International Standard

ISO/IEC 26131

First edition
2024-10

Information technology — OpenID connect — OpenID connect core 1.0 incorporating errata set 2

Reference number
ISO/IEC 26131:2024(en)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by the OpenID Foundation (OIDF) (as OpenID Connect Core 1.0 incorporating errata set 2) and drafted in accordance with its editorial rules. It was adopted, under the JTC 1 PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

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

Information technology — OpenID Connect — OpenID Connect Core 1.0 incorporating errata set 2

1. Introduction

TOC

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 [RFC6749] 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. Readers are expected to be familiar with these specifications.

OpenID Connect implements authentication as an extension to the OAuth 2.0 authorization process. Use of this extension is requested by Clients by including the `openid` scope value in the Authorization Request. Information about the authentication performed is returned in a JSON Web Token (JWT) [JWT] called an ID Token (see Section 2). OAuth 2.0 Authentication Servers implementing OpenID Connect are also referred to as OpenID Providers (OPs). OAuth 2.0 Clients using OpenID Connect are also referred to as Relying Parties (RPs).

This specification assumes that the Relying Party has already obtained configuration information about the OpenID Provider, including its Authorization Endpoint and Token Endpoint locations. This information is 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 Relying Party has already obtained sufficient credentials and provided information needed to use the OpenID Provider. This is normally done via Dynamic Registration, as described in OpenID Connect Dynamic Client Registration 1.0 [OpenID.Registration], or may be obtained via other mechanisms.

The previous versions of this specification are:

- OpenID Connect Core 1.0 incorporating errata set 1 [OpenID.Core.Errata1]
- OpenID Connect Core 1.0 (final) [OpenID.Core.Final]

1.1. Requirements Notation and Conventions

TOC

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

In the .txt version of this specification, 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. In the HTML version of this specification, values to be taken literally are indicated by the use of `this fixed-width font`.

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 specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Grant", "Authorization Server", "Client", "Client Authentication", "Client Identifier", "Client Secret", "Grant Type", "Protected Resource", "Redirection URI", "Refresh Token", "Resource Server", "Response Type", and "Token Endpoint" defined by OAuth 2.0 [RFC6749], the terms "Claim Name", "Claim Value", "JSON Web Token (JWT)", "JWT Claims Set", and "Nested JWT" defined by JSON Web Token (JWT) [JWT], the terms "Base64url Encoding", "Header Parameter", and "JOSE Header" defined by JSON Web Signature (JWS) [JWS], the term "User Agent" defined by RFC 7230 [RFC7230], and the term "Response Mode" defined by OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses].

This specification also defines the following terms:

Authentication

Process used to achieve sufficient confidence in the binding between the Entity and the presented Identity.

Authentication Request

OAuth 2.0 Authorization Request using extension parameters and scopes defined by OpenID Connect to request that the End-User be authenticated by the Authorization Server, which is an OpenID Connect Provider, to the Client, which is an OpenID Connect Relying Party.

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 an Authorization Code is returned from the Authorization Endpoint and all tokens are returned from the Token Endpoint.

Authorization Request

OAuth 2.0 Authorization Request as defined by [RFC6749].

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 participant.

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 an Authorization Code is returned from the Authorization Endpoint, 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 Flow

OAuth 2.0 flow in which all tokens are returned from the Authorization Endpoint and neither the Token Endpoint nor an Authorization Code are used.

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 Authenticating the End-User and providing Claims to a Relying Party about the Authentication event and the End-User.

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 End-User Authentication and Claims from an OpenID Provider.

Sector Identifier

Host component of a URL used by the Relying Party's organization that is an input to the computation of pairwise Subject Identifiers for that Relying Party.

Self-Issued OpenID Provider

Personal, self-hosted OpenID Provider that issues self-signed ID Tokens.

Subject Identifier

Locally unique and never reassigned identifier within the Issuer for the End-User, which is intended to be consumed by the Client.

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. The UserInfo Endpoint URL MUST use the `https` scheme and MAY contain port, path, and query parameter components.

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.

IMPORTANT NOTE TO READERS: The terminology definitions in this section are 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.

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].

TOC

1.3. Overview

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

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

These steps are illustrated in the following diagram:

```
+--------+                                   +--------+
|        |                                   |        |
|        |---------(1) AuthN Request-------->|        |
|        |                                   |        |
|        |  +--------+                       |        |
|        |  |        |                       |        |
|        |  |  End-  |<--(2) AuthN & AuthZ-->|        |
|        |  |  User  |                       |        |
|   RP   |  |        |                       |   OP   |
|        |  +--------+                       |        |
|        |                                   |        |
|        |<--------(3) AuthN Response--------|        |
|        |                                   |        |
|        |---------(4) UserInfo Request----->|        |
|        |                                   |        |
|        |<--------(5) UserInfo Response-----|        |
|        |                                   |        |
+--------+                                   +--------+
```

2. ID Token

TOC

The primary extension that OpenID Connect makes to OAuth 2.0 to enable End-Users to be Authenticated is the ID Token data structure. The ID Token is a security token that contains Claims about the Authentication of an End-User by an Authorization Server when using a Client, and potentially other requested Claims. The ID Token is represented as a JSON Web Token (JWT) [JWT].

The following Claims are used within the ID Token for all OAuth 2.0 flows used by OpenID Connect:

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 [RFC20] 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 common 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 by the RP when performing authentication with the OP. 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. Its value is a JSON [RFC8259] number representing the number of seconds from 1970-01-01T00:00:00Z as measured in UTC until the date/time. See RFC 3339 [RFC3339] for details regarding date/times in general and UTC in particular. NOTE: The ID Token expiration time is unrelated the lifetime of the authenticated session between the RP and the OP.

iat

REQUIRED. Time at which the JWT was issued. Its value is a JSON number representing the number of seconds from 1970-01-01T00:00:00Z as measured in UTC until the date/time.

auth_time

Time when the End-User authentication occurred. Its value is a JSON number representing the number of seconds from 1970-01-01T00:00:00Z 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; otherwise, its inclusion is OPTIONAL. (The `auth_time` Claim semantically corresponds to the OpenID 2.0 PAPE [OpenID.PAPE] `auth_time` response parameter.)

nonce

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 Authentication 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 Authentication Request. If present in the Authentication Request, Authorization Servers MUST include a `nonce` Claim in the ID Token with the Claim Value being the nonce value sent in the Authentication Request. Authorization Servers SHOULD perform no other processing on `nonce` values used. The `nonce` 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. For historic reasons, the value "0" is used to indicate that there is no confidence that the same person is actually there. Authentications with level 0 SHOULD NOT 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 an RFC 6711 [RFC6711] registered name 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 `amr` value is an array of case-sensitive strings. Values used in the `amr` Claim SHOULD be from those registered in the IANA Authentication Method Reference Values registry [IANA.AMR] established by [RFC8176]; parties using this claim will need to agree upon the meanings of any unregistered values used, which may be context specific.

azp

OPTIONAL. 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. The `azp` value is a case-sensitive string containing a StringOrURI value. Note that in practice, the `azp` Claim only occurs when extensions beyond the scope of this specification are used; therefore, implementations not using such extensions are encouraged to not use `azp` and to ignore it when it does occur.

ID Tokens MAY contain other Claims. Any Claims used that are not understood MUST be ignored. See Sections 3.1.3.6, 3.3.2.11, 5.1, and 7.4 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 16.14. If the ID Token is encrypted, it MUST be signed then encrypted, with the result being a Nested JWT, as defined in [JWT]. ID Tokens MUST NOT use `none` as the `alg` value unless the Response Type used returns no ID Token from the Authorization Endpoint (such as when using 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, references to keys used are communicated in advance using Discovery and Registration parameters, per Section 10.

The following is a non-normative example of the set of Claims (the JWT Claims Set) in an ID Token:

```
{
 "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"
}
```

3. Authentication

TOC

OpenID Connect performs authentication to log in the End-User or to determine that the End-User is already logged in. OpenID Connect returns the result of the Authentication performed by the Server to the Client in a secure manner so that the Client can rely on it. For this reason, the Client is called Relying Party (RP) in this case.

The Authentication result is returned in an ID Token, as defined in Section 2. It has Claims expressing such information as the Issuer, the Subject Identifier, when the authentication was performed, etc.

Authentication can follow one of three paths: the Authorization Code Flow (`response_type=code`), the Implicit Flow (`response_type=id_token token` or `response_type=id_token`), or the Hybrid Flow (using other Response Type values defined in OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses]). The flows determine how the ID Token and Access Token are returned to the Client.

The characteristics of the three flows are summarized in the following non-normative table. The table is intended to provide some guidance on which flow to choose in particular contexts.

Property	Authorization Code Flow	Implicit Flow	Hybrid Flow
All tokens returned from Authorization Endpoint	no	yes	no
All tokens returned from Token Endpoint	yes	no	no
Tokens not revealed to User Agent	yes	no	no
Client can be authenticated	yes	no	yes
Refresh Token possible	yes	no	yes
Communication in one round trip	no	yes	no
Most communication server-to-server	yes	no	varies

OpenID Connect Authentication Flows

The flow used is determined by the `response_type` value contained in the Authorization Request. These `response_type` values select these flows:

"response_type" value	Flow
`code`	Authorization Code Flow
`id_token`	Implicit Flow
`id_token token`	Implicit Flow
`code id_token`	Hybrid Flow
`code token`	Hybrid Flow
`code id_token token`	Hybrid Flow

OpenID Connect "response_type" Values

All but the `code` Response Type value, which is defined by OAuth 2.0 [RFC6749], are defined in the OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses] specification. NOTE: 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 would be returned.

3.1. 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 ID Token and an Access Token directly. This provides the benefit of not exposing any tokens to the User Agent and possibly other malicious applications with access to the 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.

3.1.1. Authorization Code Flow Steps

TOC

The Authorization Code Flow goes through the following steps.

1. Client prepares an Authentication Request containing the desired request parameters.
2. Client sends the 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 ID token and retrieves the End-User's Subject Identifier.

3.1.2. Authorization Endpoint

TOC

The Authorization Endpoint performs Authentication of the End-User. This is done by sending the User Agent to the Authorization Server's Authorization Endpoint for Authentication and Authorization, using request parameters defined by OAuth 2.0 and additional parameters and parameter values defined by OpenID Connect.

Communication with the Authorization Endpoint MUST utilize TLS. See Section 16.17 for more information on using TLS.

3.1.2.1. Authentication Request

TOC

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

Authorization Servers MUST support the use of the HTTP `GET` and `POST` methods defined in RFC 7231 [RFC7231] 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 13.1. If using the HTTP `POST` method, the request parameters are serialized using Form Serialization, per Section 13.2.

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

scope

REQUIRED. OpenID Connect requests MUST contain the `openid` scope value. If the `openid` scope value is not present, the behavior is entirely unspecified. Other scope values MAY be present. Scope values used that are not understood by an implementation SHOULD be ignored. See Sections 5.4 and 11 for additional scope values defined by this specification.

response_type

REQUIRED. OAuth 2.0 Response Type value that determines the authorization processing flow to be used, including what parameters are returned from the endpoints used. When using the Authorization Code Flow, this value is `code`.

client_id

REQUIRED. OAuth 2.0 Client Identifier valid at the Authorization Server.

redirect_uri

REQUIRED. Redirection URI to which the response will be sent. This URI MUST exactly match one of the Redirection URI values for the Client pre-registered at the OpenID Provider, with the matching performed as described in Section 6.2.1 of [RFC3986] (Simple String Comparison). When using this flow, the Redirection URI SHOULD use the `https` scheme; however, it MAY use the `http` scheme, provided that the Client Type is `confidential`, as defined in Section 2.1 of OAuth 2.0, and provided the OP allows the use of `http` Redirection URIs in this case. Also, if the Client is a native application, it MAY use the `http` scheme with `localhost` or the IP loopback literals `127.0.0.1` or `[::1]` as the hostname. The Redirection URI MAY use an alternate scheme, such as one that is intended to identify a callback into a native application.

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 a browser cookie.

OpenID Connect also uses the following OAuth 2.0 request parameter, which is defined in OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses]:

response_mode

OPTIONAL. Informs the Authorization Server of the mechanism to be used for returning parameters from the Authorization Endpoint. This use of this parameter is NOT RECOMMENDED when the Response Mode that would be requested is the default mode specified for the Response Type.

This specification also defines the following request parameters:

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 Authentication Request to the ID Token. Sufficient entropy MUST be present in the `nonce` values used to prevent attackers from guessing values. For implementation notes, see Section 15.5.2.

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 the 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 the authentication and consent UI consistent with a popup User Agent window. The popup User Agent window should be of an appropriate size for a login-focused dialog and should not obscure the entire window that it is popping up over.

touch

The Authorization Server SHOULD display the authentication and consent UI consistent with a device that leverages a touch interface.

wap

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

The Authorization Server MAY also attempt to detect the capabilities of the User Agent and present an appropriate display.

If an OP receives a `display` value outside the set defined above that it does not understand, it MAY return an error or it MAY ignore it; in practice, not returning errors for not-understood values will help facilitate phasing in extensions using new `display` values.

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. An error is returned if an 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 the request. The error code will typically be `login_required`, `interaction_required`, or another code defined in Section 3.1.2.6. 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 reauthenticate the End-User, it MUST return an error, typically `login_required`.

consent

The Authorization Server SHOULD prompt the End-User for consent before returning information to the Client. If it cannot obtain consent, it MUST return an error, typically `consent_required`.

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 obtain an account selection choice made by the End-User, it MUST return an error, typically `account_selection_required`.

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.

If an OP receives a `prompt` value outside the set defined above that it does not understand, it MAY return an error or it MAY ignore it; in practice, not returning errors for not-understood values will help facilitate phasing in extensions using new `prompt` values.

max_age

OPTIONAL. Maximum Authentication Age. Specifies the allowable elapsed time in seconds since the last time the End-User was actively authenticated by the OP. 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. Note that `max_age=0` is equivalent to `prompt=login`.

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. ID Token previously issued by the Authorization Server being passed 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 already logged in or is logged in as a result of the request (with the OP possibly evaluating other information beyond the ID Token in this decision), then the Authorization Server returns a positive response; otherwise, it MUST return an error, such as `login_required`. 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 from the OP is encrypted, to use it as an `id_token_hint`, 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 and use the re-encrypted ID token as the `id_token_hint` value.

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.

The acr Claim is requested as a Voluntary Claim by this parameter.

Other parameters MAY be sent. See Sections 3.2.2, 3.3.2, 5.2, 5.5, 6, and 7.2.1 for additional Authorization Request parameters and parameter values defined by this specification.

The following is a non-normative example HTTP 302 redirect response by the Client, which triggers the User Agent to make an Authentication Request to the Authorization Endpoint (with line wraps within values for display purposes only):

```
  HTTP/1.1 302 Found
  Location: https://server.example.com/authorize?
    response_type=code
    &scope=openid%20profile%20email
    &client_id=s6BhdRkqt3
    &state=af0ifjsldkj
    &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
```

The following is the non-normative example request that would be sent by the User Agent to the Authorization Server in response to the HTTP 302 redirect response by the Client above (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
```

3.1.2.2. Authentication 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. Verify that a scope parameter is present and contains the openid scope value. (If no openid scope value is present, the request may still be a valid OAuth 2.0 request but is not an OpenID Connect request.)

3. The Authorization Server MUST verify that all the REQUIRED parameters are present and their usage conforms to this specification.
4. 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 the End-User identified by that `sub` value has an active session with the Authorization Server or has been Authenticated as a result of the request. 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 5.5.1, if the `claims` parameter is supported by the implementation.
5. When an `id_token_hint` is present, the OP MUST validate that it was the issuer of the ID Token. The OP SHOULD accept ID Tokens when the RP identified by the ID Token has a current session or had a recent session at the OP, even when the `exp` time has passed.

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, per Section 3.1.2.6.

3.1.2.3. Authorization Server Authenticates End-User

TOC

If the request is valid, the Authorization Server attempts to Authenticate the End-User or determines whether the End-User is Authenticated, depending upon the request parameter values used. The methods used by the Authorization Server to Authenticate 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 Authenticate the End-User in the following cases:

- The End-User is not already Authenticated.
- The Authentication 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 Authentication Request contains the `prompt` parameter with the value `none`. In this case, the Authorization Server MUST return an error if an End-User is not already Authenticated or could not be silently Authenticated.

When interacting with the End-User, 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].

3.1.2.4. Authorization Server Obtains End-User Consent/Authorization

TOC

Once the End-User is authenticated, the Authorization Server MUST obtain an authorization decision before releasing information to the Relying Party. 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 the request or other means (for example, via previous administrative consent). Sections 2 and 5.3 describe information release mechanisms.

3.1.2.5. Successful Authentication Response

TOC

An Authentication Response is an OAuth 2.0 Authorization Response message returned from the OP's Authorization Endpoint in response to the Authorization Request message sent by the RP.

When using the Authorization Code Flow, the Authorization Response MUST return the parameters defined in Section 4.1.2 of OAuth 2.0 [RFC6749] by adding them as query parameters to the `redirect_uri` specified in the Authorization Request using the `application/x-www-form-urlencoded` format, unless a different Response Mode was specified.

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
```

For implementation notes on the contents of the Authorization Code, see Section 15.5.1.

3.1.2.6. Authentication Error Response

TOC

An Authentication Error Response is an OAuth 2.0 Authorization Error Response message returned from the OP's Authorization Endpoint in response to the Authorization Request message sent by the RP.

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 Section 4.1.2.1 of OAuth 2.0 [RFC6749]. (HTTP errors unrelated to RFC 6749 are returned to the User Agent using the appropriate HTTP status code.)

Unless the Redirection URI is invalid, the Authorization Server returns the Client to the Redirection URI specified in the Authorization Request with the appropriate error and state parameters. Other parameters SHOULD NOT be returned. If the Redirection URI is invalid, the Authorization Server MUST NOT redirect the User Agent to the invalid Redirection URI.

If the Response Mode value is not supported, the Authorization Server returns an HTTP response code of 400 (Bad Request) without Error Response parameters, since understanding the Response Mode is necessary to know how to return those parameters.

In addition to the error codes defined in Section 4.1.2.1 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 value in the Authentication Request is `none`, but the Authentication 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 value in the Authentication Request is `none`, but the Authentication Request cannot be completed without displaying a user interface for End-User authentication.

account_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 value in the Authentication Request is `none`, but the Authentication 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 value in the Authentication Request is `none`, but the Authentication 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 defined in Section 6.

request_uri_not_supported

The OP does not support use of the `request_uri` parameter defined in Section 6.

registration_not_supported

The OP does not support use of the `registration` parameter defined in Section 7.2.1.

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 error response parameters are added to the query component of the Redirection URI, unless a different Response Mode was specified.

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=
    Unsupported%20response_type%20value
  &state=af0ifjsldkj
```

3.1.2.7. Authentication 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.

3.1.3. Token Endpoint

TOC

To obtain an Access Token, an ID Token, and optionally a Refresh Token, the RP (Client) sends a Token Request to the Token Endpoint to obtain a Token Response, as described in Section 3.2 of OAuth 2.0 [RFC6749], when using the Authorization Code Flow.

Communication with the Token Endpoint MUST utilize TLS. See Section 16.17 for more information on using TLS.

3.1.3.1. Token Request

TOC

A Client makes a Token Request by presenting its Authorization Grant (in the form of an Authorization Code) to the Token Endpoint using the `grant_type` value `authorization_code`, as described in Section 4.1.3 of OAuth 2.0 [RFC6749]. If the Client is a Confidential Client, then it MUST authenticate to the Token Endpoint using the authentication method registered for its `client_id`, as described in Section 9.

The Client sends the parameters to the Token Endpoint using the HTTP `POST` method and the Form Serialization, per Section 13.2, 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=SplxlOBeZQQYbYS6WxSbI
A
    &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
```

3.1.3.2. Token Request Validation

TOC

The Authorization Server MUST validate the Token Request as follows:

- Authenticate the Client if it was issued Client Credentials or if it uses another Client Authentication method, per Section 9.
- Ensure the Authorization Code was issued to the authenticated Client.
- Verify that the Authorization Code is valid.
- If possible, verify that the Authorization Code has not been previously used.
- 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).
- Verify that the Authorization Code used was issued in response to an OpenID Connect Authentication Request (so that an ID Token will be returned from the Token Endpoint).

3.1.3.3. Successful Token Response

TOC

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]. The response uses the `application/json` media type.

The OAuth 2.0 token_type response parameter value MUST be Bearer, as specified in OAuth 2.0 Bearer Token Usage [RFC6750], unless another Token Type has been negotiated with the Client. Servers SHOULD support the Bearer Token Type; use of other Token Types is outside the scope of this specification. Note that the token_type value is case insensitive.

In addition to the response parameters specified by OAuth 2.0, the following parameters MUST be included in the response:

id_token

ID Token value associated with the authenticated session.

All Token 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

HTTP Response Headers and Values

The following is a non-normative example of a successful Token Response. The ID Token signature in the example can be verified with the key at Appendix A.7.

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

  {
   "access_token": "SlAV32hkKG",
   "token_type": "Bearer",
   "refresh_token": "8xLOxBtZp8",
   "expires_in": 3600,
   "id_token":
"eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzc
yI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiM
jQ4Mjg5

NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsCiAibm9uY2UiOiAi
bi0wUzZ

fV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTE
yODA5Nz

AKfQ.ggW8hZ1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU9
6Sr_P6q
     Jp6IcmD3HP99Obi1PRs-cwh3LO-
p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJ
     NqeGpe-
gccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiOtX7Tpd

QyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJ
bOEoRoS

K5hoDalrcvRYLSrQAZZKflyuVCyixEoV9GfNQC3_osjzw2PAithfubEE
BLuVVk4
     XUVrWOLrLl0nx7RkKU8NXNHq-rvKMzqg"
  }
```

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

3.1.3.4. 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

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

3.1.3.5. 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 3.1.3.7.
3. Follow the Access Token validation rules in Section 3.1.3.8.

3.1.3.6. ID Token

TOC

The contents of the ID Token are as described in Section 2. When using the Authorization Code Flow, these additional requirements for the following ID Token Claims apply:

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` Header Parameter of the ID Token's JOSE Header. For instance, if the `alg` is `RS256`, hash the `access_token` value with SHA-256, then take the left-most 128 bits and base64url-encode them. The `at_hash` value is a case-sensitive string.

3.1.3.7. ID Token Validation

TOC

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

1. If the ID Token is encrypted, decrypt it using the keys and algorithms that the Client specified during Registration that the OP was to use to encrypt the ID Token. If encryption was negotiated with the OP at Registration time and the ID Token is not encrypted, the RP SHOULD reject it.

2. The Issuer Identifier for the OpenID Provider (which is typically obtained during Discovery) MUST exactly match the value of the `iss` (issuer) Claim.

3. 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.

4. If the implementation is using extensions (which are beyond the scope of this specification) that result in the `azp` (authorized party) Claim being present, it SHOULD validate the `azp` value as specified by those extensions.

5. This validation MAY include that when an `azp` (authorized party) Claim is present, the Client SHOULD verify that its `client_id` is the Claim Value.

6. If the ID Token is received via direct communication between the Client and the Token Endpoint (which it is in this flow), 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 JWT `alg` Header Parameter. The Client MUST use the keys provided by the Issuer.

7. 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.

8. If the JWT `alg` Header Parameter uses a MAC based algorithm such as `HS256`, `HS384`, or `HS512`, the octets of the UTF-8 [RFC3629] 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. For MAC based algorithms, the behavior is unspecified if the `aud` is multi-valued.

9. The current time MUST be before the time represented by 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 Authentication 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 Authentication 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.

3.1.3.8. 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 in the same manner as for the Implicit Flow, as defined in Section 3.2.2.9, but using the ID Token and Access Token returned from the Token Endpoint.

3.2. 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 End-User and applications that have access to the End-User's User Agent. The Authorization Server does not perform Client Authentication.

3.2.1. Implicit Flow Steps

TOC

The Implicit Flow follows the following steps:

1. Client prepares an Authentication Request containing the desired request parameters.
2. Client sends the 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 ID token and retrieves the End-User's Subject Identifier.

3.2.2. 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 3.1.2, with the exception of the differences specified in this section.

3.2.2.1. Authentication Request

TOC

Authentication Requests are made as defined in Section 3.1.2.1, except that these Authentication Request parameters are used as follows:

response_type

REQUIRED. OAuth 2.0 Response Type value that determines the authorization processing flow to be used, including what parameters are returned from the endpoints used. When using the Implicit Flow, this value is `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: 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 would be returned.

redirect_uri

REQUIRED. Redirection URI to which the response will be sent. This URI MUST exactly match one of the Redirection URI values for the Client pre-registered at the OpenID Provider, 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 or the IP loopback literals 127.0.0.1 or [::1] as the hostname.

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 Authentication Request to the ID Token. Sufficient entropy MUST be present in the nonce values used to prevent attackers from guessing values. For implementation notes, see Section 15.5.2.

The following is a non-normative example request using the Implicit Flow that would be sent by the User Agent to the Authorization Server in response to a corresponding HTTP 302 redirect response by the Client (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
```

3.2.2.2. Authentication Request Validation

TOC

When using the Implicit Flow, the Authentication Request is validated in the same manner as for the Authorization Code Flow, as defined in Section 3.1.2.2.

3.2.2.3. 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 3.1.2.3.

3.2.2.4. 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 3.1.2.4.

3.2.2.5. Successful Authentication Response

TOC

When using the Implicit Flow, Authentication Responses are made in the same manner as for the Authorization Code Flow, as defined in Section 3.1.2.5, 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], unless a different Response Mode was specified.

These parameters are returned from the Authorization Endpoint:

access_token

OAuth 2.0 Access Token. 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 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.

Per Section 4.2.2 of OAuth 2.0 [RFC6749], no `code` result is returned when using the Implicit Flow.

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
```

3.2.2.6. Authentication 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 3.1.2.6, with the exception of the differences specified in this section.

Whenever Error Response parameters are returned, such as when End-User denies the authorization or the End-User authentication fails, the Authorization Server MUST return the error Authorization Response in the fragment component of the Redirection URI, as defined in Section 4.2.2.1 of OAuth 2.0 [RFC6749] and OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses], unless a different Response Mode was specified.

3.2.2.7. Redirect URI Fragment Handling

TOC

Since response parameters are returned in the Redirection URI fragment value, the Client needs to have the User Agent parse the fragment encoded values and pass them to on to the Client's processing logic for consumption. See Section 15.5.3 for implementation notes on URI fragment handling.

3.2.2.8. Authentication 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 3.2.2.11.
4. Follow the Access Token validation rules in Section 3.2.2.9, unless the `response_type` value used is `id_token`.

3.2.2.9. 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` Header Parameter of the ID Token's JOSE Header. For instance, if the `alg` is `RS256`, the hash algorithm used is SHA-256.
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.

3.2.2.10. ID Token

TOC

The contents of the ID Token are as described in Section 2. When using the Implicit Flow, these additional requirements for the following ID Token Claims apply:

nonce

Use of the `nonce` Claim is REQUIRED for this flow.

at_hash

Access Token hash value. Its value is the base64url encoding of the left-most half of the hash of the octets of the ASCII representation of the `access_token` value, where the hash algorithm used is the hash algorithm used in the `alg` Header Parameter of the ID Token's JOSE Header. For instance, if the `alg` is `RS256`, hash the `access_token` value with SHA-256, then take the left-most 128 bits and base64url-encode them. The `at_hash` value is a case-sensitive string.

If the ID Token is issued from the Authorization Endpoint with an `access_token` value, which is the case for the `response_type` value `id_token token`, this is REQUIRED; it MAY NOT be used when no Access Token is issued, which is the case for the `response_type` value `id_token`.

3.2.2.11. 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 3.1.3.7, 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` Header Parameter of the JOSE 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 Authentication Request. The Client SHOULD check the `nonce` value for replay attacks. The precise method for detecting replay attacks is Client specific.

3.3. 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].

3.3.1. Hybrid Flow Steps

TOC

The Hybrid Flow follows the following steps:

1. Client prepares an Authentication Request containing the desired request parameters.
2. Client sends the 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 and, depending on the Response Type, one or more additional parameters.

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 ID Token and retrieves the End-User's Subject Identifier.

3.3.2. 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 3.1.2, with the exception of the differences specified in this section.

3.3.2.1. Authentication Request

TOC

Authentication Requests are made as defined in Section 3.1.2.1, except that these Authentication Request parameters are used as follows:

response_type

> REQUIRED. OAuth 2.0 Response Type value that determines the authorization processing flow to be used, including what parameters are returned from the endpoints used. When using the Hybrid Flow, this value is `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].

nonce

> REQUIRED if the Response Type of the request is `code id_token` or `code id_token token` and OPTIONAL when the Response Type of the request is `code token`. It is a 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 Authentication Request to the

ID Token. Sufficient entropy MUST be present in the nonce values used to prevent attackers from guessing values. For implementation notes, see Section 15.5.2.

The following is a non-normative example request using the Hybrid Flow that would be sent by the User Agent to the Authorization Server in response to a corresponding HTTP 302 redirect response by the Client (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
```

3.3.2.2. Authentication Request Validation

TOC

When using the Hybrid Flow, the Authentication Request is validated in the same manner as for the Authorization Code Flow, as defined in Section 3.1.2.2.

3.3.2.3. 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 3.1.2.3.

3.3.2.4. 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 3.1.2.4.

3.3.2.5. Successful Authentication Response

TOC

When using the Hybrid Flow, Authentication Responses are made in the same manner as for the Implicit Flow, as defined in Section 3.2.2.5, with the exception of the differences specified in this section.

These Authorization Endpoint results are used in the following manner:

access_token

> OAuth 2.0 Access Token. 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`.

code

> Authorization Code. This is always returned when using the Hybrid Flow.

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
```

3.3.2.6. Authentication Error Response

TOC

When using the Hybrid Flow, Authorization Error Responses are made in the same manner as for the Authorization Code Flow, as defined in Section 3.1.2.6, with the exception of the differences specified in this section.

Whenever Error Response parameters are returned, such as when End-User denies the authorization or the End-User authentication fails, the Authorization Server MUST return the error Authorization Response in the fragment component of the Redirection URI, as defined in Section 4.2.2.1 of OAuth 2.0 [RFC6749] and OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses], unless a different Response Mode was specified.

3.3.2.7. Redirect URI Fragment Handling

TOC

When using the Hybrid Flow, the same requirements for Redirection URI fragment parameter handling apply as do for the Implicit Flow, as defined in Section 3.2.2.7. Also see Section 15.5.3 for implementation notes on URI fragment handling.

3.3.2.8. Authentication 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 3.3.2.12 when the `response_type` value used is `code id_token` or `code id_token token`.
4. Follow the Access Token validation rules in Section 3.3.2.9 when the `response_type` value used is `code token` or `code id_token token`.
5. Follow the Authorization Code validation rules in Section 3.3.2.10 when the `response_type` value used is `code id_token` or `code id_token token`.

3.3.2.9. 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 3.2.2.9.

3.3.2.10. Authorization 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` Header Parameter of the ID Token's JOSE Header. For instance, if the `alg` is `RS256`, the hash algorithm used is SHA-256.
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.

3.3.2.11. ID Token

TOC

The contents of the ID Token are as described in Section 2. When using the Hybrid Flow, these additional requirements for the following ID Token Claims apply to an ID Token returned from the Authorization Endpoint:

nonce

> If a `nonce` parameter is present in the Authentication Request, Authorization Servers MUST include a `nonce` Claim in the ID Token.

at_hash

Access Token hash value. Its value is the base64url encoding of the left-most half of the hash of the octets of the ASCII representation of the `access_token` value, where the hash algorithm used is the hash algorithm used in the `alg` Header Parameter of the ID Token's JOSE Header. For instance, if the `alg` is `RS256`, hash the `access_token` value with SHA-256, then take the left-most 128 bits and base64url-encode them. The `at_hash` value is a case-sensitive string.

If the ID Token is issued from the Authorization Endpoint with an `access_token` value, which is the case for the `response_type` value `code id_token token`, this is REQUIRED; otherwise, its inclusion is OPTIONAL.

c_hash

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` Header Parameter of the ID Token's JOSE 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 for the `response_type` values `code id_token` and `code id_token token`, this is REQUIRED; otherwise, its inclusion is OPTIONAL.

3.3.2.12. ID Token Validation

TOC

When using the Hybrid Flow, the contents of an ID Token returned from the Authorization Endpoint MUST be validated in the same manner as for the Implicit Flow, as defined in Section 3.2.2.11.

3.3.3. 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 3.1.3, with the exception of the differences specified in this section.

3.3.3.1. 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 3.1.3.1.

3.3.3.2. 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 3.1.3.2.

3.3.3.3. Successful Token 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 3.1.3.3.

3.3.3.4. 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 3.1.3.4.

3.3.3.5. 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 3.1.3.5.

3.3.3.6. ID Token

TOC

When using the Hybrid Flow, the contents of an ID Token returned from the Token Endpoint are the same as for an ID Token returned from the Authorization Endpoint, as defined in Section 3.3.2.11, 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 for the `response_type` values `code id_token` and `code id_token token`, the `iss` and `sub` Claim Values MUST be identical in both ID Tokens. All Claims about the Authentication event present in either SHOULD be present in both. If either ID Token contains Claims about the End-User, any that are present in both SHOULD have the same values in both. Note that the OP MAY choose to return fewer Claims about the End-User from the Authorization Endpoint, for instance, for privacy reasons. The `at_hash` and `c_hash` Claims MAY be omitted from the ID Token returned from the Token Endpoint even when these Claims are present in the ID Token returned from the Authorization Endpoint, because the ID Token and Access Token values returned from the Token Endpoint are already cryptographically bound together by the TLS encryption performed by the Token Endpoint.

3.3.3.7. ID Token Validation

TOC

When using the Hybrid Flow, the contents of an ID Token returned from the Token Endpoint MUST be validated in the same manner as for the Authorization Code Flow, as defined in Section 3.1.3.7.

3.3.3.8. Access Token

TOC

If an Access Token is returned from both the Authorization Endpoint and from the Token Endpoint, which is the case for the `response_type` values `code token` and `code id_token token`, their values MAY be the same or they MAY be different. Note that different Access Tokens might be returned be due to the different security characteristics of the two endpoints and the lifetimes and the access to resources granted by them might also be different.

3.3.3.9. Access Token Validation

TOC

When using the Hybrid 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 3.1.3.8.

4. Initiating Login from a Third Party

TOC

In some cases, the login flow is initiated by an OpenID Provider or another party, rather than the Relying Party. In this case, the initiator redirects to the RP at its login initiation endpoint, which requests that the RP send an Authentication Request to a specified OP. Note that this login initiation endpoint can be a different page at the RP than the RP's default landing page. RPs supporting OpenID Connect Dynamic Client Registration 1.0 [OpenID.Registration] register this endpoint value using the `initiate_login_uri` Registration parameter.

The party initiating the login request does so by redirecting to the login initiation endpoint at the RP, passing the following parameters:

iss

> REQUIRED. Issuer Identifier for the OP that the RP is to send the Authentication Request to. Its value MUST be a URL using the `https` scheme.

login_hint

OPTIONAL. Hint to the Authorization Server about the End-User to be authenticated. The meaning of this string-valued parameter is left to the OP's discretion. In common use cases, the value will contain an e-mail address, phone number, or username collected by the RP before requesting authentication at the OP. For example, this hint can be used by an RP after it asks the End-User for their e-mail address (or other identifier), passing that identifier as a hint to the OpenID Provider. It is RECOMMENDED that the hint value match the value provided for discovery. Other uses MAY include using the `sub` claim from the ID Token as the hint value or potentially other kinds of information about the requested authentication.

target_link_uri

OPTIONAL. URL that the RP is requested to redirect to after authentication. RPs MUST verify the value of the `target_link_uri` to prevent being used as an open redirector to external sites.

The parameters can either be passed as query parameters using the HTTP `GET` method or be passed as HTML form values that are auto-submitted in the User Agent, and thus are transmitted via the HTTP `POST` method.

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.

5. Claims

This section specifies how the Client can obtain Claims about the End-User and the Authentication event. It also 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.

5.1. Standard Claims

TOC

This specification defines a set of standard Claims. They can be requested to be returned either in the UserInfo Response, per Section 5.3.2, or in the ID Token, per Section 2.

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 by which 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. The RP MUST NOT rely upon this value being unique, as discussed in Section 5.7.
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. The RP MUST NOT rely upon this value being unique, as discussed in Section 5.7.
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 8601-1 [ISO8601-1] `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 IANA Time Zone Database [IANA.time-zones] 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 Alpha-2 [ISO639] 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`; Relying Parties 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 postal address. The value of the address member is a JSON [RFC8259] structure containing some or all of the members defined in Section 5.1.1.
updated_at	number	Time the End-User's information was last updated. Its value is a JSON number representing the number of seconds from 1970-01-01T00:00:00Z as measured in UTC until the date/time.

Table 1: Registered Member Definitions

5.1.1. 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 represent the same address, with the formatted address indicating how the component fields are combined.

All the address values defined below are represented as JSON strings.

formatted

Full mailing address, formatted for display or use on a mailing label. This field MAY contain multiple lines, separated by newlines. Newlines can be represented either as a carriage return/line feed pair ("\r\n") or as a single line feed character ("\n").

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 newlines. Newlines can be represented either as a carriage return/line feed pair ("\r\n") or as a single line feed character ("\n").

locality

City or locality component.

region

State, province, prefecture, or region component.

postal_code

Zip code or postal code component.

country

Country name component.

5.1.2. Additional Claims

TOC

While this specification defines only a 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 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 the JWT specification. Or, if specific additional Claims will have broad and general applicability, they can be registered with Registered Claim Names, per the JWT specification.

5.2. 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 name 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.)

OpenID Connect defines the following Authorization Request parameter to enable specify the preferred languages and scripts to be used for the returned Claims:

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.

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.

5.3. 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 request to the UserInfo Endpoint using an Access Token obtained through OpenID Connect Authentication. These Claims are normally 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 16.17 for more information on using TLS.

The UserInfo Endpoint MUST support the use of the HTTP `GET` and HTTP `POST` methods defined in RFC 7231 [RFC7231].

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 JavaScript Clients and other Browser-Based Clients to access it.

5.3.1. UserInfo Request

TOC

The Client sends the UserInfo Request using either HTTP `GET` or HTTP `POST`. The Access Token obtained from an OpenID Connect Authentication Request MUST be sent as a Bearer Token, per Section 2 of OAuth 2.0 Bearer Token Usage [RFC6750].

It is RECOMMENDED that the request use the HTTP `GET` method and the Access Token be sent using the `Authorization` header field.

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

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

TOC

5.3.2. Successful UserInfo Response

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 5.1 can be returned, as can additional Claims not specified there.

For privacy reasons, OpenID Providers MAY elect to not return values for some requested Claims. It is not an error condition to not return a requested Claim.

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: Due to the possibility of token substitution attacks (see Section 16.11), the UserInfo 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 Response MUST be verified to exactly match the `sub` Claim in the ID Token; if they do not match, the UserInfo Response values MUST NOT be used.

Upon receipt of the UserInfo Request, the UserInfo Endpoint MUST return the JSON Serialization of the UserInfo Response as in Section 13.3 in the HTTP response body unless a different format was specified during Registration [OpenID.Registration]. The UserInfo Endpoint MUST return a content-type header to indicate which format is being returned. The content-type of the HTTP response MUST be `application/json` if the response body is a text JSON object; the response body SHOULD be encoded using UTF-8.

If the UserInfo Response is signed and/or encrypted, then the Claims are returned in a JWT and the content-type MUST be `application/jwt`. The response MAY be encrypted without also being signed. If both signing and encryption are requested, the response MUST be signed then encrypted, with the result being a Nested JWT, as defined in [JWT].

If signed, the UserInfo Response MUST contain the Claims `iss` (issuer) and `aud` (audience) as members. The `iss` value MUST be the OP's Issuer Identifier URL. The `aud` value MUST be or include the RP's Client ID value.

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",
 "preferred_username": "j.doe",
 "email": "janedoe@example.com",
 "picture": "http://example.com/janedoe/me.jpg"
}
```

5.3.3. 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]. (HTTP errors unrelated to RFC 6750 are returned to the User Agent using the appropriate HTTP status code.)

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

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

5.3.4. UserInfo Response Validation

TOC

The Client MUST validate the UserInfo Response as follows:

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

TOC

5.4. Requesting Claims using Scope Values

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.

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

OpenID Connect defines the following scope values that are used to request Claims:

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 5.3.2, when a `response_type` value is used that results in an Access Token being issued. However, when no Access Token is issued (which is the case for the `response_type` value `id_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 RPs. To minimize the amount of information that the End-User is being asked to disclose, an RP can elect to only request a subset of the information available from the UserInfo Endpoint.

The following is a non-normative example of an unencoded `scope` request:

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

5.5. 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` Authentication 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 6.1, 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 the Client for 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, as per the ID Token definition in Section 2 and per the additional per-flow ID Token requirements in Sections 3.1.3.6, 3.2.2.10, 3.3.2.11, and 3.3.3.6.

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 5.1, the (example) http://example.info/claims/groups Claim, is being requested. Using the claims parameter is the only way to request specific combinations of Claims that cannot be specified using scope values.

TOC

5.5.1. Individual Claims Requests

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 RP 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. An equality comparison is used to determine whether the requested Claim value matches.

When the Claim value does not match the requested value, the Claim is not included in the response. If the Claim was sub, a mismatch MUST cause the authentication to fail, as described in Section 3.1.2.2.

values

OPTIONAL. Requests that the Claim be returned with one of a set of values, with the values appearing in order of preference. This is processed equivalently to a value request, except that a choice of acceptable Claim values is provided.

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. An equality comparison is used to determine whether the requested Claim values match.

When the Claim value does not match any of the requested values, the Claim is not included in the response.

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 5.4, 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
  }
}
```

5.5.1.1. Requesting the "acr" Claim

TOC

If the `acr` Claim is requested as an Essential Claim for the ID Token with a `value` or `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 RP 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.

5.5.2. Languages and Scripts for Individual Claims

TOC

As described in Section 5.2, human-readable Claim 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 5.2. 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 does not have, any versions of those Claims returned that do not use the requested language and script SHOULD use a language tag in the Claim Name.

5.6. Claim Types

TOC

Three representations of Claim Values are defined by this specification:

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.

Normal Claims MUST be supported. Support for Aggregated Claims and Distributed Claims is OPTIONAL.

5.6.1. 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:

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

TOC

5.6.2. Aggregated and Distributed Claims

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. The OP MAY omit some Claims available from referenced Claims Providers from the set of Claim Names.

_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, RPs MAY need to retrieve the Access Token out of band or use an Access Token that was pre-negotiated between the Claims Provider and RP, or the Claims Provider MAY reauthenticate the End-User and/or reauthorize the RP.

Since it is not an error condition to not return a requested Claim, RPs MUST be prepared to handle the condition that some Claims listed in _claim_sources are not returned from the Claims Provider. They SHOULD treat this the same as when any other requested Claim is not returned.

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.

An iss (issuer) Claim SHOULD be included in any JWT issued by a Claims Provider so that the Claims Provider's keys can be retrieved for signature validation of the JWT. The value of the Claim is the Claims Provider's Issuer Identifier URL.

In general, it is up to the OP when it is appropriate to use Aggregated Claims and Distributed Claims. In some cases, information about when to use what Claim Types might be negotiated out of band between RPs and OPs.

TOC

5.6.2.1. Example of Aggregated Claims

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. (The example also includes the Claims Provider's Issuer Identifier URL.)

```
{
 "iss": "https://a.example.com",
 "address": {
   "street_address": "1234 Hollywood Blvd.",
   "locality": "Los Angeles",
   "region": "CA",
   "postal_code": "90210",
   "country": "United States of America"},
 "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:

```
{
 "sub": "248289761001",
 "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"}
 }
}
```

5.6.2.2. 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). (The example also includes the Claims Provider's Issuer Identifier URL.)

```
{
 "iss": "https://bank.example.com",
 "shipping_address": {
   "street_address": "1234 Hollywood Blvd.",
   "locality": "Los Angeles",
   "region": "CA",
   "postal_code": "90210",
   "country": "United States of America"},
 "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). (The example also includes the Claims Provider's Issuer Identifier URL.)

```
{
 "iss": "https://creditagency.example.com",
 "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:

```
  {
   "sub": "248289761001",
   "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"}
   }
  }
```

Note that not returning `phone_number`, which is held by Claims Provider B, demonstrates that not all Claims held by a utilized Claims Provider need be included.

5.7. Claim Stability and Uniqueness

TOC

The `sub` (subject) and `iss` (issuer) Claims from the ID Token, used together, are the only Claims that an RP 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. 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`, `preferred_username`, and `name` MUST NOT be used as unique identifiers for the End-User, whether obtained from the ID Token or the UserInfo Endpoint.

---

6. 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 optionally signed and/or encrypted. The parameter value is a Request Object value, as specified in Section 6.1. It represents the request as a JWT whose Claims are the request parameters.

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 referencing a resource containing a Request Object value, which is a JWT containing the request parameters. This URL MUST use the `https` scheme unless the target Request Object is signed in a way that is verifiable by the OP.

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.

Note that the Request Objects defined here are compatible with those specified by The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR) [RFC9101].

TOC

6.1. Passing a Request Object by Value

The `request` Authorization Request parameter enables OpenID Connect requests to be passed in a single, self-contained parameter and to be optionally signed and/or encrypted. It represents the request as a JWT whose Claims are the request parameters specified in Section 3.1.2. 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, parameters MAY also 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.

So that the request is a valid OAuth 2.0 Authorization Request, values for the `response_type` and `client_id` parameters MUST be included using the OAuth 2.0 request syntax, since they are REQUIRED by OAuth 2.0. The values for these parameters MUST match those in the Request Object, if present.

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 (unsecured). When it is unsecured, this is indicated by use of the `none` algorithm [JWA] in the JOSE Header. If signed, the Request Object SHOULD contain the Claims `iss` (issuer) and `aud` (audience) as members. The `iss` value SHOULD be the Client ID of the RP, unless it was signed by a different party than the RP. The `aud` value SHOULD be or include the OP's Issuer Identifier URL.

The Request Object MAY also be encrypted using JWE [JWE] and MAY be encrypted without also being signed. If both signing and encryption are performed, it MUST be signed then encrypted, with the result being a Nested JWT, as defined in [JWT].

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:

```
  {
   "iss": "s6BhdRkqt3",
   "aud": "https://server.example.com",
   "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):

```

eyJhbGciOiJSUzI1NiIsImtpZCI6ImsyYmRjIn0.ew0KICJpc3MiOiAi
czZCaGRSa3

F0MyIsDQogImF1ZCI6ICJodHRwczovL3NlcnZlci5leGFtcGxlLmNvbS
IsDQogInJl
```

```
c3BvbnNlX3R5cGUiOiAiY29kZSBpZF90b2tlbiIsDQogImNsaWVudF9p
ZCI6ICJzNk

JoZFJrcXQzIiwNCiAicmVkaXJlY3RfdXJpIjogImh0dHBzOi8vY2xpZW
50LmV4YW1w

bGUub3JnL2NiIiwNCiAic2NvcGUiOiAib3BlbmlkIiwNCiAic3RhdGUi
OiAiYWYwaW

Zqc2xka2oiLA0KICJub25jZSI6ICJuLTBTNl9XekEyTWoiLA0KICJtYX
hfYWdlIjog

ODY0MDAsDQogImNsYWltcyI6IA0KICB7DQogICAidXNlcmluZm8iOiAN
CiAgICB7DQ

ogICAgICJnaXZlbl9uYW1lIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCi
AgICAgIm5p

Y2tuYW1lIjogbnVsbCwNCiAgICAgImVtYWlsIjogeyJlc3NlbnRpYWwi
OiB0cnVlfS

wNCiAgICAgImVtYWlsX3ZlcmlmaWVkIjogeyJlc3NlbnRpYWwiOiB0cn
VlfSwNCiAg

ICAgInBpY3R1cmUiOiBudWxsDQogICAgfSwNCiAgICJpZF90b2tlbiI6
IA0KICAgIH

sNCiAgICAgImdlbmRlciI6IG51bGwsDQogICAgICJiaXJ0aGRhdGUiOi
B7ImVzc2Vu

dGlhbCI6IHRydWV9LA0KICAgICAiYWNyIjogeyJ2YWx1ZXMiOiBbInVy
bjptYWNlOm

luY29tbW9uOmlhcDpzaWx2ZXIiXX0NCiAgICB9DQogIH0NCn0.nwwnNs
k1-Zkbmnvs
  F6zTHm8CHERFMGQPhos-EJcaH4Hh-
sMgk8ePrGhw_trPYs8KQxsn6R9Emo_wHwajyF

KzuMXZFSZ3p6Mb8dkxtVyjoy2GIzvuJT_u7PkY2t8QU9hjBcHs68Pkgj
DVTrG1uRTx

0GxFbuPbj96tVuj11pTnmFCUR6IEOXKYr7iGOCRB3btfJhM0_AKQUfqK
nRlrRscc8K
  ol-
cSLWoYE9l5QqholImzjT_cMnNIznW9E7CDyWXTsO70xnB4SkG6pXfLSj
LLlxmPG
  iyon_-Te111V8uE83IlzCYIb_NMXvtTIVc1jpspnTSD7xMbpL-
2QgwUsAlMGzw
```

The following RSA public key, represented in JWK format, 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",
   "kid":"k2bdc",
   "n":"y9Lqv4fCp6Ei-u2-
ZCKq83YvbFEk6JMs_pSj76eMkddWRuWX2aBKGHAtKlE5P

7_vn__PCKZWePt3vGkB6ePgzAFu08NmKemwE5bQI0e6kIChtt_6KzT5O
aaXDF

I6qCLJmk51Cc4VYFaxgqevMncYrzaW_50mZ1yGSFIQzLYP8bijAHGVjd
EFgZa

ZEN9lsn_GdWLaJpHrB3ROlS50E45wxrlg9xMncVb8qDPuXZarvghLL0H
zOuYR
        adBJVoWZowDNTpKpk2RklZ7QaBO7XDv3uR7s_sf2g-
bAjSYxYUGsqkNA9b3xV
        W53am_UZZ3tZbFTIh557JICWKHlWj5uzeJXaw",
   "e":"AQAB"
  }
```

6.1.1. 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=eyJhbGciOiJSUzI1NiIsImtpZCI6ImsyYmRjIn0.ew0KICJ
pc3MiOiA
```

```
iczZCaGRSa3F0MyIsDQogImF1ZCI6ICJodHRwczovL3NlcnZlci5leGF
tcGxlLmN

vbSIsDQogInJlc3BvbnNlX3R5cGUiOiAiY29kZSBpZF90b2tlbiIsDQo
gImNsaWV

udF9pZCI6ICJzNkJoZFJrcXQzIiwNCiAicmVkaXJlY3RfdXJpIjogImh
0dHBzOi8

vY2xpZW50LmV4YW1wbGUub3JnL2NiIiwNCiAic2NvcGUiOiAib3Blbml
kIiwNCiA

ic3RhdGUiOiAiYWYwaWZqc2xka2oiLA0KICJub25jZSI6ICJuLTBTNl9
XekEyTWo

iLA0KICJtYXhfYWdlIjogODY0MDAsDQogImNsYWltcyI6IA0KICB7DQo
gICAidXN

lcmluZm8iOiANCiAgICB7DQogICAgICJnaXZlbl9uYW1lIjogeyJlc3N
lbnRpYWw

iOiB0cnVlfSwNCiAgICAgIm5pY2tuYW1lIjogbnVsbCwNCiAgICAgImV
tYWlsIjo

geyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAgICAgImVtYWlsX3ZlcmlmaWV
kIjogeyJ

lc3NlbnRpYWwiOiB0cnVlfSwNCiAgICAgInBpY3R1cmUiOiBudWxsDQo
gICAgfSw

NCiAgICJpZF90b2tlbiI6IA0KICAgIHsNCiAgICAgImdlbmRlciI6IG5
1bGwsDQo

gICAgICJiaXJ0aGRhdGUiOiB7ImVzc2VudGlhbCI6IHRydWV9LA0KICA
gICAiYWN

yIjogeyJ2YWx1ZXMiOiBbInVybjptYWNlOmluY29tbW9uOmlhcDpzaWx
2ZXIiXX0
    NCiAgICB9DQogIH0NCn0.nwwnNsk1-
ZkbmnvsF6zTHm8CHERFMGQPhos-EJcaH4H
    h-
sMgk8ePrGhw_trPYs8KQxsn6R9Emo_wHwajyFKzuMXZFSZ3p6Mb8dkxt
Vyjoy2

GIzvuJT_u7PkY2t8QU9hjBcHs68PkgjDVTrG1uRTx0GxFbuPbj96tVuj
11pTnmFC
    UR6IEOXKYr7iGOCRB3btfJhM0_AKQUfqKnRlrRscc8Kol-
cSLWoYE9l5QqholImz
```

```
    jT_cMnNIznW9E7CDyWXTsO70xnB4SkG6pXfLSjLLlxmPGiyon_-
Te111V8uE83Il
    zCYIb_NMXvtTIVc1jpspnTSD7xMbpL-2QgwUsAlMGzw
```

6.2. 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, parameters MAY also 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.

So that the request is a valid OAuth 2.0 Authorization Request, values for the `response_type` and `client_id` parameters MUST be included using the OAuth 2.0 request syntax, since they are REQUIRED by OAuth 2.0. The values for these parameters MUST match those in the Request Object, if present.

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 SHOULD 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):

```


eyJhbGciOiJSUzI1NiIsImtpZCI6ImsyYmRjIn0.ew0KICJpc3MiOiAi
czZCaGRSa3

F0MyIsDQogImF1ZCI6ICJodHRwczovL3NlcnZlci5leGFtcGxlLmNvbS
IsDQogInJl

c3BvbnNlX3R5cGUiOiAiY29kZSBpZF90b2tlbiIsDQogImNsaWVudF9p
ZCI6ICJzNk

JoZFJrcXQzIiwNCiAicmVkaXJlY3RfdXJpIjogImh0dHBzOi8vY2xpZW
50LmV4YW1w

bGUub3JnL2NiIiwNCiAic2NvcGUiOiAib3BlbmlkIiwNCiAic3RhdGUi
OiAiYWYwaW

Zqc2xka2oiLA0KICJub25jZSI6ICJuLTBTNl9XekEyTWoiLA0KICJtYX
hfYWdlIjog

ODY0MDAsDQogImNsYWltcyI6IA0KICB7DQogICAidXNlcmluZm8iOiAN
CiAgICB7DQ

ogICAgICJnaXZlbl9uYW1lIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCi
AgICAgIm5p

Y2tuYW1lIjogbnVsbCwNCiAgICAgImVtYWlsIjogeyJlc3NlbnRpYWwi
OiB0cnVlfS

wNCiAgICAgImVtYWlsX3ZlcmlmaWVkIjogeyJlc3NlbnRpYWwiOiB0cn
VlfSwNCiAg
```

```
ICAgInBpY3R1cmUiOiBudWxsDQogICAgfSwNCiAgICJpZF90b2tlbiI6
IA0KICAgIH

sNCiAgICAgImdlbmRlciI6IG51bGwsDQogICAgICJiaXJ0aGRhdGUiOi
B7ImVzc2Vu

dGlhbCI6IHRydWV9LA0KICAgICAiYWNyIjogeyJ2YWx1ZXMiOiBbInVy
bjptYWNlOm

luY29tbW9uOmlhcDpzaWx2ZXIiXX0NCiAgICB9DQogIH0NCn0.nwwnNs
k1-Zkbmnvs
  F6zTHm8CHERFMGQPhos-EJcaH4Hh-
sMgk8ePrGhw_trPYs8KQxsn6R9Emo_wHwajyF

KzuMXZFSZ3p6Mb8dkxtVyjoy2GIzvuJT_u7PkY2t8QU9hjBcHs68Pkgj
DVTrG1uRTx

0GxFbuPbj96tVuj11pTnmFCUR6IEOXKYr7iGOCRB3btfJhM0_AKQUfqK
nRlrRscc8K
  ol-
cSLWoYE9l5QqholImzjT_cMnNIznW9E7CDyWXTsO70xnB4SkG6pXfLSj
LLlxmPG
  iyon_-Te111V8uE83IlzCYIb_NMXvtTIVc1jpspnTSD7xMbpL-
2QgwUsAlMGzw
```

6.2.1. URI Referencing the Request Object

TOC

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

If the Request Object includes requested values for Claims, 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
```

6.2.2. 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
```

6.2.3. Authorization Server Fetches Request Object

TOC

Upon receipt of the Request, the Authorization Server MUST send an HTTP `GET` request to the `request_uri` to retrieve the referenced Request Object, 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
```

6.2.4. "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.

6.3. 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 3.1.2.2, 3.2.2.2, or 3.3.2.2. These steps are to validate the JWT containing the Request Object and to validate the Request Object itself.

6.3.1. 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 (unsecured) Request Object. In the former case, signature validation MUST be performed as defined in Section 6.3.2.

The Authorization Server MUST return an error if decryption fails.

6.3.2. Signed Request Object

TOC

To perform Signature Validation, the `alg` Header Parameter in the JOSE 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 appropriate key for that `client_id` and algorithm.

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

6.3.3. 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 the assembled set of Authorization Request parameters, the Authorization Server then validates the request the normal manner for the flow being used, as specified in Sections 3.1.2.2, 3.2.2.2, or 3.3.2.2.

7. Self-Issued OpenID Provider

TOC

OpenID Connect supports Self-Issued OpenID Providers - personal, self-hosted 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.

7.1. 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 RPs 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.

7.2. Self-Issued OpenID Provider Registration

TOC

When using a Self-Issued OP, registration is not required. The Client can proceed without registration as if it had registered with the OP and obtained the 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 RPs 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.

7.2.1. 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 a UTF-8 encoded JSON object (which ends up being form-urlencoded when passed as an OAuth parameter). When used in a Request Object value, per Section 6.1, the JSON object 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 typically use the jwks_uri, id_token_encrypted_response_alg and id_token_encrypted_response_enc parameters.

7.3. Self-Issued OpenID Provider Request

TOC

The self-issued OP's Authorization Endpoint is the URI openid:.

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

scope

REQUIRED. scope parameter value, as specified in Section 3.1.2.

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 3.1.2. Encrypting content to Self-Issued OPs is not supported.

claims

OPTIONAL. `claims` parameter value, as specified in Section 5.5.

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 7.2.1.

request

OPTIONAL. Request Object value, as specified in Section 6.1. Encrypting content to Self-Issued OPs is not supported.

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 HTTP 302 redirect response by the Client, which triggers the User Agent to make an Authentication Request to the Self-Issued OpenID Provider (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
```

7.4. Self-Issued OpenID Provider Response

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

sub_jwk

REQUIRED. Public key used to check the signature of an ID Token issued by a Self-Issued OpenID Provider, as specified in Section 7. 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, unless a different Response Mode was specified.

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 used to check the signature of the ID Token.
3. The `sub` (subject) Claim value is the base64url-encoded representation of the thumbprint of the key in the `sub_jwk` Claim. This thumbprint value is computed as the SHA-256 hash of the octets of the UTF-8 representation of a JWK constructed containing only the REQUIRED members to represent the key, with the member names sorted into lexicographic order, and with no whitespace or line breaks. For instance, when the `kty` value is `RSA`, the member names `e`, `kty`, and `n` are the ones present in the constructed JWK used in the thumbprint computation and appear in that order; when the `kty` value is `EC`, the member names `crv`, `kty`, `x`, and `y` are present in that order. Note that this thumbprint calculation is the same as that defined in the JWK Thumbprint [JWK.Thumbprint] specification.
4. No Access Token is returned for accessing a UserInfo Endpoint, so all Claims returned MUST be in the ID Token.

TOC

7.5. Self-Issued ID Token Validation

To validate the ID Token received, the Client MUST do the following:

1. The Client MUST validate that the value of the `iss` (issuer) Claim is `https://self-issued.me`. If `iss` contains a different value, the ID Token is not Self-Issued, and instead it MUST be validated according to Section 3.1.3.7.
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` Header Parameter of the JOSE Header, 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` Claim value is the base64url-encoded representation of the thumbprint of the key in the `sub_jwk` Claim, as specified in Section 7.4.
6. The current time MUST be before the time represented by 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. 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 Authentication 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": "NzbLsXh8uDCcd-6MNwXF4W_7noWXFZAfHkxZsRGC9Xs",
   "aud": "https://client.example.org/cb",
   "nonce": "n-0S6_WzA2Mj",
   "exp": 1311281970,
   "iat": 1311280970,
   "sub_jwk": {
     "kty":"RSA",
     "n":
"0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx

4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n
3oknjhMs
     tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-
65YGjQR0_FDW2

QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n
91CbOpbI
     SD08qNLyrdkt-
bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
     w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",
     "e":"AQAB"
    }
  }
```

8. Subject Identifier Types

TOC

A Subject Identifier is a locally unique and never reassigned identifier within the Issuer for the End-User, which is intended to be consumed by the Client. Two Subject Identifier types are defined by this specification:

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, so as not to enable Clients to correlate the End-User's activities without permission.

The OpenID Provider's Discovery document MUST list its supported Subject 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.

8.1. Pairwise Identifier Algorithm

TOC

When pairwise Subject Identifiers are used, the OpenID Provider MUST calculate a unique `sub` (subject) value for each Sector Identifier. The Subject Identifier value MUST NOT be reversible by any party other than the OpenID Provider.

Providers that use pairwise `sub` values and support Dynamic Client Registration [OpenID.Registration] SHOULD use the `sector_identifier_uri` parameter. 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 re-register 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.

Any algorithm with the following properties can be used by OpenID Providers to calculate pairwise Subject Identifiers:

- The Subject Identifier value MUST NOT be reversible by any party other than the OpenID Provider.
- Distinct Sector Identifier values MUST result in distinct Subject Identifier values.
- The algorithm MUST be deterministic.

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.

9. Client Authentication

TOC

This section defines a set of Client Authentication methods that are used by Clients to authenticate to the Authorization Server when using the Token Endpoint. During Client Registration, the RP (Client) MAY register a Client Authentication method. If no method is registered, the default method is `client_secret_basic`.

These Client Authentication methods 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 2.3.1 of OAuth 2.0 [RFC6749] using the 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 2.3.1 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 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, which can be used to prevent reuse of the token. These tokens MUST only be used once, unless conditions for reuse were negotiated between the parties; any such negotiation is beyond the scope of this specification.

exp

REQUIRED. Expiration time on or after which the JWT 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].

private_key_jwt

Clients that have registered a public key sign a JWT using that key. The Client authenticates in accordance with 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, which can be used to prevent reuse of the token. These tokens MUST only be used once, unless conditions for reuse were negotiated between the parties; any such negotiation is beyond the scope of this specification.

exp

REQUIRED. Expiration time on or after which the JWT 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
```

none

The Client does not authenticate itself at the Token Endpoint, either because it uses only the Implicit Flow (and so does not use the Token Endpoint) or because it is a Public Client with no Client Secret or other authentication mechanism.

10. 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 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 16.14, with the result being a Nested JWT, as specified in [JWT]. Note that all JWE encryption methods perform integrity checking.

The OP advertises its supported signing and encryption algorithms in its Discovery document or may supply this information by other means. The RP declares its required signing and encryption algorithms in its Dynamic Registration request or may communicate this information by other means.

The OP advertises its public keys via its Discovery document or may supply this information by other means. The RP declares its public keys via its Dynamic Registration request or may communicate this information by other means.

10.1. Signing

TOC

The signing party MUST select a signature algorithm based on the algorithms supported by the recipient.

Asymmetric Signatures

> When using RSA or ECDSA Signatures, the `alg` Header Parameter value of the JOSE Header MUST be set to an appropriate algorithm as defined in JSON Web Algorithms [JWA]. The private key used to sign the content MUST be associated with a public key used for signature verification published by the sender in its JWK Set document. If there are multiple keys in the referenced JWK Set document, a `kid` value MUST be provided in the JOSE Header. The key usage of the respective keys MUST support signing.

Symmetric Signatures

> When using MAC-based signatures, the alg Header Parameter value of the JOSE 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 16.19 for a discussion of entropy requirements for client_secret values. Symmetric signatures MUST NOT be used by public (non-confidential) Clients because of their inability to keep secrets.

See Section 16.20 for Security Considerations about the need for signed requests.

10.1.1. 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 its jwks_uri location and includes the kid of the signing key in the JOSE 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 the jwks_uri location. 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 location 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.

10.2. Encryption

TOC

The encrypting party MUST select an encryption algorithm based on the algorithms supported by the recipient.

Asymmetric Encryption: RSA

> The public key to which the content was encrypted MUST be a public key used for encryption published by the recipient in its JWK Set document. If there are multiple keys in the referenced JWK Set document, a kid value MUST be

provided in the JOSE Header. Use the supported RSA encryption algorithm to encrypt a random Content Encryption 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 JOSE Header. The other public key used for the key agreement computation MUST be a public key published by the recipient in its JWK Set document. If there are multiple keys in the referenced JWK Set document, a `kid` value MUST be provided in the JOSE Header. Use the ECDH-ES algorithm to agree upon a Content Encryption 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 the left-most bits of a truncated SHA-2 hash of the octets of the UTF-8 representation of the `client_secret`. For keys of 256 or fewer bits, SHA-256 is used; for keys of 257-384 bits, SHA-384 is used; for keys of 385-512 bits, SHA-512 is used. The hash value MUST be truncated retaining the left-most bits to the appropriate bit length for the AES key wrapping or direct encryption algorithm used, for instance, truncating the SHA-256 hash to 128 bits for `A128KW`. If a symmetric key with greater than 512 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 NOT be used by public (non-confidential) Clients because of their inability to keep secrets.

See Section 16.21 for Security Considerations about the need for encrypted requests.

10.2.1. Rotation of Asymmetric Encryption Keys

TOC

Rotating encryption keys necessarily uses a different process than the one for signing keys because the encrypting party starts the process and thus cannot rely on a change in `kid` as a signal that keys need to change. The encrypting party still uses the `kid` Header Parameter 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 the recipient's `jwks_uri` location.

To rotate keys, the decrypting party can publish new keys at its `jwks_uri` location 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 7234 [RFC7234], 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 referenced by `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 10.1.1.

11. 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 the request 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 the request 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 offline access when the registered application_type is web,
- SHOULD explicitly receive or have consent for offline access 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.

12. 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.

12.1. 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 9. The Client sends the parameters via HTTP POST to the Token Endpoint using Form Serialization, per Section 13.2.

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, MUST verify that it was issued to the Client, and must verify that the Client successfully authenticated it has a Client Authentication method.

12.2. Successful Refresh Response

TOC

Upon successful validation of the Refresh Token, the response body is the Token Response of Section 3.1.3.3 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,
- if the implementation is using extensions (which are beyond the scope of this specification) that result in the `azp` (authorized party) Claim being present, those extensions might specify that its `azp` Claim Value MUST be the same as in the ID Token issued when the original authentication occurred; likewise, they might specify that if no `azp` Claim was present in the original ID Token, one MUST NOT be present in the new ID Token,
- it SHOULD NOT have a `nonce` Claim, even when the ID Token issued at the time of the original authentication contained `nonce`; however, if it is present, its value MUST be the same as in the ID Token issued at the time of the original authentication, 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

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

12.3. 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].

13. Serializations

TOC

Messages are serialized using one of the following methods:

1. Query String Serialization
2. Form Serialization
3. JSON Serialization

This section describes the syntax of these serialization methods; other sections describe when they can and must be used. Note that not all methods can be used for all messages.

13.1. 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.SPSD-html401-20180327]. 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
```

13.2. 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.SPSD-html401-20180327]. 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
```

13.3. JSON Serialization

TOC

The parameters are serialized into a JSON object structure by adding each parameter at the highest structure level. Parameter names and string values are represented as JSON strings. Numerical values are represented as JSON numbers. Boolean values are represented as JSON booleans. Omitted parameters and parameters with no value SHOULD be omitted from the object and not represented by a JSON `null` value, unless otherwise specified. A parameter MAY have a JSON object or a JSON array 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"
}
```

14. 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 [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, a single ASCII space character (0x20) MUST be used as the delimiter.

15. 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.

15.1. 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`), unless the OP only supports returning ID Tokens from the Token Endpoint (as is the case for the Authorization Code Flow) and only allows Clients to register specifying `none` as the requested ID Token signing algorithm.

Prompt Parameter

> OPs MUST support the `prompt` parameter, as defined in Section 3.1.2, including the specified user interface behaviors such as `none` and `login`.

Display Parameter

> OPs MUST support the `display` parameter, as defined in Section 3.1.2. (Note that the minimum level of support

required for this parameter is simply that its use must 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 3.1.2. (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, when requested, as defined in Section 2.

Maximum Authentication Age

OPs MUST support enforcing a maximum authentication age via the `max_age` parameter, as defined in Section 3.1.2.

Authentication Context Class Reference

OPs MUST support requests for specific Authentication Context Class Reference values via the `acr_values` parameter, as defined in Section 3.1.2. (Note that the minimum level of support required for this parameter is simply to have its use not result in an error.)

15.2. 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 `code` and `id_token token` 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 5.3. (Self-Issued OPs do not issue Access Tokens.)

Public Keys Published as Bare Keys

> These OPs MUST publish their public keys as bare JWK keys (which MAY also be accompanied by X.509 representations of those keys).

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 6.2.

15.3. 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.

15.4. 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 statically registered 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.

15.5. Implementation Notes

TOC

15.5.1. Authorization Code Implementation Notes

TOC

When using the Authorization Code or Hybrid flows, an ID Token is returned from the Token Endpoint in response to a Token Request using an Authorization Code. Some implementations may choose to encode state about the ID Token to be returned in the Authorization Code value. Others may use the Authorization Code value as an index into a database storing this state.

15.5.2. Nonce Implementation Notes

TOC

The nonce parameter value needs to include per-session state and be unguessable to attackers. One method to achieve this for Web Server Clients is to store a cryptographically random value as an HttpOnly session cookie and use a cryptographic hash of the value as the nonce parameter. In that case, the nonce in the returned ID Token is compared to the hash of the session cookie to detect ID Token replay by third parties. A related method applicable to JavaScript Clients and other Browser-Based Clients is to store the cryptographically random value in HTML5 local storage and use a cryptographic hash of this value.

15.5.3. Redirect URI Fragment Handling Implementation Notes

TOC

When response parameters are returned in the Redirection URI fragment value, the Client needs to have the User Agent parse the fragment encoded values and pass them to on to the Client's processing logic for consumption. User Agents that have direct access to cryptographic APIs may be able to be self-contained, for instance, with all Client code being written in JavaScript.

However, if the Client does not run entirely in the User Agent, one way to achieve this is to post them to a 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 and use 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);
```

15.6. Compatibility Notes

TOC

NOTE: Potential compatibility issues that were previously described in the original version of this specification have since been addressed.

15.7. Related Specifications and Implementer's Guides

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

- OpenID Connect Discovery 1.0 [OpenID.Discovery] - Defines how Relying Parties dynamically discover information about OpenID Providers
- OpenID Connect Dynamic Client Registration 1.0 [OpenID.Registration] - Defines how Relying Parties dynamically register with OpenID Providers
- OAuth 2.0 Form Post Response Mode [OAuth.Post] - Defines how to return OAuth 2.0 Authorization Response parameters (including OpenID Connect Authentication Response parameters) using HTML form values that are auto-submitted by the User Agent using HTTP `POST`
- OpenID Connect RP-Initiated Logout 1.0 [OpenID.RPInitiated] - Defines how a Relying Party requests that an OpenID Provider log out the End-User
- OpenID Connect Session Management 1.0 [OpenID.Session] - Defines how to manage OpenID Connect sessions, including postMessage-based logout and RP-initiated logout functionality
- OpenID Connect Front-Channel Logout 1.0 [OpenID.FrontChannel] - Defines a front-channel logout mechanism that does not use an OP iframe on RP pages
- OpenID Connect Back-Channel Logout 1.0 [OpenID.BackChannel] - Defines a logout mechanism that uses direct back-channel communication between the OP and RPs being logged out

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

16. 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.

16.1. 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.

16.2. 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.

16.3. Token Manufacture/Modification

TOC

An Attacker might generate a bogus token or modify the token contents (such as Claims values or the signature) 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 16.17 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.

16.4. 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 an End-User's authorization and MUST NOT be exposed to unauthorized parties.

16.5. Server Response Disclosure

TOC

The server response might contain authentication data and Claims 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.

16.6. 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.

16.7. 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 its integrity is intact.

16.8. 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.

16.9. 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.

16.10. 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 either Client Authentication or an encrypted response is used.

16.11. 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 ID Token and Access Token were 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), `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 Authorization Code substitution. The `at_hash` in the ID Token enables Clients to prevent Access Token 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 Authorization 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.

16.12. 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.