# oidcservice Documentation

*Release 0.1*

**Roland Hedberg**

**Oct 05, 2020**

# Contents

OpenID Connect and OAuth2 (O/O) are both request-response protocols. The client sends a request and the server responds either direct on the same connection or after a while on another connection.

When I use *client* below I refer to a piece of software that implements O/O and works on behalf of an application.

The client follows the same pattern disregarding which request/response it is dealing with. I does the following when sending a request:

1. Gathers the request arguments
2. If client authentication is involved it gathers the necessary data for that
3. If the chosen client authentication method involved adding information to the request it does so.
4. Adds information to the HTTP headers like Content-Type
5. Serializes the request into the expected format

after that follows the act of sending the request to the server and receiving the response from it. Once the response have been received, The client will follow this path:

1. Deserialize the received message into a internal format
2. Verify that the message was correct. That it contains the required claims and that all claims are of the correct data type. If it's signed and/or encrypted verify signature and/or decrypt.
3. Store the received information in a data base and/or passes it on to the application.

oidcservice is built to allow clients to be constructed that supports any number and type of of request-response services. The basic Open ID Connect set is:

- Webfinger
- Dynamic provider information discovery
- Dynamic client registration
- Authorization/Authentication request
- Access token request
- User info request

To these one can add services like session management and token introspection. The only thing we can be sure of is that this is not the final set of services, there will be more. And there will be variants of the standard ones. Like when you want to add multi lateral federation support to provider information discovery and client registration.

Over all it seemed like a good idea to write a piece of code that implements all the functionality that is needed to support any of this services and any future services that follows the same pattern.

That is the thought behind **oidcservice**.

CHAPTER 1

# ServiceContext

## 1.1 Introduction

ServiceContext holds information that is necessary for a OAUth2 client or an OpenID Connect (OIDC) RP to work properly. When an RP/Client receives information from an OP/AS it will store all or parts of it in the ServiceContext. When it constructs requests it will use the ServiceContext to find values for parameters in the request.

## 1.2 Content of the ServiceContext

There are a number of distinct parts of information in the ServiceContext. One can group them into a couple of groups:

- Information about the client
- Information about the OP/AS

Information about the client and the OP/AS can be gotten in two different ways. Either by static configuration or by dynamic discovery/registration.

### 1.2.1 Client Information

These are the ServiceContext parameters that deals with information about the client.

**Note:** Even though I talk about RPs below most of the things I describe is equally valid for OAUth2 Clients.

The information can broadly be grouped into two groups. The first being information about the client that is unconnected to which OP/AS the RP/Client interacts with, those are:

**base_url**

- This is the part of URLs that the client presents to the outside that doesn't vary between the URLs.

**requests_dir**

- If the request_uri parameter is used then signed JWTs will be stored in this directory.

The second group of parameters then is OP/AS dependent

**allow**

- This is used to make the client allow divergence from the standards. The only present use if for non-matching issuer values. According to the OIDC standard the value of iss in an ID Token must be the same as the Issuer ID of the OP. The value of allow is a dictionary
- example: allow={"issuer_missmatch": True}

**keyjar** A container for all the keys that the RP needs. To begin with the key jar will only contain keys that is owned by the RP. Over time it will also be populated with keys used by the OP

**client_id**

- The identifier of the client. This value is not globally unique but only unique for a special RP-OP combination.
- The client ID can either be returned by a out-of-band registration service connected to the OP or during OIDC dynamic client registration.
- There must be a client ID

**client_secret**

- There may be a client secret.
- The client secret can be used as a symmetric key in symmetric key cryptography or as a password while doing client authentication.
- As with client_id the client secret can either be returned by some out-of-band registration service together with the client_id or obtained during OIDC dynamic client registration

**redirect_uris**

- A list or URLs to where the OP should redirect the user agent after the authorization/authentication process has completed.

**callback**

- Depending on the response_type and response_mode used you may want to pick one specific redirect_uri out of a given set.
- The keys presently understood by the system are the ones listed in the example.
- Example:

```
{
    "code": "https://example.com/authz_cb",
    "implicit": "https://example.com/authz_im_cb",
    "form_post": "https://example.com/authz_fp_cb"
}
```

**registration_response**

- A positive response received from the OP on a client registration request.
- This is an oidcmsg.oidc.RegistrationResponse instance
- The possible content here is described in http://openid.net/specs/openid-connect-registration-1_0.html#ClientMetadata

**client_secret_expires_at**

- Set when a positive client registration has been received and the OP has added client_secret_expires_at to the response

**registration_access_token**

- Set when a positive client registration has been received and the OP has added registration_access_token to the response

**behaviour**

- If OIDC dynamic client registration is not supported by the OP or if dynamic registration is not used then this is where necessary information about how the RP should behave against the OP must be stored.
- If dynamic client registration is used then the result after matching the registration response against the client_preferences are store here.
- Example:

```
{
   "response_types": ["code"],
   "scope": ["openid", "profile", "email"],
   "token_endpoint_auth_method": ["client_secret_basic",
                                  'client_secret_post']
}
```

**client_preferences**

- When dynamic client registration is used this is where it's specified what should be sent in the registration request. This information will be added to before sending it to the OP, more about that below. The format is the same as for behaviour.
- The possible content is described in http://openid.net/specs/openid-connect-registration-1_0.html#ClientMetadata
- Example:

```
{
   "application_type": "web",
   "application_name": "rphandler",
   "contacts": ["ops@example.com"],
   "response_types": ["code", "id_token", "id_token token",
                      "code id_token", "code id_token token",
                      "code token"],
   "scope": ["openid", "profile", "email", "address", "phone"],
   "token_endpoint_auth_method": ["client_secret_basic",
                                  "Client_secret_post"],
}
```

**NOTE:** If you do static client configuration you **MUST** define behaviour in configuration.

If you do dynamic client registration you **MAY** use *behaviour* and you should use *client_preferences*. The result of matching the client_preferences with registration response will be used to update *behaviour*.

### 1.2.2 OP information

Basically only 2 pieces of information:

**issuer**

- The issuer ID of the OP. This must be an URL.
- **This is found by using WebFinger, by some other issuer discovery service** or by static configuration.

**provider_info**

- **This is either statically configured or obtained by using OIDC provider** info discovery.
- Should be a oidcmsg.oidc.ProviderConfigurationResponse instance
- **The possible content is described in** http://openid.net/specs/openid-connect-discovery-1_0.html#ProviderMetadata

### 1.2.3 Session information

Stored in the state_db database. The database should be some kind of persistent data storage. For testing an in-memory database is OK but not for production.

The database must be of the key-value type. The key into the session state information is the value of the state parameter in the authorization request.

The following data is stored per session:

**client_id** Client ID

**iss** Issuer ID

**iat** When the entry in the state_db was created

**response_type** The response_type specified in the authorization request

**scope** The scope specified in the authorization request

**redirect_uri** The redirect_uri used in the authorization request

**token**

- Information about the access token received
- Example:

```
{'access_token': 'Z0FBQUFBQmFkdFF', 'token_type': 'Bearer',
 'scope': ['openid']}
```

**id_token** The received ID Token as a signed JWT

## 1.3 Using the ServiceContext

The objects that use the ServiceContext are the oidcservice.service.Service instances. These object read and write to the ServiceContext while a session is active.

Below I'll go through the interaction between a certain type of service and the ServiceContext. There interaction takes place when the service is constructing a request and when after having parsed the response it wants to update the ServiceContext.

### 1.3.1 WebFinger

#### Constructing request

If Webfinger is used then nothing but an identifier for a user is in place so the ServiceContext doesn't contain any useful information.

#### Updating the ServiceContext

If the WebFinger request got a positive response then the URL which is the OP issuer ID is now know and will be stored in ServiceContext.issuer .

### 1.3.2 ProviderInfoDiscovery

There are 2 paths here, either the information is provided in the configuration setup or the information is expected to be fetch using OIDC dynamic provider info discovery.

If it's in the configuration updating the Service Context consists of initiating a oidcmsg.oidc.ProviderConfigurationResponse class with the provided information. Setting ServiceContext.issuer to the issuer value provided in the configuration and adding the oidcmsg.oidc.ProviderConfigurationResponse instance as value to ServiceContext.provider_info.

If discovery is done then the following happens:

#### Constructing request

The URL that is the Issuer ID is picked from ServiceContext.issuer and the ".well-known/openid-configuration" path is added to the the URL. The resulting URL is then used for the discovery request

#### Updating the ServiceContext

The parsed response, if it is an oidcmsg.oidc.ProviderConfigurationResponse instance is added to ServiceContext.provider_info. Also if dynamic client registration is to be used and therefor ServiceContext.client_preferences has been define this is where the preferences together with the provider info response are converted into a ServiceContext.behaviour value.

### 1.3.3 Registration

As for ProviderInfoDiscovery there are 2 possible path. The first using static client registration in which case all the necessary information must be included in the configuration. As a similar process to what happens in ProviderInfoDiscovery a oidcmsg.oidc.RegistrationResponse instance is created with the information in the configuration.

If dynamic client registration is to happen, then the following happens.

#### Constructing request

Apart from the information given in client_preferences some more information are gathered from the ServiceContext. From ServiceContext.provider_info we get:

**authorization_endpoint** This just so we know where to send the user-agent

**require_request_uri_registration** If this is set to True we need to construct request_uris and add them to the registration request

From ServiceContext you can get *redirect_uris* and/or *callback*. Depending on what is configured a set of *redirect_uris* are added to the request

Same goes for *post_logout_redirect_uris*

### Updating the ServiceContext

The parsed registration response if it was positive is stored in ServiceContext.registration_response. Sets the following parameters in ServiceContext if present in the registration response:

- client_id
- client_secret
- client_secret_expires_at
- registration_access_token

Also if *token_endpoint_auth_method*

CHAPTER 2

# The Service class

This class contains 2 pipe lines, one for the request construction and one for response parsing. The interface to HTTP is kept to a minimum to allow users of oidcservice to chose their favorite HTTP client/server libraries.

The class has a number of attributes:

**msg_type** The message subclass that describes the request. Default is oicmsg.message.Message

**response_cls** The message subclass that describes the response Default is oicmsg.message.Message

**error_msg** The message subclass that describes an error response Default is oicmsg.message.oauth2.ErrorResponse

**endpoint_name** The name of the endpoint on the server that the request should be sent to. No default

**synchronous** *True* if the response will be returned as a direct response to the request. The only exception right now to this is the Authorization request where the response is delivered to the client at some later date. Default is *True*

**request** A name of the service. Later when a RP/client is implemented instances of different services are found by using this name. No default

**default_authn_method** The client authentication method to use if nothing else is specified. Default is '' which means none.

**http_method** Which HTTP method to use when sending the request. Default is **GET**

**request_body_type** The serialization method to be used for the request Default is *urlencoded*

**response_body_type** The deserialization method to use on the response Default is *json*

## 2.1 The request pipeline

Below follows a desciption of the parts of the request pipeline in the order they are called.

The overall call sequence looks like this:

- **get_request_parameters**

- **construct_request**
  - **construct**
    - do_pre_construct (*)
    - gather_request_args
    - do_post_construct (*)
- get_http_url
- get_authn_header
- get_http_body

The result of the request pipeline is a dictionary that in its simplest form will look something like this:

```
{
    'url' : 'https://example.com/authorize?response_type=code&state=state&client_
↪id=client_id&scope=openid&redirect_uri=https%3A%2F%2Fexample.com%2Fcli%2Fauthz_cb&
↪nonce=P1B1nPCnzU4Mwg1hjzxkrA3DmnMQKPWl'
}
```

It will look like that when the request is to be transmitted as the urlencoded query part of a HTTP GET operation. If instead a HTTP POST with a json body is expected the outcome of *get_request_parameters* will be something like this:

```
{
    'url': 'https://example.com/token',
    'body': 'grant_type=authorization_code&redirect_uri=https%3A%2F%2Fexample.com
↪%2Fcli%2Fauthz_cb&code=access_code&client_id=client_id',
    'headers': {'Authorization': 'Basic Y2xpZW50X2lkOnBhc3N3b3Jk', 'Content-Type':
↪'application/x-www-form-urlencoded'}
}
```

Here you have the url that the request should go to, the body of the request and header arguments to add to the HTTP request.

### 2.1.1 get_request_parameters

Implemented in `oidcservice.service.Service.get_request_parameters()`

Nothing much happens locally in this method, it starts with gathering information about which HTTP method is used, the client authentication method and the how the request should be serialized.

It the calls the next method

#### construct_request

Implemented in `oidcservice.service.Service.construct_request()`

The method where most is done leading up to the sending of the request. The request information is gathered and the where to and how of sending the request is decided.

#### construct

Implemented in `oidcservice.service.Service.construct()`

Instantiate the request as a message class instance with attribute values from the message call and gathered by the *pre_construct* methods and the *gather_request_args* method and possibly modified by a *post_construct* method.

### do_pre_construct

Implemented in `oidcservice.service.Service.do_pre_construct()`

Updates the arguments in the method call with preconfigure argument from the client configuration.

Then it will run the list of pre_construct methods one by one in the order they appear in the list.

The call API that all the pre_construct methods must adhere to is:

```
meth(request_args, service_context, **_args)
```

service_context is an instance of `oidcservice.service_context.ServiceContext` The methods MUST return a tuple with request arguments and arguments to be used by the post_construct methods.

### gather_request_args

Implemented in `oidcservice.service.Service.gather_request_args()`

Has a number of sources where it can get request arguments from. In priority order:

1. Arguments to the method call
2. Information kept in the service context instance
3. Information in the client configuration targeted for this method.
4. Standard protocol defaults.

It will go through the list of possible (required/optional) attributes as specified in the oicmsg.message.Message class that is defined to be used for this request and add values to the attributes if any can be found.

### do_post_construct

Implemented in `oidcservice.service.Service.do_post_construct()`

These methods are there to do modifications to the request that can not be done until all request arguments have been gathered. The prime example of this is to construct a signed Jason Web Token to be add as value to the *request* parameter or referenced to by *request_uri*.

### get_authn_header

Implemented in `oidcservice.service.Service.get_authn_header()`

oidcservice supports 6 different client authentication/authorization methods.

2 from https://tools.ietf.org/html/rfc6750:

- bearer_body
- bearer_header

and these described in http://openid.net/specs/openid-connect-core-1_0.html#ClientAuthentication:

- client_secret_basic

- client_secret_jwt
- client_secret_post
- private_key_jwt

Depending on which of these, if any, is supposed to be used, different things has to happen.

#### get_http_url

Implemented in `oidcservice.service.Service.get_http_url()`

Depending on where the request are to be placed in the request (part of the URL or as a POST body) and which serialization is to be used, the request in it's proper format will be constructed and tagged with destination.

## 2.2 The response pipeline

Below follows a description of the response pipeline methods in the order they are called.

The overall call sequence looks like this:

- ***parse_response***
  - *get_urlinfo*
  - *do_post_parse_response* (#)
- *parse_error_mesg*

### 2.2.1 parse_response

Will initiate a *response_cls* instance with the result of deserializing the result. If the response turned out to be an error response even though the status_code was in the 200 <= x < 300 range that is dealt with and an *error_msg* instance is instantiated with the response.

Either way the response is verified (checked for required parameters and parameter values being of the correct data types) and if it was not an error response *do_post_parse_response* is called.

#### get_urlinfo

Picks out the query or fragment component from an URL

#### do_post_parse_response

Runs the list of *post_parse_response* methods in the order they appear in the list.

The API of these methods are:

```
method(response, service_context, state=state, **_args)
```

The parameters being:

**response** A Message subclass instance

**service_context** A `oidcservice.service_context.ServiceContext` instance

**state** The state value that was used in the authorization request

**_args** A set of extra keyword arguments

### 2.2.2 parse_error_mesg

Parses an error message return with a 4XX error message. OAuth2 expects 400 errors, OpenID Connect also uses a 402 error. But we accept the full range since serves seems to be able to use them all. Also there are OP/AS implementations that return error messages in a HTTP 200 response.

CHAPTER 3

# The state data base

All services are running in a context. A set of services that runs in a sequence defines a session. Each such session must be able to keep information on what has happened in order to be able to know what the next step should be. So each service instance must be able to access a data storage. This data storage is in our model provided by the RP implementation. What is defined here is the interface to that data storage.

## 3.1 Data format

The data store is of the key-value format where the keys are strings and the values are JSON documents (http://json.org). We reuse our knowledge on how to construct messages and serialise/deserialise them that we have from oidcmsg.

The basic message is defined by:

```
from oidcmsg.message import Message
from oidcmsg.message import SINGLE_OPTIONAL_JSON
from oidcmsg.message import SINGLE_REQUIRED_STRING


class State(Message):
    c_param = {
        'iss': SINGLE_REQUIRED_STRING,
        'auth_request': SINGLE_OPTIONAL_JSON,
        'auth_response': SINGLE_OPTIONAL_JSON,
        'token_response': SINGLE_OPTIONAL_JSON,
        'refresh_token_request': SINGLE_OPTIONAL_JSON,
        'refresh_token_response': SINGLE_OPTIONAL_JSON,
        'user_info': SINGLE_OPTIONAL_JSON
    }
```

Additional attributes and values may be added to this base class by service extensions.

## 3.2 Methods

We defined two methods; *set* and get* to be used like this:

```
$ from oidcservice.service import State
$ _state = State(iss='issuer_id')
$ state_db.set('abcdef', _state.to_json())
```

and then sometime later:

```
$ _json = state_db.get('abcdef')
$ _state = State().from_json(_json)
$ print(_state['iss'])
'issuer_id'
```

If a get is done with a key that does not exist in the data base, a None value will be returned.

If something stored in the database must be modified it has to be read from the database, modified locally and then written back to the database.

Anything in the database will be silently overwritten by a new *set* command.

CHAPTER 4

# What a service should do

This document is a companion to **oidcservice-service_** where the how is covered. This document aims to tell you the what. What is divided into two part one is what happens before the HTTP request is sent and the other is what happens once the response has been reveived. In the following I am concentrating on what happens after a positive response. What happens if an error message is received is not covered here.

The services will be covered one by one it the order they normally appear in a conversation between an OpenID Connect (OIDC) Relying Party (RP) and an OpenID Provider (OP).

For the examples it will be assumed that we have an instance of `oidcservice.service_context.ServiceContext` called *service_context* . The service will also have access to a database (*state_db*) where information about the communication between the RP and the OP is stored on a per End-User basis.

Each service will verify the correctness of the response regarding:

- presence of required claims
- correct value types and in certain context also that the value is from a given set.
- If the response is a signed and/or encrypted JWT, decrypt and verify signature.

This is all done by using services provided by oidcmsg and cryptojwt.

## 4.1 OpenID Provider Issuer discovery

### 4.1.1 Service description

OpenID Provider Issuer discovery is the process of determining the location of the OpenID Provider as described in discovery . In this document it is described how you can use Webfinger to implement this service.

OP issuer discovery is implemented in `oidcservice.oidc.service.WebFinger`

### 4.1.2 Pre request work

The process starts with the End-User supplying the RP with an identifier. The service will then apply normalization rules to the identifier to determine the resource and the host. This together with the defined *rel* value allows the service to construct the URL that can be used to get the location of the requested service.

In discovery you can find a number of examples of how this is done.

### 4.1.3 Post request work

A positive response from a WebFinger resource returns a JavaScript Object Notation (json) object describing the entity that is queried. The JSON object is referred to as the JSON Resource Descriptor (JRD). An example would be:

```
{
    "subject": "https://example.com/joe",
    "links":
    [
        {
            "rel": "http://openid.net/specs/connect/1.0/issuer",
            "href": "https://server.example.com"
        }
    ]
}
```

The service will parse this JSON object and store the *href* value in the **oidcservice-service-context_** as value on *issuer*.:

```
# _jrd is A parsed JRD response
OIDC_REL = "http://openid.net/specs/connect/1.0/issuer"

for link in _jrd['links']:
    if link['rel'] == OIDC_REL:
        service_context.issuer = link['href']
        break
```

## 4.2 Obtaining OpenID Provider Configuration Information

### 4.2.1 Service description

Some way the RP has gotten the Issuer location. It could have been using WebFinger as described above but there are many other ways of doing this too. No matter what as used we assume the Issuer location is known to the RP and we now want to find the metadata for the Issuer. This is where this service comes in.

Obtaining OpenID Provider Configuration Information is implemented in `oidcservice.oidc.service.ProviderInfoDiscovery`

### 4.2.2 Pre request work

Not really anything to do here. The only thing necessary to do is to construct the URL to use and that is done by adding a path component to the Issuer location.

The path component is: */.well-known/openid-configuration*

### 4.2.3 Post request work

The response is a JSON object that contains a set of claims that are a subset of the Metadata values defined in section 3 of discovery . Other claims may be returned.

This service parses the JSON object using `oidcmsg.oidc.ProviderConfigurationResponse`. Verifies it's correctness and then it does a number of checks:

1. Validates that the value of *issuer* returned in the response is the same as the issuer location
2. Verifies that the RP will be able to talk to the OP. Like supporting the crypto algorithms favored by the OP.
3. Verifies that the endpoint URLs are HTTPS URLs
4. If a jwks_uri is given verify that it points to a syntactically correct JWKS

Using the information in the response the service is also expected to combine what the OP can do and what the RP prefers to do (according to the configuration) and produce a description of the behaviour of the RP.

And lastly set the correct endpoints for all the services.

## 4.3 Authentication Request

### 4.3.1 Service description

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.

Authentication is implemented by `oidcservice.oidc.service.Authorization`

### 4.3.2 Pre request work

There are a number of things that must be done dynamically when the RP is constructing an authentication request.

If it's the first time the RP sends such a request to the OP it should as described by oauth security create OP specific redirect_uris.

For each request it **MUST** create a new *state* value and possibly also a *nonce* if ID Token is expected to be returned by the OP.

These attributes it can pick from configuration and from the OP metadata:

- authorization_endpoint
- scope
- response_type
- response_mode (if different from the default given by the choice of response_type)

If the request or rrequest_uri parameters are used the this service will construct the signed JSON Web Token.

And finally the service will store the request in the state_db using the *state* value as key.

### 4.3.3 Post request work

If *expires_in* is provided in the response and extra attribute *__expires_at* is added to the response.

The response as a whole is added to the *state_db* database.

## 4.4 Token Request

### 4.4.1 Service description

To obtain an Access Token, an ID Token, and optionally a Refresh Token, the RP sends a Token Request to the Token Endpoint to obtain a Token Response.

Token request is implemented by `oidcservice.oidc.service.AccessToken`

### 4.4.2 Pre request work

Fetches the necessary claim values from the authentication request/response copies in the *state_db*.

This includes claims like *code* and *redirect_uri*.

*client_id* and *client_secret* is picked from the client registration response.

Depending on which client authentication methods the RP is expected to use the necessary information is constructed.

### 4.4.3 Post request work

The ID Token is validated using the process described in section 3.1.3.7 in OIDC core.

If *expires_in* is provided in the response and extra attribute *__expires_at* is added to the response.

The response as a whole is added to the *state_db* database.

## 4.5 Refresh Request

### 4.5.1 Service description

To refresh an Access Token, the RP sends a Refresh Token Request to the Token Endpoint to obtain a Token Response.

Token request is implemented by `oidcservice.oidc.service.RefreshAccessToken`

### 4.5.2 Pre request work

Fetches the necessary claim values from the authentication request/response and possibly the access token response (depends on which flow that was used) copies in the *state_db*.

This includes claims like *code*, *redirect_uri* and *refresh_token*.

*client_id* and *client_secret* is picked from the client registration response.

Depending on which client authentication methods the RP is expected to use the necessary information is constructed.

### 4.5.3 Post request work

If *expires_in* is provided in the response and extra attribute *__expires_at* is added to the response.

The response as a whole is added to the *state_db* database.

## 4.6 User info Request

### 4.6.1 Service description

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.

### 4.6.2 Pre request work

Fetches the access token from the authentication or the access token response depending on which flow was used.

### 4.6.3 Post request work

The response as a whole is added to the *state_db* database.

# CHAPTER 5

# A conversation

This section will walk you through what might happen when a user wants to use OIDC to authenticate/authorize and the Relying Party (RP) has never seen the OpenID Connect Provider (OP) before. This is an example of how dynamic the interaction between an RP and an OP can be using OIDC.

We start from knowing absolutely nothing, having to use WebFinger to find the OP. Then follows dynamic provider info discovery and client registration before the user can be brought in and do the authentication/authorization bit. And lastly the RP will ask for an access token and after that information about the user.

## 5.1 Initial setup

We need a couple of things initiated before we start.

**state_db instance** For this example we have an in-memory data store:

```
class DB(object):
def __init__(self):
    self.db = {}

def set(self, key, value):
    self.db[key] = value

def get(self, item):
    try:
        return self.db[item]
    except KeyError:
        return None
```

**service_context** Which is where information common to more then one service is kept. A `oidcservice.service_context.ServiceContext` instance:

```
BASEURL = "https://example.org/rp
service_context = ServiceContext(
```

(continues on next page)

(continued from previous page)

```
    KEYJAR,
    {
        "client_prefs":
            {
                "application_type": "web",
                "application_name": "rphandler",
                "contacts": ["ops@example.org"],
                "response_types": ["code"],
                "scope": ["openid", "profile", "email", "address", "phone"],
                "token_endpoint_auth_method": ["client_secret_basic",
                                               'client_secret_post'],
            },
        "redirect_uris": ["{}/authz_cb".format(BASEURL)],
        "jwks_uri": "{}/static/jwks.json".format(BASEURL)
    }
)
```

**service specifications** A dictionary of service class names and service configurations:

```
service_spec = {
    'WebFinger': {},
    'ProviderInfoDiscovery': {},
    'Registration': {},
    'Authorization': {},
    'AccessToken': {},
    'UserInfo': {}
}
```

To initiate the services we need to run:

```
from oidcservice.oic.service import factory
from oidcservice.state_interface import InMemoryStateDataBase

service = build_services(service_spec, factory,
                         state_db=InMemoryStateDataBase(),
                         service_context=service_context)
```

The resulting **service** is a dictionary with services identifiers as keys and `oidcservice.service.Service` instances as values:

```
$ set(service.keys())
{'accesstoken', 'authorization', 'webfinger', 'registration', 'userinfo', 'provider_
↪info'}
```

That's all we have to do when it comes to setup so now on to the actual conversation.

## 5.2 Webfinger

We will use WebFinger (RFC7033) to find out where we can learn more about the OP. What we have to start with is an user identifier provided by the user. The identifier we got was: **foobar@example.com** . With this information we can do:

```
info = service['webfinger'].get_request_parameters(service_context, resource=
↪'foobar@example.com')
```

service['webfinger'] will return the WebFinger service instance and running the method *get_request_parameters* will return the information necessary to do a HTTP request. In this case the value of *info* will be:

```
{
    'url': 'https://example.com/.well-known/webfinger?resource=acct%3Afoobar
↪%40example.com&rel=http%3A%2F%2Fopenid.net%2Fspecs%2Fconnect%2F1.0%2Fissuer'
}
```

as you can see the *get_request_parameters* constructed a URL that can be used to get the wanted information.

Doing HTTP GET on this URL will return a JSON document that looks like this:

```
{
"subject": "acct:foobar@example.com",
"links": [{"rel": "http://openid.net/specs/connect/1.0/issuer",
           "href": "https://example.com"}],
"expires": "2018-02-04T11:08:41Z"}
```

To parse and use it I can run another method provide by the service instance:

```
response = service['webfinger'].parse_response(webfinger_response,
                                               service_context)
```

It's assumed that *webfinger_response* contains the JSON document mentioned above. *parse_response* only parses the response. So apart from that method we also need to invoke *update_service_context*:

```
service['webfinger'].update_service_context(response)
```

The result of this is that the information in **service_context** will change. We now has this:

```
service_context.issuer: "https://example.com"
```

And that is all we need to fetch the provider info

## 5.3 Provider info discovery

We use the same process as with webfinger but with another service instance:

```
info = service['provider_info'].get_request_parameters()
```

*info* will now contain:

```
{'url': 'https://example.com/.well-known/openid-configuration'}
```

And this is the first example of **magic** that you will see.

*get_request_parameters knows how to contruct the OpenID Connect providers discovery URL from information stored in the service_context instance. Now, if you don't wanted to do webfinger because for instance the other side did not provide that service. Then you would have to set *service_context.issuer* to the correct value.*

Doing HTTP GET on the provided URL should get us the provider info. It does and we get a JSON document that looks something like this:

```
{
"version": "3.0",
"token_endpoint_auth_methods_supported": [
```

(continues on next page)

(continued from previous page)

```
    "client_secret_post", "client_secret_basic",
    "client_secret_jwt", "private_key_jwt"],
"claims_parameter_supported": True,
"request_parameter_supported": True,
"request_uri_parameter_supported": True,
"require_request_uri_registration": True,
"grant_types_supported": ["authorization_code",
                          "implicit",
                          "urn:ietf:params:oauth:grant-type:jwt-bearer",
                          "refresh_token"],
"response_types_supported": ["code", "id_token",
                             "id_token token",
                             "code id_token",
                             "code token",
                             "code id_token token"],
"response_modes_supported": ["query", "fragment",
                             "form_post"],
"subject_types_supported": ["public", "pairwise"],
"claim_types_supported": ["normal", "aggregated",
                          "distributed"],
"claims_supported": ["birthdate", "address",
                     "nickname", "picture", "website",
                     "email", "gender", "sub",
                     "phone_number_verified",
                     "given_name", "profile",
                     "phone_number", "updated_at",
                     "middle_name", "name", "locale",
                     "email_verified",
                     "preferred_username", "zoneinfo",
                     "family_name"],
"scopes_supported": ["openid", "profile", "email",
                     "address", "phone",
                     "offline_access", "openid"],
"userinfo_signing_alg_values_supported": [
    "RS256", "RS384", "RS512",
    "ES256", "ES384", "ES512",
    "HS256", "HS384", "HS512",
    "PS256", "PS384", "PS512", "none"],
"id_token_signing_alg_values_supported": [
    "RS256", "RS384", "RS512",
    "ES256", "ES384", "ES512",
    "HS256", "HS384", "HS512",
    "PS256", "PS384", "PS512", "none"],
"request_object_signing_alg_values_supported": [
    "RS256", "RS384", "RS512", "ES256", "ES384",
    "ES512", "HS256", "HS384", "HS512", "PS256",
    "PS384", "PS512", "none"],
"token_endpoint_auth_signing_alg_values_supported": [
    "RS256", "RS384", "RS512", "ES256", "ES384",
    "ES512", "HS256", "HS384", "HS512", "PS256",
    "PS384", "PS512"],
"userinfo_encryption_alg_values_supported": [
    "RSA1_5", "RSA-OAEP", "RSA-OAEP-256",
    "A128KW", "A192KW", "A256KW",
    "ECDH-ES", "ECDH-ES+A128KW", "ECDH-ES+A192KW", "ECDH-ES+A256KW"],
"id_token_encryption_alg_values_supported": [
    "RSA1_5", "RSA-OAEP", "RSA-OAEP-256",
```

(continues on next page)

(continued from previous page)

```
    "A128KW", "A192KW", "A256KW",
    "ECDH-ES", "ECDH-ES+A128KW", "ECDH-ES+A192KW", "ECDH-ES+A256KW"],
"request_object_encryption_alg_values_supported": [
    "RSA1_5", "RSA-OAEP", "RSA-OAEP-256", "A128KW",
    "A192KW", "A256KW", "ECDH-ES", "ECDH-ES+A128KW",
    "ECDH-ES+A192KW", "ECDH-ES+A256KW"],
"userinfo_encryption_enc_values_supported": [
    "A128CBC-HS256", "A192CBC-HS384", "A256CBC-HS512",
    "A128GCM", "A192GCM", "A256GCM"],
"id_token_encryption_enc_values_supported": [
    "A128CBC-HS256", "A192CBC-HS384", "A256CBC-HS512",
    "A128GCM", "A192GCM", "A256GCM"],
"request_object_encryption_enc_values_supported": [
    "A128CBC-HS256", "A192CBC-HS384", "A256CBC-HS512",
    "A128GCM", "A192GCM", "A256GCM"],
"acr_values_supported": ["PASSWORD"],
"issuer": "https://example.com",
"jwks_uri": "https://example.com/static/jwks_tE2iLbOAqXhe8bqh.json",
"authorization_endpoint": "https://example.com/authorization",
"token_endpoint": "https://example.com/token",
"userinfo_endpoint": "https://example.com/userinfo",
"registration_endpoint": "https://example.com/registration",
"end_session_endpoint": "https://example.com/end_session"}
```

Quite a lot of information as you can see. We feed this information into *parse_response* and *update_service_context* and let them do their business:

```
resp = service['provider_info'].parse_response(json_document)

service['provider_info'].update_service_context(resp)
```

*json_document* contains the JSON document from the HTTP response. *parse_response* will parse and verify the response. One such verification is to check that the value provided as **issuer** is the same as the URL used to fetch the information without the '.well-known' part. In our case the exact value that the webfinger query produced.

As with the *webfinger* service *update_service_context* adds things to **service_context**. So we now have:

```
service_context.provider_info['issuer']: https://example.com
service_context.provider_info['authorization_endpoint']: https://example.com/
↪authorization
```

As you can guess from the above the whole response from the OP was stored in the service_context instance. Such that it is easily accessible in the future.

Now we know what we need to know to register the RP with the OP. If the OP had not provided a 'registration_endpoint' it would not have supported dynamic client registration but this one has so it does.

## 5.4 Client registration

By now you should recognize the pattern:

```
info = service['registration'].get_request_parameters()
```

Now *info* contains 3 parts:

**uri** The URL to which the HTTP request should be sent

**body** A JSON document that should go in the body of the HTTP request

**http_args:** HTTP arguments to be used with the request

and we got:

```
uri: https://example.com/registration
body: {
    "application_type": "web",
    "response_types": ["code"],
    "contacts": ["ops@example.org"],
    "jwks_uri": "https://example.org/static/jwks.json",
    "token_endpoint_auth_method":
    "client_secret_basic",
    "redirect_uris": ["https://example.org/authz_cb"]
    }
http_args: {'headers': {'Content-Type': 'application/json'}}
```

The information in the body comes from the client configuration. If we use this information and does an HTTP POST to the provided URL we will receive a response like this:

```
{
"client_id": "zls2qhN1jO6A",
"client_secret": "c8434f28cf9375d9a7f3b50dcfdf6a20d6e702e310066874f794817f",
"registration_access_token": "NdGrGR7LCuzNtixvBFnDphGXv7wRcONn",
"registration_client_uri": "https://localhost:8080/oicrp/registration?client_
↪id=zls2qhN1jO6A",
"client_secret_expires_at": 1517823388,
"client_id_issued_at": 1517736988,
"application_type": "web",
"response_types": ["code"],
"contacts": ["ops@example.com"],
"token_endpoint_auth_method": "client_secret_basic",
"redirect_uris": ["https://example.com/authz_cb"]
}
```

Again a JSON document. This is the OP's response to the RP's registration request.

We stuff the response into *json_document* and feed it to *parse_response* which will parse, verify and interpret the response and then *update_service_context* which updates *service_context*:

```
response = service['registration'].parse_response(json_document,
                                                  service_context)
service['registration'].update_service_context(response)
```

The information stored in *service_context* is most under the heading *registration_response* but some, more important, will be stored at a directly reachable place:

```
service_context.client_id: zls2qhN1jO6A
service_context.client_secret:␣
↪c8434f28cf9375d9a7f3b50dcfdf6a20d6e702e310066874f794817f
```

By that we have finalized the dynamic discovery and registration now we can get down to doing the authentication/authorization bits.

## 5.5 Authorization

In the following example I'm using code flow since that allows me to show more of what the oidcservice package can do.

Like when I used the other services this one is no different:

```
info = service['authorization'].get_request_parameters(service_context)
```

*info* will only contain one piece of data and that is a URL:

```
uri: https://example.com/authorization?state=Oh3w3gKlvoM2ehFqlxI3HIK5&
↪nonce=UvudLKz287YByZdsY3AJoPAlEXQkJ0dK&response_type=code&client_id=zls2qhN1jO6A&
↪scope=openid&redirect_uri=https%3A%2F%2Fexample.org%2Fauthz_cb
```

Where did all the information come from ?:

- the authorization endpoint comes from the dynamic provider info discovery,
- client_id from the client registration,
- response_type, scope and redirect_uri from the client configuration and
- state and nonce are dynamically created by the service instance.

When this *service* instance creates a request it will also create a *session* instance in *state_db* keyed on the state value.

I do HTTP GET on the provided URL and will eventually get redirected back to the RP with the response in the query part of the redirect URL. Below you have just the query component:

```
state=Oh3w3gKlvoM2ehFqlxI3HIK5&scope=openid&
↪code=Z0FBQUFBQmFkdFFjUVpFWE81SHU5N1N4N01aQWJ1Y3Y1MWFfMTVXXzhEcll2a0lkd0Z2Qk9lOHYtTUZjRnRjUzhNc1FOdr
↪%3D&iss=https%3A%2F%2Fexample.com&client_id=zls2qhN1jO6A
```

I feed the *query_part* into the *parse_response* method of the authorization service instance and hope for the best:

```
_resp = service['authorization'].parse_response(query_part)
service['authorization'].update_service_context(_resp)
```

Now as mentioned above one thing that happened when the authorization request was constructed was that some information of that request got stored away with the *state* value as key. All in the state_db instance.

The response on the authorization query will be stored in the same place. To get the code I can now use:

```
from oidcmsg.oidc import AuthorizationResponse

authn_response = service_context.get_item(AuthorizationResponse,
                                          'auth_response',
                                          'Oh3w3gKlvoM2ehFqlxI3HIK5')
code = authn_response['code']
```

State information will be use when we take the next step, which is to get an access token.

## 5.6 Access token

When sending an access token request I have to use the correct *code* value. To accomplish that *get_request_parameters* need to get state as an argument:

```
request_args = {'state': _state}

info = service['accesstoken'].get_request_parameters(service_context,
                                                     request_args=request_args)
```

The OIDC standard says that the *redirect_uri* used for the authorization request should be provided in the access token request, therefor the service will add it if I don't.

This time *info* has these parts:

```
uri: https://example.com/token
body: grant_type=authorization_code&state=Oh3w3gKlvoM2ehFqlxI3HIK5&redirect_uri=https
↪%3A%2F%2Fexample.org%2Fauthz_cb&
↪code=Z0FBQUFBQmFkdFFjUVpFWE81SHU5N1N4N01aQWJ1Y3Y1MWFfMTVXXzhEcll2a0lkd0Z2Qk9lOHYtTUZjRnRjUzhNc1FOdr
↪%3D&client_id=zls2qhN1jO6A
http_args: {'headers': {'Authorization': 'Basic␣
↪emxzMnFoTjFqTzZBOmM4NDM0ZjI4Y2Y5Mzc1ZDlhN2YzYjUwZGNmZGY2YTIwZDZlNzAyZTMxMDA2Njg3NGY3OTQ4MTdm
↪', 'Content-Type': 'application/x-www-form-urlencoded'}}
```

*uri* was picked from the discovered provider info. The Authorization header looks like it does because the default client authentication method is defined to be 'client_secret_basic'. The body is, a bit surprising but according to the standard, urlencoded.

The response has this JSON document in the body:

```
{
'state': 'Oh3w3gKlvoM2ehFqlxI3HIK5',
'scope': 'openid',
'access_token':
↪'Z0FBQUFBQmFkdFFjc0hyU2lialZyUkhvQjliUjU2R3hTQWZ4cDZFMnRTdGxkV3VoQmppZllyN2htWHlhU2Ria0tRV2NqcjEwO0
↪',
'token_type': 'Bearer',
'id_token':
↪'eyJhbGciOiJSUzI1NiIsImtpZCI6IlEwMl92cXJIYlFpRk5kemZ4aFhUblhpMWphemZhTlFJMlNNa2NvNmMxdFEifQ.
↪eyJpc3MiOiAiaHR0cHM6Ly9sb2NhbGhvc3Q6ODA4MC9vaWNycC9ycC11c2VyaW5mby1iZWFyZXItaGVhZGVyIiwgInN1YiI6ICI
↪cOJYa-yNeVgHeitol2Zw3Z3TYh9Fxys8BwAmACSZEYzwNnt1DwSfvhLTOeSFcAh2vsrvmNh2HqOy4plnH5-
↪uB-KIEJY3E9GTmmK5uZDGvtSfMXqq2M45MA-71lJx2xrWwE5aH59WWJkEOY9s-gl0KJyMh7VFFP-B86d_
↪16rg2hB6y9ajH5ieR9mc_E0RdwZVDLF_uBcWj0tLiTH2AaZK4akCAiFUant261M2OQnreJ7D6WPfZl_
↪UHYPCm_6nhazvrQuovj9ahxAnqkg3UFBSycX4qr1brfi1Ak-
↪xKRdTQ08NYJwtC8JVxSM0ic3E2XsOIW0hThofKwQUiolWW4yq0Q',
}
```

We will deal with this in the now well know fashion:

```
_resp = service['accesstoken'].parse_response(json_document, state=
↪'Oh3w3gKlvoM2ehFqlxI3HIK5')

service['accesstoken'].update_service_context(_resp, state='Oh3w3gKlvoM2ehFqlxI3HIK5')
```

Note that we need to provide the methods with the *state* parameter so they will know where to find the correct information needed to verify the response and later store the received information.

Once the verification has been done one parameter will be added to the response before it is stored in the state database, namely:

```
'verified_id_token': {
    'iss': 'https://localhost:8080/oicrp/rp-userinfo-bearer-header',
    'sub': '1b2fc9341a16ae4e30082965d537ae47c21a0f27fd43eab78330ed81751ae6db',
```

(continues on next page)

(continued from previous page)

```
    'aud': ['zls2qhN1jO6A'],
    'exp': 1517823388,
    'acr': 'PASSWORD',
    'iat': 1517736988,
    'auth_time': 1517736988,
    'nonce': 'UvudLKz287YByZdsY3AJoPAlEXQkJ0dK'}
```

Here you have the content of the ID Token revealed.

And finally the last step, getting the user info.

## 5.7 User info

Again we have to provide the *get_request_parameters* method with the correct state value:

```
info = service['userinfo'].get_request_parameters(service_context,
                                                  state='Oh3w3gKlvoM2ehFqlxI3HIK5')
```

And the response is a JSON document:

```
{"sub": "1b2fc9341a16ae4e30082965d537ae47c21a0f27fd43eab78330ed81751ae6db"}
```

Only the *sub* parameter because the asked for scope was 'openid'.

Parsing, verifying and storing away the information is done the usual way:

```
_resp = service['userinfo'].parse_response(json_document,state=
↪'Oh3w3gKlvoM2ehFqlxI3HIK5')
service['userinfo'].update_service_context(_resp, state='Oh3w3gKlvoM2ehFqlxI3HIK5')
```

And we are done !! :-)

In the state_db we have the following information:

```
$ list(service['userinfo'].get_state(STATE).keys())
['iss', 'auth_request', 'auth_response', 'token_response', 'user_info']
```

CHAPTER 6

# oidcservice

## 6.1 oidcservice package

### 6.1.1 Subpackages

**oidcservice.oauth2 package**

**Submodules**

**oidcservice.oauth2.access_token module**

**oidcservice.oauth2.authorization module**

**oidcservice.oauth2.provider_info_discovery module**

**oidcservice.oauth2.refresh_access_token module**

**oidcservice.oauth2.utils module**

**Module contents**

**oidcservice.oic package**

**Submodules**

**oidcservice.oidc.access_token module**

**oidcservice.oidc.authorization module**

### 6.1.2 Submodules

### 6.1.3 oidcservice.client_auth module

### 6.1.4 oidcservice.service_context module

### 6.1.5 oidcservice.exception module

### 6.1.6 oidcservice.service module

### 6.1.7 oidcservice.util module

### 6.1.8 oidcservice.state_interface module

### 6.1.9 Module contents

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search