined in **RFC 6749** [RFC6749].

17.3.1. Registry Contents

TOC

- Error name: `interaction_required`
- Error usage location: Authorization Endpoint
- Related protocol extension: OpenID Connect
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2.6** of this document

- Error name: `login_required`
- Error usage location: Authorization Endpoint
- Related protocol extension: OpenID Connect
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2.6** of this document

- Error name: `session_selection_required`
- Error usage location: Authorization Endpoint
- Related protocol extension: OpenID Connect
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net

- Specification document(s): **Section 2.1.2.6** of this document

- Error name: `consent_required`
- Error usage location: Authorization Endpoint
- Related protocol extension: OpenID Connect
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2.6** of this document

- Error name: `invalid_request_uri`
- Error usage location: Authorization Endpoint
- Related protocol extension: OpenID Connect
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2.6** of this document

- Error name: `invalid_request_object`
- Error usage location: Authorization Endpoint
- Related protocol extension: OpenID Connect
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2.6** of this document

- Error name: `request_not_supported`
- Error usage location: Authorization Endpoint
- Related protocol extension: OpenID Connect
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2.6** of this document

- Error name: `request_uri_not_supported`
- Error usage location: Authorization Endpoint
- Related protocol extension: OpenID Connect
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2.6** of this document

- Error name: `registration_not_supported`
- Error usage location: Authorization Endpoint
- Related protocol extension: OpenID Connect
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): **Section 2.1.2.6** of this document

18. References

TOC

18.1. Normative References

TOC

[CORS] Opera Software ASA, "**Cross-Origin Resource Sharing**," July 2010.

[E.164] International Telecommunication Union, "**E.164: The international public telecommunication numbering plan**," 2010.

[IANA.Language] Internet Assigned Numbers Authority (IANA), "**Language Subtag Registry**," 2005.

[ISO29115] International Organization for Standardization, "**ISO/IEC 29115:2013 -- Information technology - Security techniques - Entity authentication assurance framework**," ISO/IEC 29115, March 2013.

[ISO3166-1] International Organization for Standardization, "**ISO 3166-1:1997. Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes**," 1997.

[ISO639-1] International Organization for Standardization, "ISO 639-1:2002. Codes for the representation of names of languages -- Part 1: Alpha-2 code," 2002.

[ISO8601-2004] International Organization for Standardization, "ISO 8601:2004. Data elements and interchange formats - Information interchange - Representation of dates and times," 2004.

[JWA] Jones, M., "**JSON Web Algorithms (JWA)**," draft-ietf-jose-json-web-algorithms (work in progress), October 2013 (**HTML**).

[JWE] Jones, M., Rescorla, E., and J. Hildebrand, "**JSON Web Encryption (JWE)**," draft-ietf-jose-json-web-encryption (work in progress), October 2013 (**HTML**).

[JWK] Jones, M., "**JSON Web Key (JWK)**," draft-ietf-jose-json-web-key (work in progress), October 2013 (**HTML**).

[JWS] Jones, M., Bradley, J., and N. Sakimura, "**JSON Web Signature (JWS)**," draft-ietf-jose-json-web-signature (work in progress), October 2013 (**HTML**).

[JWT] Jones, M., Bradley, J., and N. Sakimura, "**JSON Web Token (JWT)**," draft-ietf-oauth-json-web-token (work in progress), October 2013 (**HTML**).

[OAuth.Assertions] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "**Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants**," draft-ietf-oauth-assertions (work in progress), July 2013 (**HTML**).

[OAuth.JWT] Jones, M., Campbell, B., and C. Mortimore, "**JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants**," draft-ietf-oauth-jwt-bearer (work in progress), July 2013 (**HTML**).

[OAuth.Responses] de Medeiros, B., Ed., Scurtescu, M., Tarjan, P., and M. Jones, "**OAuth 2.0 Multiple Response Type Encoding Practices**," October 2013.

[OpenID.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "**OpenID Connect Discovery 1.0**," October 2013.

[OpenID.Registration] Sakimura, N., Bradley, J., and M. Jones, "**OpenID Connect Dynamic Client Registration 1.0**," October 2013.

[RFC2119] **Bradner, S.**, "**Key words for use in RFCs to Indicate Requirement Levels**," BCP 14, RFC 2119, March 1997 (**TXT**, **HTML**, **XML**).

[RFC2246] **Dierks, T.** and **C. Allen**, "**The TLS Protocol Version 1.0**," RFC 2246, January 1999 (**TXT**).

[RFC2616] **Fielding, R.**, **Gettys, J.**, **Mogul, J.**, **Frystyk, H.**, **Masinter, L.**, **Leach, P.**, and **T. Berners-Lee**, "**Hypertext Transfer Protocol -- HTTP/1.1**," RFC 2616, June 1999 (**TXT**, **PS**, **PDF**, **HTML**, **XML**).

[RFC3339] **Klyne, G., Ed.** and **C. Newman**, "**Date and Time on the Internet: Timestamps**," RFC 3339, July 2002 (**TXT**, **HTML**, **XML**).

[RFC3966] Schulzrinne, H., "**The tel URI for Telephone Numbers**," RFC 3966, December 2004 (**TXT**).

[RFC3986] **Berners-Lee, T.**, **Fielding, R.**, and **L. Masinter**, "**Uniform Resource Identifier (URI): Generic Syntax**," STD 66, RFC 3986, January 2005 (**TXT**, **HTML**, **XML**).

[RFC4627] Crockford, D., "**The application/json Media Type for JavaScript Object Notation (JSON)**," RFC 4627, July 2006 (**TXT**).

[RFC5246] Dierks, T. and E. Rescorla, "**The Transport Layer Security (TLS) Protocol Version 1.2**," RFC 5246, August 2008 (**TXT**).

[RFC5322] **Resnick, P., Ed.**, "**Internet Message Format**," RFC 5322, October 2008 (**TXT**, **HTML**, **XML**).

[RFC5646] Phillips, A. and M. Davis, "**Tags for Identifying Languages**," BCP 47, RFC 5646, September 2009 (**TXT**).

[RFC6125] Saint-Andre, P. and J. Hodges, "**Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)**," RFC 6125, March 2011 (**TXT**).

[RFC6711] Johansson, L., "**An IANA Registry for Level of Assurance (LoA) Profiles**," RFC 6711, August 2012 (**TXT**).

[RFC6749]	Hardt, D., "**The OAuth 2.0 Authorization Framework**," RFC 6749, October 2012 (**TXT**).
[RFC6750]	Jones, M. and D. Hardt, "**The OAuth 2.0 Authorization Framework: Bearer Token Usage**," RFC 6750, October 2012 (**TXT**).
[RFC6819]	Lodderstedt, T., McGloin, M., and P. Hunt, "**OAuth 2.0 Threat Model and Security Considerations**," RFC 6819, January 2013 (**TXT**).
[USA15]	**Davis, M.**, **Whistler, K.**, and M. Dürst, "Unicode Normalization Forms," Unicode Standard Annex 15, 09 2009.
[W3C.REC-html401-19991224]	Hors, A., Raggett, D., and I. Jacobs, "**HTML 4.01 Specification**," World Wide Web Consortium Recommendation REC-html401-19991224, December 1999 (**HTML**).
[zoneinfo]	Public Domain, "**The tz database**," June 2011.

18.2. Informative References

TOC

[I-D.ietf-appsawg-acct-uri]	Saint-Andre, P., "**The 'acct' URI Scheme**," draft-ietf-appsawg-acct-uri-06 (work in progress), July 2013 (**TXT**).
[OpenID.2.0]	OpenID Foundation, "OpenID Authentication 2.0," December 2007 (**TXT**, **HTML**).
[OpenID.Basic]	Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "**OpenID Connect Basic Client Implementer's Guide 1.0**," October 2013.
[OpenID.Implicit]	Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C., and E. Jay, "**OpenID Connect Implicit Client Implementer's Guide 1.0**," October 2013.
[OpenID.PAPE]	**Recordon, D.**, **Jones, M.**, **Bufu, J., Ed.**, **Daugherty, J., Ed.**, and **N. Sakimura**, "OpenID Provider Authentication Policy Extension 1.0," December 2008 (**TXT**, **HTML**).
[OpenID.Session]	Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C., and E. Jay, "**OpenID Connect Session Management 1.0**," October 2013.
[RFC4949]	Shirey, R., "**Internet Security Glossary, Version 2**," RFC 4949, August 2007 (**TXT**).
[X.1252]	International Telecommunication Union, "**ITU-T Recommendation X.1252 -- Cyberspace security -- Identity management -- Baseline identity management terms and definitions**," ITU-T X.1252, November 2010.

Appendix A. Authorization Examples

TOC

The following are non-normative examples of Authorization Requests with differing `response_type` values and their responses (with line wraps within values for display purposes only):

A.1. Example using response_type=code

TOC

```
GET /authorize?
  response_type=code
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile%20email
  &nonce=n-0S6_WzA2Mj
  &state=af0ifjsldkj HTTP/1.1
Host: server.example.com

HTTP/1.1 302 Found
Location: https://client.example.org/cb?
  code=Qcb0Orv1zh30vL1MPRsbm-diHiMwcLyZvn1arpZv-Jxf_11jnpEX3Tgfvk
  &state=af0ifjsldkj
```

A.2. Example using response_type=id_token

TOC

```
GET /authorize?
  response_type=id_token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile%20email
  &nonce=n-0S6_WzA2Mj
  &state=af0ifjsldkj HTTP/1.1
Host: server.example.com

HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  id_token=eyJhbGciOiJSUzI1NiJ9.ew0KICJpc3MiOiAiaHR0cDovL3Nlc
  nZlci5leGFtcGxlLmNvbSIsDQogInN1YiI6ICIyNDgyODk3NjEwMDEiLA0KI
  CJhdWQiOiAiczZCaGRSa3F0MyIsDQogIm5vbmNlIjogIm4tMFM2X1d6QTJNa
  iIsDQogImV4cCI6IDEzMTEyODE5NzAsDQogImlhdCI6IDEzMTEyODA5NzAsD
  QogIm5hbWUiOiAiSmFuZSBEb2UiLA0KICJnaXZlbl9uYW1lIjogIkphbmUiL
  A0KICJmYW1pbHlfbmFtZSI6ICJEb2UiLA0KICJnZW5kZXIiOiAiZmVtYWxlI
  iwNCiAiYmlydGhkYXRlIjogIjAwMDAtMTAtMzEiLA0KICJlbWFpbCI6ICJqY
  W5lZG9lQGV4YW1wbGUuY29tIiwNCiAicGljdHVyZSI6ICJodHRwOi8vZXhhb
  XBsZS5jb20vamFuZWRvZS9tZS5qcGciDQp9.Bgdr1pzosIrnnnpIekmJ7ooe
  DbXuA2AkwfMf90Po2TrMcl3NQzUE_9dcr9r8VOuk4jZxNpV5kCu0RwqqF11-
  6pQ2KQx_ys2i0arLikdResxvJlZzSm_UG6-21s97IaXC97vbnTCcpAkokSe8
  Uik6f8-U61zVmCBMJnpvnxEJllfV8fYldo8lWCqlOngScEbFQUh4fzRsH8O3
  Znr20UZib4V4mGZqYPtPDVGTeu8xkty1t0aK-wEhbm6Hi-TQTi4kltJlw47M
  cSVgF_8SswaGcW6Bf_954ir_ddi4Nexo9RBiWu4n3JMNcQvZU5xMPhu-EF-6
  _nJNotp-lbnBUyxTSg
  &state=af0ifjsldkj
```

Verifying and decoding the ID Token will yield the following Claims:

```
{
 "iss": "http://server.example.com",
 "sub": "248289761001",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "gender": "female",
 "birthdate": "0000-10-31",
 "email": "janedoe@example.com",
 "picture": "http://example.com/janedoe/me.jpg"
}
```

TOC

A.3. Example using response_type=id_token token

```
GET /authorize?
  response_type=id_token%20token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile%20email
  &nonce=n-0S6_WzA2Mj
  &state=af0ifjsldkj HTTP/1.1
Host: server.example.com

HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  access_token=jHkWEdUXMU1BwAsC4vtUsZwnNvTIxEl0z9K3vx5KF0Y
  &token_type=Bearer
  &id_token=eyJhbGciOiJSUzI1NiJ9.ew0KICJpc3MiOiAiaHR0cDovL3NlcnZlc
  i5leGFtcGxlLmNvbSIsDQogInN1YiI6ICIyNDgyODk3NjEwMDEiLA0KICJhdWQiO
  iAiczZCaGRSa3F0MyIsDQogIm5vbmNlIjogIm4tMFM2X1d6QTJNaiIsDQogImV4c
  CI6IDEzMTEyODE5NzAsDQogImlhdCI6IDEzMTEyODA5NzAsDQogImF0X2hhc2giO
  iAiNzdRbVVQdGpQZnpXdEYyQW5wSzlSUSINCn0.g7UR4IDBNIjoPFV8exQCosUNV
  eh8bNUTeL4wdQp-2WXIWnly0_4ZK0sh4A4uddfenzo4Cjh4wuPPrSw6lMeujYbGy
  zKspJrRYL3iiYWc2VQcl8RKdHPz_G-7yf5enut1YE8v7PhKucPJCRRoobMjqD73f
  1nJNwQ9KBrfh21Ggbx1p8hNqQeeLLXb9b63JD84hVOXwyHmmcVgvZskge-wExwnh
  Ivv_cxTzxIXsSxcYlh3d9hnu0wdxPZOGjT0_nNZJxvdIwDD4cAT_LE5Ae447qB90
  ZF89Nmb0Oj2b1GdGVQEIr8-FXrHlyD827f0N_hLYPdZ73YK6p10qY9oRtMimg
  &state=af0ifjsldkj
```

Verifying and decoding the ID Token will yield the following Claims:

```
{
 "iss": "http://server.example.com",
 "sub": "248289761001",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "at_hash": "77QmUPtjPfzWtF2AnpK9RQ"
}
```

A.4. Example using response_type=code id_token

TOC

```
GET /authorize?
  response_type=code%20id_token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile%20email
  &nonce=n-0S6_WzA2Mj
  &state=af0ifjsldkj HTTP/1.1
Host: server.example.com
```

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  code=Qcb0Orv1zh30vL1MPRsbm-diHiMwcLyZvn1arpZv-Jxf_11jnpEX3Tgfvk
  &id_token=eyJhbGciOiJSUzI1NiJ9.ew0KICJpc3MiOiAiaHR0cDovL3NlcnZlc
  i5leGFtcGxlLmNvbSIsDQogInN1YiI6ICIyNDgyODk3NjEwMDEiLA0KICJhdWQiO
  iAiczZCaGRSa3F0MyIsDQogIm5vbmNlIjogIm4tMFM2X1d6QTJNaiIsDQogImV4c
  CI6IDEzMTEyODE5NzAsDQogImlhdCI6IDEzMTEyODA5NzAsDQogImNfaGFzaCI6I
  CJMRGt0S2RvUWFrM1BrMGNuWHhDbHRBIg0KfQ.dAVXerlNOJ_tqMUysD_k1Q_bRX
  RJbLkTOsCPVxpKUis5V6xMRvtjfRg8gUfPuAMYrKQMEqZZmL87Hxkv6cFKavb4ft
  BUrY2qUnrvqe_bNjVEz89QSdxGmdFwSTgFVGWkDf5dV5eIiRxXfIkmlgCltPNocR
  AyvdNrsWC661rHz5F9MzBho2vgi5epUa_KAl6tK4ksgl68pjZqlBqsWfTbGEsWQX
  Efu664dJkdXMLEnsPUeQQLjMhLH7qpZk2ry0nRx0sS1mRwOM_Q0Xmps0vOkNn284
  pMUpmWEAjqklWITgtVYXOzF4ilbmZK6ONpFyKCpnSkAYtTEuqz-m7MoLCD_A
  &state=af0ifjsldkj
```

Verifying and decoding the ID Token will yield the following Claims:

```
{
 "iss": "http://server.example.com",
 "sub": "248289761001",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "c_hash": "LDktKdoQak3Pk0cnXxCltA"
}
```

A.5. Example using response_type=code token

TOC

```
GET /authorize?
  response_type=code%20token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile%20email
  &nonce=n-0S6_WzA2Mj
  &state=af0ifjsldkj HTTP/1.1
Host: server.example.com

HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  code=Qcb0Orv1zh30vL1MPRsbm-diHiMwcLyZvn1arpZv-Jxf_11jnpEX3Tgfvk
  &access_token=jHkWEdUXMU1BwAsC4vtUsZwnNvTIxEl0z9K3vx5KF0Y
  &token_type=Bearer
  &state=af0ifjsldkj
```

A.6. Example using response_type=code id_token token

TOC

```
GET /authorize?
  response_type=code%20id_token%20token
  &client_id=s6BhdRkqt3
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=openid%20profile%20email
  &nonce=n-0S6_WzA2Mj
  &state=af0ifjsldkj HTTP/1.1
Host: server.example.com

HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  code=Qcb0Orv1zh30vL1MPRsbm-diHiMwcLyZvn1arpZv-Jxf_11jnpEX3Tgfvk
  &access_token=jHkWEdUXMU1BwAsC4vtUsZwnNvTIxEl0z9K3vx5KF0Y
  &token_type=Bearer
  &id_token=eyJhbGciOiJSUzI1NiJ9.ew0KICJpc3MiOiAiaHR0cDovL3NlcnZlc
  i5leGFtcGxlLmNvbSIsDQogInN1YiI6ICIyNDgyODk3NjEwMDEiLA0KICJhdWQiO
  iAiczZCaGRSa3F0MyIsDQogIm5vbmNlIjogIm4tMFM2X1d6QTJNaiIsDQogImV4c
  CI6IDEzMTEyODE5NzAsDQogImlhdCI6IDEzMTEyODA5NzAsDQogImF0X2hhc2giO
  iAiNzdRbVVQdGpQZnpXdEYyQW5wSzlSUSIsDQogImNfaGFzaCI6ICJMRGt0S2RvU
  WFrM1BrMGNuWHhDbHRBIg0KfQ.JQthrBsOirujair9aD5gj1Yd5qEv0j4fhLgl8h
  3RaH3soYhwPOiN2Iy_yb7wMCO6I3bPoGJc3zCkpjgUtdB4O2eEhFqXHdwnE4c0oV
  TaTHJi_PdV2ox9g-1ikDB0ckWk0f0SzBd7yM2RoYYxJCiGBQlsSSRQz6ehykonI3
  hLAhXFdpfbK-3_a3HBNKOv_9Mr_JJrz2pqSygk5IBNvwzf1ouVeM91KKvr7EdriK
  N8ysk68fctbFAga1p8rE3cfBOX7Acn4p9QSNpUx0i_x4WHktyKDvH_hLdUw91Fql
  _UOgMP_9h8TYdkAjcq8n1tFzaO7kVaazlZ5SM32J7OSDgNSA
  &state=af0ifjsldkj
```

Verifying and decoding the ID Token will yield the following Claims:

```
{
 "iss": "http://server.example.com",
 "sub": "248289761001",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "at_hash": "77QmUPtjPfzWtF2AnpK9RQ",
 "c_hash": "LDktKdoQak3Pk0cnXxCltA"
}
```

A.7. RSA Key Used in Examples

TOC

The following is the RSA public key in JWK format that can be used to validate the ID Token signatures in the above examples (with line wraps within values for display purposes only):

```
{
 "kty":"RSA",
 "n":"zhEWTBJVTfcUeqnMzOQFMCEVQWOyOUZwP8LrBWh88tKrZyPGCvBkTDp-E2Bzy
      HMQV4pK51Uys2YOwzL9se5THDWMda9rtsCJVcj1V7WaE7wPgl-kIIdWWf4o2g
```

```
      6ZszOy_Fp4q0nG3OTtDRCkBu2iEP21j82pRSRrkCBxnzaChflA7KZbI1n_yhK
      txyA7FdA480LaSVZyKApvrKiYhocACSwf0y6CQ-wkEi6mVXRJt1aBSywlLYA0
      8ojp5hkZQ39eCM2k1EdXdhbar998Q9PZTwXA1cfvuGTZbDWxEKLjMKVuKrT1Y
      vs-2NTXhZAW1KjFS_3UwLkDk-w4dVN-x5tDnw",
 "e":"AQAB"
}
```


Appendix B. Acknowledgements

TOC

As a successor version of OpenID, this specification heavily relies on ideas explored in **OpenID Authentication 2.0** [OpenID.2.0]. Please refer to Appendix C of OpenID Authentication 2.0 for the full list of the contributors for that specification.

In addition, the OpenID Community would like to thank the following people for the work they have done in the drafting and editing of this specification.

Naveen Agarwal (naa@google.com), Google

Amanda Anganes (aanganes@mitre.org), MITRE

Casper Biering (cb@peercraft.com), Peercraft

John Bradley (ve7jtb@ve7jtb.com), Ping Identity

Tim Bray (tbray@textuality.com), Google

Johnny Bufu (jbufu@janrain.com), Janrain

Brian Campbell (bcampbell@pingidentity.com), Ping Identity

Blaine Cook (romeda@gmail.com), Independent

Breno de Medeiros (breno@gmail.com), Google

Pamela Dingle (pdingle@pingidentity.com), Ping Identity

Vladimir Dzhuvinov (vladimir@nimbusds.com), Nimbus Directory Services

George Fletcher (george.fletcher@corp.aol.com), AOL

Roland Hedberg (roland.hedberg@adm.umu.se), University of Umea

Ryo Ito (ryo.ito@mixi.co.jp), mixi, Inc.

Edmund Jay (ejay@mgi1.com), Illumila

Michael B. Jones (mbj@microsoft.com), Microsoft

Torsten Lodderstedt (t.lodderstedt@telekom.de), Deutsche Telekom

Nov Matake (nov@matake.jp), Independent

Chuck Mortimore (cmortimore@salesforce.com), Salesforce

Anthony Nadalin (tonynad@microsoft.com), Microsoft

Hideki Nara (hdknr@ic-tact.co.jp), Tact Communications

Axel Nennker (axel.nennker@telekom.de), Deutsche Telekom

David Recordon (dr@fb.com), Facebook

Justin Richer (jricher@mitre.org), MITRE

Nat Sakimura (n-sakimura@nri.co.jp), Nomura Research Institute, Ltd.

Luke Shepard (lshepard@fb.com), Facebook

Andreas Akre Solberg (andreas.solberg@uninett.no), UNINET

Paul Tarjan (pt@fb.com), Facebook

Appendix C. Notices

TOC

Appendix D. Document History

TOC

[[To be removed from the final specification]]

-14

- Fixed #862 - Clarified `azp` definition.
- Fixed #866 - Stated that the behavior is unspecified if the `acr` Claim is requested with both the `acr_values` request parameter and an individual claim request listing requested values.
- Fixed #867 - Allow ID Tokens to use "alg":"none" when using the Authorization Code Flow and when explicitly requested at registration time.
- Fixed #869 - Registered the OAuth error codes `request_not_supported`, `request_uri_not_supported`, and `registration_not_supported`.
- Fixed #874 - Said more about frame busting.
- Fixed #877 - Specified that a user interface MUST NOT be displayed when `prompt=none` is used.
- Fixed #878 - Defined negative response for "id_token_hint".
- Updated the description of the plans to host the site `https://self-issued.me/`, per tasks #879 and #880.
- Fixed #876 - Described that Google's `iss` value currently omits the required `https://` scheme prefix.
- Fixed #882 - Called out pre-final IETF specifications used.
- Fixed #884 - Changed the descriptions of Basic and Implicit from being profiles to being implementer's guides containing subsets of OpenID Connect Core.

-13

- Restructured to separate Authentication from other features and to have separate Authentication sections for the Authorization Code Flow, the Implicit Flow, and the Hybrid Flow. The validation procedures for steps are now specified immediately following the descriptions of those steps.
- Completed restructuring into functional sections.

-12

- Created the new OpenID Connect Core specification by combining OpenID Connect Messages draft 20 and OpenID Connect Standard draft 21, with no normative changes. (These versions are the second Implementer's Drafts.)

Authors' Addresses

TOC

Nat Sakimura
Nomura Research Institute, Ltd.
Email: n-sakimura@nri.co.jp
URI: http://nat.sakimura.org/

John Bradley
Ping Identity
Email: ve7jtb@ve7jtb.com
URI: http://www.thread-safe.com/

Michael B. Jones
Microsoft
Email: mbj@microsoft.com
URI: http://self-issued.info/

Breno de Medeiros
Google
Email: breno@google.com

URI: http://stackoverflow.com/users/311376/breno

Chuck Mortimore
Salesforce
Email: cmortimore@salesforce.com
URI: https://twitter.com/cmort

5-1 Oct 20, 2013, 5:27 PM, George Fletcher
This wording doesn't flow well. I should suggest something better.

6-1 Oct 20, 2013, 5:27 PM, George Fletcher
Suggest: what the entity knows, possesses, behavior patterns, has as physical features, or combinations of these utilizing heuristics.

7-1 Oct 20, 2013, 5:27 PM, George Fletcher
Is the intent that the whole Issuer Identifier is case sensitive? Or just the path component as per normal URLs?

7-2 Oct 20, 2013, 5:27 PM, George Fletcher
Maybe a forward reference to 2.1.3.6 would be helpful here?

9-1 Oct 20, 2013, 5:27 PM, George Fletcher
Subject identifier is not capitalized. Should it be?

10-1 Oct 20, 2013, 5:27 PM, George Fletcher
The -> a

10-2 Oct 20, 2013, 5:27 PM, George Fletcher
This special exception is confusing. I almost wonder if it could be added to the security considerations and then the text here is... MUST except for the case x.x.x.x in Security Considerations. Another case where the will not be http or https is a mobile client implementing the code flow.

10-3 Oct 20, 2013, 5:27 PM, George Fletcher
Is this use of 'nonce' in addition to that described in the validation steps for the hybrid flow? Or a different method of doing the same thing?

12-1 Oct 20, 2013, 5:27 PM, George Fletcher
This is the first use of 'audience' in conjunction with the id_token. It might not make sense to someone just reading the specs without any other context. Maybe add a reference to the id_token processing rules section?

13-1 Oct 20, 2013, 5:27 PM, George Fletcher
This is confusing. If I specify a id_token_hint and ask for the 'sub' claim then the AS must not response with a successful response if the user doesn't match the id_token_hint. However, if I don't ask for a sub claim then the AS can return an successful response where the id_token doesn't match the id_token_hint?

15-1 Oct 20, 2013, 5:27 PM, George Fletcher
What case is this covering that isn't already covered by the other *_required error codes? Is my OP compliant if I only return I the interaction_required error even if the case is a login_required?

15-2 Oct 20, 2013, 5:27 PM, George Fletcher
Not sure the reg error makes sense.

22-1 Oct 20, 2013, 5:27 PM, George Fletcher
Why not? Wouldn't the secret associated with the azp work for the client to validate the id_token?

If we want interoperability across the use of audience and azp we are going to need to describe how it works in an extension document. It is not clear from this spec how it is to work and I was on most of the calls:)

23-1 Oct 20, 2013, 5:27 PM, George Fletcher
The -> in the

24-1 Oct 20, 2013, 5:27 PM, George Fletcher
We say the same thing three different times. Once in 2.2.2 and twice in 2.2.2.1

24-2 Oct 20, 2013, 5:27 PM, George Fletcher
I'm not sure we got these examples correct. Native application can be both mobile or rich desktop. In the mobile case it is most likely the scheme will not be http related at all. I suppose in either case the client could be running a local web server and use it to load the JS to process the fragment. Maybe the real question is wether local host should be allowed in the code flow.

24-3 Oct 20, 2013, 5:27 PM, George Fletcher
I'm not sure this suggestion makes sense for the implicit flow. The client would need to write a cookie value on the domain of the redirect_uri and the attempt to read it on the return of the implicit flow. Wondering if a local storage example would make more sense.

25-1 Oct 20, 2013, 5:27 PM, George Fletcher
Did we chose the negative constraint here to leave the door open for other types? If not a positive constraint is easier to understand. Something like "this is only returned when the response_type is 'id_token token'"

29-1 Oct 20, 2013, 5:27 PM, George Fletcher
I don't think this is necessary as 'id_token' is not one of the allowed response_type values. Or maybe it's supposed to be 'code id_token'?

32-1 Oct 20, 2013, 5:27 PM, George Fletcher
Why not also the 'code token' flow?

34-1 Oct 20, 2013, 5:27 PM, George Fletcher
Why isn't the id_token returned in the 'code token' case as the scope requires an 'openid' value which ensures that the response from the token endpoint includes an id_token.