MSAL Python Documentation

Release 1.24.0

Microsoft

Sep 12, 2023

CONTENTS:

You can find high level conceptual documentations in the project README.

CHAPTER ONE

SCENARIOS

There are many different application scenarios. MSAL Python supports some of them. **The following diagram serves as a map. Locate your application scenario on the map. If the corresponding icon is clickable, it will bring you to an MSAL Python sample for that scenario.**

- Most authentication scenarios acquire tokens on behalf of signed-in users.
- There are also daemon apps. In these scenarios, applications acquire tokens on behalf of themselves with no user.
- There are other less common samples, such for ADAL-to-MSAL migration, available inside the project code base.

CHAPTER TWO

API REFERENCE

The following section is the API Reference of MSAL Python. The API Reference is like a dictionary. You **read this API section when and only when**:

- You already followed our sample(s) above and have your app up and running, but want to know more on how you could tweak the authentication experience by using other optional parameters (there are plenty of them!)
- You read the MSAL Python source code and found a helper function that is useful to you, then you would want to double check whether that helper is documented below. Only documented APIs are considered part of the MSAL Python public API, which are guaranteed to be backward-compatible in MSAL Python 1.x series. Undocumented internal helpers are subject to change anytime, without prior notice.

Note: Only APIs and their parameters documented in this section are part of public API, with guaranteed backward compatibility for the entire 1.x series.

Other modules in the source code are all considered as internal helpers, which could change at anytime in the future, without prior notice.

MSAL proposes a clean separation between public client applications and confidential client applications.

They are implemented as two separated classes, with different methods for different authentication scenarios.

CHAPTER

THREE

CLIENTAPPLICATION

class msal.**ClientApplication**(*client_id*, *client_credential=None*, *authority=None*, *validate_authority=True*, *token_cache=None*, *http_client=None*, *verify=True*, *proxies=None*, *timeout=None*, *client_claims=None*, *app_name=None*, *app_version=None*, *client_capabilities=None*, *azure_region=None*, *exclude_scopes=None*, *http_cache=None*, *instance_discovery=None*, *allow_broker=None*, *enable_pii_log=None*)

You do not usually directly use this class. Use its subclasses instead: `PublicClientApplication` and `ConfidentialClientApplication`.

__init__(*client_id*, *client_credential=None*, *authority=None*, *validate_authority=True*, *token_cache=None*, *http_client=None*, *verify=True*, *proxies=None*, *timeout=None*, *client_claims=None*, *app_name=None*, *app_version=None*, *client_capabilities=None*, *azure_region=None*, *exclude_scopes=None*, *http_cache=None*, *instance_discovery=None*, *allow_broker=None*, *enable_pii_log=None*)

Create an instance of application.

Parameters

- **client_id** (*str*) – Your app has a client_id after you register it on AAD.
- **client_credential** (*Union[str, _sphinx_paramlinks_msal.ClientApplication.dict]*) – For `PublicClientApplication`, you simply use *None* here. For `ConfidentialClientApplication`, it can be a string containing client secret, or an X509 certificate container in this form:

```
{
    "private_key": "...-----BEGIN PRIVATE KEY-----... in PEM format
↪",
    "thumbprint": "A1B2C3D4E5F6...",
    "public_certificate": "...-----BEGIN CERTIFICATE-----...␣
↪(Optional. See below.)",
    "passphrase": "Passphrase if the private_key is encrypted␣
↪(Optional. Added in version 1.6.0)",
}
```

MSAL Python requires a "private_key" in PEM format. If your cert is in a PKCS12 (.pfx) format, you can also convert it to PEM and get the thumbprint.

The thumbprint is available in your app's registration in Azure Portal. Alternatively, you can calculate the thumbprint.

Added in version 0.5.0: public_certificate (optional) is public key certificate which will be sent through 'x5c' JWT header only for subject name and issuer authentication to support cert auto rolls.

Per specs, “the certificate containing the public key corresponding to the key used to digitally sign the JWS MUST be the first certificate. This MAY be followed by additional certificates, with each subsequent certificate being the one used to certify the previous one.” However, your certificate’s issuer may use a different order. So, if your attempt ends up with an error AADSTS700027 - “The provided signature value did not match the expected signature value”, you may try use only the leaf cert (in PEM/str format) instead.

Added in version 1.13.0: It can also be a completely pre-signed assertion that you’ve assembled yourself. Simply pass a container containing only the key “client_assertion”, like this:

```
{
    "client_assertion": "...a JWT with claims aud, exp, iss, jti,␣
↪nbf, and sub..."
}
```

- **client_claims** (*dict*) – *Added in version 0.5.0*: It is a dictionary of extra claims that would be signed by by this *ConfidentialClientApplication* ‘s private key. For example, you can use {“client_ip”: “x.x.x.x”}. You may also override any of the following default claims:

```
{
    "aud": the_token_endpoint,
    "iss": self.client_id,
    "sub": same_as_issuer,
    "exp": now + 10_min,
    "iat": now,
    "jti": a_random_uuid
}
```

- **authority** (*str*) – A URL that identifies a token authority. It should be of the format `https://login.microsoftonline.com/your_tenant` By default, we will use `https://login.microsoftonline.com/common`

  Changed in version 1.17: you can also use predefined constant and a builder like this:

```
from msal.authority import (
    AuthorityBuilder,
    AZURE_US_GOVERNMENT, AZURE_CHINA, AZURE_PUBLIC)
my_authority = AuthorityBuilder(AZURE_PUBLIC, "contoso.onmicrosoft.
↪com")
# Now you get an equivalent of
# "https://login.microsoftonline.com/contoso.onmicrosoft.com"

# You can feed such an authority to msal's ClientApplication
from msal import PublicClientApplication
app = PublicClientApplication("my_client_id", authority=my_
↪authority, ...)
```

- **validate_authority** (*bool*) – (optional) Turns authority validation on or off. This parameter default to true.
- **cache** (TokenCache) – Sets the token cache used by this ClientApplication instance. By default, an in-memory cache will be created and used.
- **http_client** – (optional) Your implementation of abstract class HttpClient <msal.oauth2cli.http.http_client> Defaults to a requests session instance. Since MSAL

1.11.0, the default session would be configured to attempt one retry on connection error. If you are providing your own http_client, it will be your http_client's duty to decide whether to perform retry.

- **verify** – (optional) It will be passed to the verify parameter in the underlying requests library This does not apply if you have chosen to pass your own Http client
- **proxies** – (optional) It will be passed to the proxies parameter in the underlying requests library This does not apply if you have chosen to pass your own Http client
- **timeout** – (optional) It will be passed to the timeout parameter in the underlying requests library This does not apply if you have chosen to pass your own Http client
- **app_name** – (optional) You can provide your application name for Microsoft telemetry purposes. Default value is None, means it will not be passed to Microsoft.
- **app_version** – (optional) You can provide your application version for Microsoft telemetry purposes. Default value is None, means it will not be passed to Microsoft.
- **client_capabilities** (*list[str]*) – (optional) Allows configuration of one or more client capabilities, e.g. ["CP1"].

  Client capability is meant to inform the Microsoft identity platform (STS) what this client is capable for, so STS can decide to turn on certain features. For example, if client is capable to handle *claims challenge*, STS can then issue CAE access tokens to resources knowing when the resource emits *claims challenge* the client will be capable to handle.

  Implementation details: Client capability is implemented using "claims" parameter on the wire, for now. MSAL will combine them into claims parameter which you will later provide via one of the acquire-token request.
- **azure_region** (*str*) – AAD provides regional endpoints for apps to opt in to keep their traffic remain inside that region.

  As of 2021 May, regional service is only available for `acquire_token_for_client()` sent by any of the following scenarios:

  1. An app powered by a capable MSAL (MSAL Python 1.12+ will be provisioned)
  2. An app with managed identity, which is formerly known as MSI. (However MSAL Python does not support managed identity, so this one does not apply.)
  3. An app authenticated by Subject Name/Issuer (SNI).
  4. An app which already onboard to the region's allow-list.

  This parameter defaults to None, which means region behavior remains off.

  App developer can opt in to a regional endpoint, by provide its region name, such as "westus", "eastus2". You can find a full list of regions by running `az account list-locations -o table`, or referencing to this doc.

  An app running inside Azure Functions and Azure VM can use a special keyword `ClientApplication.ATTEMPT_REGION_DISCOVERY` to auto-detect region.

  Note: Setting `azure_region` to non-`None` for an app running outside of Azure Function/VM could hang indefinitely.

  You should consider opting in/out region behavior on-demand, by loading `azure_region=None` or `azure_region="westus"` or `azure_region=True` (which means opt-in and auto-detect) from your per-deployment configuration, and then do `app = ConfidentialClientApplication(..., azure_region=azure_region)`.

Alternatively, you can configure a short timeout, or provide a custom http_client which has a short timeout. That way, the latency would be under your control, but still less performant than opting out of region feature.

New in version 1.12.0.

- **`exclude_scopes`** (*`list[str]`*) – (optional) Historically MSAL hardcodes *offline_access* scope, which would allow your app to have prolonged access to user's data. If that is unnecessary or undesirable for your app, now you can use this parameter to supply an exclusion list of scopes, such as `exclude_scopes = ["offline_access"]`.
- **`http_cache`** (*`dict`*) – MSAL has long been caching tokens in the `token_cache`. Recently, MSAL also introduced a concept of `http_cache`, by automatically caching some finite amount of non-token http responses, so that *long-lived* `PublicClientApplication` and `ConfidentialClientApplication` would be more performant and responsive in some situations.

  This `http_cache` parameter accepts any dict-like object. If not provided, MSAL will use an in-memory dict.

  If your app is a command-line app (CLI), you would want to persist your http_cache across different CLI runs. The following recipe shows a way to do so:

```
# Just add the following lines at the beginning of your CLI script
import sys, atexit, pickle
http_cache_filename = sys.argv[0] + ".http_cache"
try:
    with open(http_cache_filename, "rb") as f:
        persisted_http_cache = pickle.load(f)  # Take a snapshot
except (
        FileNotFoundError,  # Or IOError in Python 2
        pickle.UnpicklingError,  # A corrupted http cache file
        ):
    persisted_http_cache = {}  # Recover by starting afresh
atexit.register(lambda: pickle.dump(
    # When exit, flush it back to the file.
    # It may occasionally overwrite another process's concurrent␣
↪write,
    # but that is fine. Subsequent runs will reach eventual␣
↪consistency.
    persisted_http_cache, open(http_cache_file, "wb")))

# And then you can implement your app as you normally would
app = msal.PublicClientApplication(
    "your_client_id",
    ...,
    http_cache=persisted_http_cache,  # Utilize persisted_http_cache
    ...,
    #token_cache=...,  # You may combine the old token_cache trick
        # Please refer to token_cache recipe at
        # https://msal-python.readthedocs.io/en/latest/#msal.
↪SerializableTokenCache
    )
app.acquire_token_interactive(["your", "scope"], ...)
```

  Content inside `http_cache` are cheap to obtain. There is no need to share them among different apps.

Content inside `http_cache` will contain no tokens nor Personally Identifiable Information (PII). Encryption is unnecessary.

New in version 1.16.0.

- **instance_discovery** (*boolean*) – Historically, MSAL would connect to a central endpoint located at `https://login.microsoftonline.com` to acquire some metadata, especially when using an unfamiliar authority. This behavior is known as Instance Discovery.

  This parameter defaults to None, which enables the Instance Discovery.

  If you know some authorities which you allow MSAL to operate with as-is, without involving any Instance Discovery, the recommended pattern is:

```
known_authorities = frozenset([  # Treat your known authorities as␣
↪const
    "https://contoso.com/adfs", "https://login.azs/foo"])
...
authority = "https://contoso.com/adfs"  # Assuming your app will␣
↪use this
app1 = PublicClientApplication(
    "client_id",
    authority=authority,
    # Conditionally disable Instance Discovery for known authorities
    instance_discovery=authority not in known_authorities,
    )
```

  If you do not know some authorities beforehand, yet still want MSAL to accept any authority that you will provide, you can use a `False` to unconditionally disable Instance Discovery.

  New in version 1.19.0.

- **allow_broker** (*boolean*) – This parameter is NOT applicable to *ConfidentialClientApplication*.

  A broker is a component installed on your device. Broker implicitly gives your device an identity. By using a broker, your device becomes a factor that can satisfy MFA (Multi-factor authentication). This factor would become mandatory if a tenant's admin enables a corresponding Conditional Access (CA) policy. The broker's presence allows Microsoft identity platform to have higher confidence that the tokens are being issued to your device, and that is more secure.

  An additional benefit of broker is, it runs as a long-lived process with your device's OS, and maintains its own cache, so that your broker-enabled apps (even a CLI) could automatically SSO from a previously established signed-in session.

  This parameter defaults to None, which means MSAL will not utilize a broker. If this parameter is set to True, MSAL will use the broker whenever possible, and automatically fall back to non-broker behavior. That also means your app does not need to enable broker conditionally, you can always set allow_broker to True, as long as your app meets the following prerequisite:

  – Installed optional dependency, e.g. `pip install msal[broker]>=1.20,<2`. (Note that broker is currently only available on Windows 10+)

  – Register a new redirect_uri for your desktop app as: `ms-appx-web://Microsoft.AAD.BrokerPlugin/your_client_id`

  - Tested your app in following scenarios:
    - Windows 10+
    - PublicClientApplication's following methods:: acquire_token_interactive(), acquire_token_by_username_password(), acquire_token_silent() (or acquire_token_silent_with_error()).
    - AAD and MSA accounts (i.e. Non-ADFS, non-B2C)

  New in version 1.20.0.

- **enable_pii_log** (*boolean*) – When enabled, logs may include PII (Personal Identifiable Information). This can be useful in troubleshooting broker behaviors. The default behavior is False.

  New in version 1.24.0.

acquire_token_by_auth_code_flow(*auth_code_flow*, *auth_response*, *scopes=None*, ***kwargs*)

Validate the auth response being redirected back, and obtain tokens.

It automatically provides nonce protection.

Parameters

- **auth_code_flow** (*dict*) – The same dict returned by *initiate_auth_code_flow()*.
- **auth_response** (*dict*) – A dict of the query string received from auth server.
- **scopes** (*list[str]*) – Scopes requested to access a protected API (a resource).

  Most of the time, you can leave it empty.

  If you requested user consent for multiple resources, here you will need to provide a subset of what you required in *initiate_auth_code_flow()*.

  OAuth2 was designed mostly for singleton services, where tokens are always meant for the same resource and the only changes are in the scopes. In AAD, tokens can be issued for multiple 3rd party resources. You can ask authorization code for multiple resources, but when you redeem it, the token is for only one intended recipient, called audience. So the developer need to specify a scope so that we can restrict the token to be issued for the corresponding audience.

Returns

- A dict containing "access_token" and/or "id_token", among others, depends on what scope was used. (See https://tools.ietf.org/html/rfc6749#section-5.1)
- A dict containing "error", optionally "error_description", "error_uri". (It is either this or that)
- Most client-side data error would result in ValueError exception. So the usage pattern could be without any protocol details:

```
def authorize():  # A controller in a web app
    try:
        result = msal_app.acquire_token_by_auth_code_flow(
            session.get("flow", {}), request.args)
        if "error" in result:
            return render_template("error.html", result)
        use(result)  # Token(s) are available in result and␣
↪cache
```

(continues on next page)

(continued from previous page)

```
    except ValueError:  # Usually caused by CSRF
        pass  # Simply ignore them
    return redirect(url_for("index"))
```

acquire_token_by_authorization_code(*code*, *scopes*, *redirect_uri=None*, *nonce=None*, *claims_challenge=None*, ***kwargs*)

The second half of the Authorization Code Grant.

Parameters

- **code** – The authorization code returned from Authorization Server.
- **scopes** (*list[str]*) – (Required) Scopes requested to access a protected API (a resource).

  If you requested user consent for multiple resources, here you will typically want to provide a subset of what you required in AuthCode.

  OAuth2 was designed mostly for singleton services, where tokens are always meant for the same resource and the only changes are in the scopes. In AAD, tokens can be issued for multiple 3rd party resources. You can ask authorization code for multiple resources, but when you redeem it, the token is for only one intended recipient, called audience. So the developer need to specify a scope so that we can restrict the token to be issued for the corresponding audience.
- **nonce** – If you provided a nonce when calling *get_authorization_request_url()*, same nonce should also be provided here, so that we'll validate it. An exception will be raised if the nonce in id token mismatches.
- **claims_challenge** – The claims_challenge parameter requests specific claims requested by the resource provider in the form of a claims_challenge directive in the www-authenticate header to be returned from the UserInfo Endpoint and/or in the ID Token and/or Access Token. It is a string of a JSON object which contains lists of claims being requested from these locations.

Returns

A dict representing the json response from AAD:

- A successful response would contain "access_token" key,
- an error response would contain "error" and usually "error_description".

acquire_token_by_refresh_token(*refresh_token*, *scopes*, ***kwargs*)

Acquire token(s) based on a refresh token (RT) obtained from elsewhere.

You use this method only when you have old RTs from elsewhere, and now you want to migrate them into MSAL. Calling this method results in new tokens automatically storing into MSAL.

You do NOT need to use this method if you are already using MSAL. MSAL maintains RT automatically inside its token cache, and an access token can be retrieved when you call *acquire_token_silent()*.

Parameters

- **refresh_token** (*str*) – The old refresh token, as a string.
- **scopes** (*list*) – The scopes associate with this old RT. Each scope needs to be in the Microsoft identity platform (v2) format. See Scopes not resources.

Returns

- A dict contains "error" and some other keys, when error happened.
- A dict contains no "error" key means migration was successful.

acquire_token_by_username_password(*username*, *password*, *scopes*, *claims_challenge=None*, ***kwargs*)

Gets a token for a given resource via user credentials.

See this page for constraints of Username Password Flow. https://github.com/AzureAD/microsoft-authentication-library-for-python/wiki/Username-Password-Authentication

Parameters

- **username** (*str*) – Typically a UPN in the form of an email address.
- **password** (*str*) – The password.
- **scopes** (*list[str]*) – Scopes requested to access a protected API (a resource).
- **claims_challenge** – The claims_challenge parameter requests specific claims requested by the resource provider in the form of a claims_challenge directive in the www-authenticate header to be returned from the UserInfo Endpoint and/or in the ID Token and/or Access Token. It is a string of a JSON object which contains lists of claims being requested from these locations.

Returns

A dict representing the json response from AAD:

- A successful response would contain "access_token" key,
- an error response would contain "error" and usually "error_description".

acquire_token_silent(*scopes*, *account*, *authority=None*, *force_refresh=False*, *claims_challenge=None*, ***kwargs*)

Acquire an access token for given account, without user interaction.

It has same parameters as the *acquire_token_silent_with_error()*. The difference is the behavior of the return value. This method will combine the cache empty and refresh error into one return value, *None*. If your app does not care about the exact token refresh error during token cache look-up, then this method is easier and recommended.

Returns

- A dict containing no "error" key, and typically contains an "access_token" key, if cache lookup succeeded.
- None when cache lookup does not yield a token.

acquire_token_silent_with_error(*scopes*, *account*, *authority=None*, *force_refresh=False*, *claims_challenge=None*, ***kwargs*)

Acquire an access token for given account, without user interaction.

It is done either by finding a valid access token from cache, or by finding a valid refresh token from cache and then automatically use it to redeem a new access token.

This method will differentiate cache empty from token refresh error. If your app cares the exact token refresh error during token cache look-up, then this method is suitable. Otherwise, the other method *acquire_token_silent()* is recommended.

Parameters

- **scopes** (*list[str]*) – (Required) Scopes requested to access a protected API (a resource).

- **account** – (Required) One of the account object returned by *get_accounts()*. Starting from MSAL Python 1.23, a `None` input will become a NO-OP and always return `None`.
- **force_refresh** – If True, it will skip Access Token look-up, and try to find a Refresh Token to obtain a new Access Token.
- **claims_challenge** – The claims_challenge parameter requests specific claims requested by the resource provider in the form of a claims_challenge directive in the www-authenticate header to be returned from the UserInfo Endpoint and/or in the ID Token and/or Access Token. It is a string of a JSON object which contains lists of claims being requested from these locations.

Returns

- A dict containing no "error" key, and typically contains an "access_token" key, if cache lookup succeeded.
- None when there is simply no token in the cache.
- A dict containing an "error" key, when token refresh failed.

`get_accounts`(*username=None*)

Get a list of accounts which previously signed in, i.e. exists in cache.

An account can later be used in *acquire_token_silent()* to find its tokens.

Parameters
username – Filter accounts with this username only. Case insensitive.

Returns
A list of account objects. Each account is a dict. For now, we only document its "username" field. Your app can choose to display those information to end user, and allow user to choose one of his/her accounts to proceed.

`get_authorization_request_url`(*scopes*, *login_hint=None*, *state=None*, *redirect_uri=None*, *response_type='code'*, *prompt=None*, *nonce=None*, *domain_hint=None*, *claims_challenge=None*, ***kwargs*)

Constructs a URL for you to start a Authorization Code Grant.

Parameters

- **scopes** (*list[str]*) – (Required) Scopes requested to access a protected API (a resource).
- **state** (*str*) – Recommended by OAuth2 for CSRF protection.
- **login_hint** (*str*) – Identifier of the user. Generally a User Principal Name (UPN).
- **redirect_uri** (*str*) – Address to return to upon receiving a response from the authority.
- **response_type** (*str*) – Default value is "code" for an OAuth2 Authorization Code grant.

  You could use other content such as "id_token" or "token", which would trigger an Implicit Grant, but that is not recommended.
- **prompt** (*str*) – By default, no prompt value will be sent, not even "none". You will have to specify a value explicitly. Its valid values are defined in Open ID Connect specs https://openid.net/specs/openid-connect-core-1_0.html#AuthRequest

- **nonce** – A cryptographically random value used to mitigate replay attacks. See also OIDC specs.
- **domain_hint** – Can be one of "consumers" or "organizations" or your tenant domain "contoso.com". If included, it will skip the email-based discovery process that user goes through on the sign-in page, leading to a slightly more streamlined user experience. More information on possible values available in Auth Code Flow doc and domain_hint doc.
- **claims_challenge** – The claims_challenge parameter requests specific claims requested by the resource provider in the form of a claims_challenge directive in the www-authenticate header to be returned from the UserInfo Endpoint and/or in the ID Token and/or Access Token. It is a string of a JSON object which contains lists of claims being requested from these locations.

Returns
The authorization url as a string.

initiate_auth_code_flow(*scopes*, *redirect_uri=None*, *state=None*, *prompt=None*, *login_hint=None*, *domain_hint=None*, *claims_challenge=None*, *max_age=None*, *response_mode=None*)

Initiate an auth code flow.

Later when the response reaches your redirect_uri, you can use *acquire_token_by_auth_code_flow()* to complete the authentication/authorization.

Parameters

- **scopes** (*list*) – It is a list of case-sensitive strings.
- **redirect_uri** (*str*) – Optional. If not specified, server will use the pre-registered one.
- **state** (*str*) – An opaque value used by the client to maintain state between the request and callback. If absent, this library will automatically generate one internally.
- **prompt** (*str*) – By default, no prompt value will be sent, not even "none". You will have to specify a value explicitly. Its valid values are defined in Open ID Connect specs https://openid.net/specs/openid-connect-core-1_0.html#AuthRequest
- **login_hint** (*str*) – Optional. Identifier of the user. Generally a User Principal Name (UPN).
- **domain_hint** – Can be one of "consumers" or "organizations" or your tenant domain "contoso.com". If included, it will skip the email-based discovery process that user goes through on the sign-in page, leading to a slightly more streamlined user experience. More information on possible values available in Auth Code Flow doc and domain_hint doc.
- **max_age** (*int*) – OPTIONAL. Maximum Authentication Age. Specifies the allowable elapsed time in seconds since the last time the End-User was actively authenticated. If the elapsed time is greater than this value, Microsoft identity platform will actively re-authenticate the End-User.

  MSAL Python will also automatically validate the auth_time in ID token.

  New in version 1.15.
- **response_mode** (*str*) – OPTIONAL. Specifies the method with which response parameters should be returned. The default value is equivalent to `query`, which is still secure enough in MSAL Python (because MSAL Python does not transfer tokens via query parameter in the first place). For even better security, we recommend

using the value `form_post`. In "form_post" mode, response parameters will be encoded as HTML form values that are transmitted via the HTTP POST method and encoded in the body using the application/x-www-form-urlencoded format. Valid values can be either "form_post" for HTTP POST to callback URI or "query" (the default) for HTTP GET with parameters encoded in query string. More information on possible values *here <https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html#ResponseModes>* and *here <https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html#FormPostResponseMode>*

Returns

The auth code flow. It is a dict in this form:

```
{
    "auth_uri": "https://...",  // Guide user to visit this
    "state": "...",  // You may choose to verify it by yourself,
                     // or just let acquire_token_by_auth_code_
↪flow()
                     // do that for you.
    "...": "...",  // Everything else are reserved and internal
}
```

The caller is expected to:

1. somehow store this content, typically inside the current session,
2. guide the end user (i.e. resource owner) to visit that auth_uri,
3. and then relay this dict and subsequent auth response to *acquire_token_by_auth_code_flow()*.

`remove_account`(*account*)

Sign me out and forget me from token cache

CHAPTER

FOUR

PUBLICCLIENTAPPLICATION

class msal.**PublicClientApplication**(*client_id*, *client_credential=None*, ***kwargs*)

__init__(*client_id*, *client_credential=None*, ***kwargs*)

Same as `ClientApplication.__init__()`, except that `client_credential` parameter shall remain `None`.

acquire_token_by_device_flow(*flow*, *claims_challenge=None*, ***kwargs*)

Obtain token by a device flow object, with customizable polling effect.

Parameters

- **flow** (*dict*) – A dict previously generated by `initiate_device_flow()`. By default, this method's polling effect will block current thread. You can abort the polling loop at any time, by changing the value of the flow's "expires_at" key to 0.
- **claims_challenge** – The claims_challenge parameter requests specific claims requested by the resource provider in the form of a claims_challenge directive in the www-authenticate header to be returned from the UserInfo Endpoint and/or in the ID Token and/or Access Token. It is a string of a JSON object which contains lists of claims being requested from these locations.

Returns

A dict representing the json response from AAD:

- A successful response would contain "access_token" key,
- an error response would contain "error" and usually "error_description".

acquire_token_interactive(*scopes*, *prompt=None*, *login_hint=None*, *domain_hint=None*, *claims_challenge=None*, *timeout=None*, *port=None*, *extra_scopes_to_consent=None*, *max_age=None*, *parent_window_handle=None*, *on_before_launching_ui=None*, ***kwargs*)

Acquire token interactively i.e. via a local browser.

Prerequisite: In Azure Portal, configure the Redirect URI of your "Mobile and Desktop application" as `http://localhost`. If you opts in to use broker during `PublicClientApplication` creation, your app also need this Redirect URI: `ms-appx-web://Microsoft.AAD.BrokerPlugin/YOUR_CLIENT_ID`

Parameters

- **scopes** (*list*) – It is a list of case-sensitive strings.
- **prompt** (*str*) – By default, no prompt value will be sent, not even "none". You will have to specify a value explicitly. Its valid values are defined in Open ID Connect specs https://openid.net/specs/openid-connect-core-1_0.html#AuthRequest

- **login_hint** (*str*) – Optional. Identifier of the user. Generally a User Principal Name (UPN).
- **domain_hint** – Can be one of "consumers" or "organizations" or your tenant domain "contoso.com". If included, it will skip the email-based discovery process that user goes through on the sign-in page, leading to a slightly more streamlined user experience. More information on possible values available in Auth Code Flow doc and domain_hint doc.
- **claims_challenge** – The claims_challenge parameter requests specific claims requested by the resource provider in the form of a claims_challenge directive in the www-authenticate header to be returned from the UserInfo Endpoint and/or in the ID Token and/or Access Token. It is a string of a JSON object which contains lists of claims being requested from these locations.
- **timeout** (*int*) – This method will block the current thread. This parameter specifies the timeout value in seconds. Default value `None` means wait indefinitely.
- **port** (*int*) – The port to be used to listen to an incoming auth response. By default we will use a system-allocated port. (The rest of the redirect_uri is hard coded as `http://localhost`.)
- **extra_scopes_to_consent** (*list*) – "Extra scopes to consent" is a concept only available in AAD. It refers to other resources you might want to prompt to consent for, in the same interaction, but for which you won't get back a token for in this particular operation.
- **max_age** (*int*) – OPTIONAL. Maximum Authentication Age. Specifies the allowable elapsed time in seconds since the last time the End-User was actively authenticated. If the elapsed time is greater than this value, Microsoft identity platform will actively re-authenticate the End-User.

  MSAL Python will also automatically validate the auth_time in ID token.

  New in version 1.15.
- **parent_window_handle** (*int*) – OPTIONAL. If your app is a GUI app running on modern Windows system, and your app opts in to use broker, you are recommended to also provide its window handle, so that the sign in UI window will properly pop up on top of your window.

  New in version 1.20.0.
- **on_before_launching_ui** (*function*) – A callback with the form of `lambda ui="xyz", **kwargs: print("A {} will be launched".format(ui))`, where `ui` will be either "browser" or "broker". You can use it to inform your end user to expect a pop-up window.

  New in version 1.20.0.

Returns

- A dict containing no "error" key, and typically contains an "access_token" key.
- A dict containing an "error" key, when token refresh failed.

initiate_device_flow(*scopes=None*, ***kwargs*)

Initiate a Device Flow instance, which will be used in *acquire_token_by_device_flow()*.

Parameters

scopes (*list[str]*) – Scopes requested to access a protected API (a resource).

Returns

A dict representing a newly created Device Flow object.

- A successful response would contain "user_code" key, among others
- an error response would contain some other readable key/value pairs.

CHAPTER

FIVE

CONFIDENTIALCLIENTAPPLICATION

class msal.**ConfidentialClientApplication**(*client_id*, *client_credential=None*, *authority=None*, *validate_authority=True*, *token_cache=None*, *http_client=None*, *verify=True*, *proxies=None*, *timeout=None*, *client_claims=None*, *app_name=None*, *app_version=None*, *client_capabilities=None*, *azure_region=None*, *exclude_scopes=None*, *http_cache=None*, *instance_discovery=None*, *allow_broker=None*, *enable_pii_log=None*)

Same as *ClientApplication.__init__()*, except that `allow_broker` parameter shall remain `None`.

acquire_token_for_client(*scopes*, *claims_challenge=None*, ***kwargs*)

Acquires token for the current confidential client, not for an end user.

Since MSAL Python 1.23, it will automatically look for token from cache, and only send request to Identity Provider when cache misses.

Parameters

- **scopes** (*list[str]*) – (Required) Scopes requested to access a protected API (a resource).
- **claims_challenge** – The claims_challenge parameter requests specific claims requested by the resource provider in the form of a claims_challenge directive in the www-authenticate header to be returned from the UserInfo Endpoint and/or in the ID Token and/or Access Token. It is a string of a JSON object which contains lists of claims being requested from these locations.

Returns

A dict representing the json response from AAD:

- A successful response would contain "access_token" key,
- an error response would contain "error" and usually "error_description".

acquire_token_on_behalf_of(*user_assertion*, *scopes*, *claims_challenge=None*, ***kwargs*)

Acquires token using on-behalf-of (OBO) flow.

The current app is a middle-tier service which was called with a token representing an end user. The current app can use such token (a.k.a. a user assertion) to request another token to access downstream web API, on behalf of that user. See detail docs here .

The current middle-tier app has no user interaction to obtain consent. See how to gain consent upfront for your middle-tier app from this article. https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-on-behalf-of-flow#gaining-consent-for-the-middle-tier-application

Parameters

- `user_assertion` (*str*) – The incoming token already received by this app
- `scopes` (*list[str]*) – Scopes required by downstream API (a resource).
- `claims_challenge` – The claims_challenge parameter requests specific claims requested by the resource provider in the form of a claims_challenge directive in the www-authenticate header to be returned from the UserInfo Endpoint and/or in the ID Token and/or Access Token. It is a string of a JSON object which contains lists of claims being requested from these locations.

Returns

A dict representing the json response from AAD:

- A successful response would contain "access_token" key,
- an error response would contain "error" and usually "error_description".

CHAPTER

SIX

TOKENCACHE

One of the parameters accepted by both *PublicClientApplication* and *ConfidentialClientApplication* is the *TokenCache*.

class msal.**TokenCache**

This is considered as a base class containing minimal cache behavior.

Although it maintains tokens using unified schema across all MSAL libraries, this class does not serialize/persist them. See subclass *SerializableTokenCache* for details on serialization.

add(*event*, *now=None*)

Handle a token obtaining event, and add tokens into cache.

You can subclass it to add new behavior, such as, token serialization. See *SerializableTokenCache* for example.

class msal.**SerializableTokenCache**

This serialization can be a starting point to implement your own persistence.

This class does NOT actually persist the cache on disk/db/etc.. Depending on your need, the following simple recipe for file-based persistence may be sufficient:

```
import os, atexit, msal
cache = msal.SerializableTokenCache()
if os.path.exists("my_cache.bin"):
    cache.deserialize(open("my_cache.bin", "r").read())
atexit.register(lambda:
    open("my_cache.bin", "w").write(cache.serialize())
    # Hint: The following optional line persists only when state changed
    if cache.has_state_changed else None
    )
app = msal.ClientApplication(..., token_cache=cache)
...
```

Variables

has_state_changed (*bool*) – Indicates whether the cache state in the memory has changed since last *serialize()* or *deserialize()* call.

add(*event*, ***kwargs*)

Handle a token obtaining event, and add tokens into cache.

deserialize(*state*)

Deserialize the cache from a state previously obtained by serialize()

serialize()

Serialize the current cache state into a string.

INDEX