Package ‘AzureAuth’

December 20, 2025

Title Authentication Services for Azure Active Directory

Version 1.3.4

Description Provides Azure Active Directory (AAD) authentication functionality for R users of Microsoft's 'Azure' cloud <https://azure.microsoft.com/en-us>. Use this package to obtain 'OAuth' 2.0 tokens for services including Azure Resource Manager, Azure Storage and others. It supports both AAD v1.0 and v2.0, as well as multiple authentication methods, including device code and resource owner grant. Tokens are cached in a user-specific directory obtained using the 'rappdirs' package. The interface is based on the 'OAuth' framework in the 'httr' package, but customised and streamlined for Azure. Part of the 'AzureR' family of packages.

URL https://github.com/Azure/AzureAuth https://github.com/Azure/AzureR

BugReports https://github.com/Azure/AzureAuth/issues

License MIT + file LICENSE

VignetteBuilder knitr

Depends R (>= 3.3)

Imports utils, httr (>= 1.3), openssl, jsonlite, jose, R6, rappdirs

Suggests knitr, rmarkdown, testthat, httpuv, shiny, shinyjs, AzureRMR, AzureGraph

RoxygenNote 7.3.2

NeedsCompilation no

Author Hong Ooi [aut, cre],
Tyler Littlefield [ctb],
httr development team [ctb] (Original OAuth listener code),
Scott Holden [ctb] (Advice on AAD authentication),
Chris Stone [ctb] (Advice on AAD authentication),
Microsoft [cph]

Maintainer Hong Ooi <hongooi73@gmail.com>

Repository CRAN

Date/Publication 2025-12-20 12:10:02 UTC

Contents

`AzureManualToken` *Manual Azure Token*

Description

Initialize a manual token from a raw access token string.

Refresh the token. Manual tokens cannot be refreshed.

Check if this token can be refreshed.

Cache the token. Manual tokens are not cached.

Compute a hash for this token.

Print the token object.

Arguments

`token`	A character string containing the access token.
`type`	The token type, usually "Bearer".
`tenant`	Optional tenant ID. If NULL, extracted from JWT claims.
`resource`	Optional resource/audience. If NULL, extracted from JWT claims.

Format

An R6 object of class `AzureManualToken`, inheriting from `AzureToken`.

Details

Create an Azure token object from a pre-existing access token string. This is useful when you have obtained a token externally (e.g., via Azure CLI, Python, or another authentication mechanism) and want to use it with the AzureR ecosystem.

The `AzureManualToken` class provides a way to wrap an externally-obtained access token string so it can be used with packages like `AzureGraph`, `AzureRMR`, and other AzureR family packages that expect an `AzureToken` object.

If the token is a JWT (JSON Web Token), the class will attempt to parse it to extract metadata such as the tenant ID, audience (resource), and expiration time. If parsing fails (e.g., for opaque tokens), the class will still function but with limited metadata.

Since manual tokens are managed externally, the `refresh()` method cannot obtain a new token. When the token expires, you must create a new `AzureManualToken` object with a fresh token string.

Value

Returns self invisibly.

Always returns FALSE for manual tokens.

Returns NULL invisibly.

An MD5 hash string based on the token content.

Methods

- `new(token, type, tenant, resource)`: Initialize a new manual token object.
- `refresh()`: Cannot refresh a manual token; issues a warning and returns self.
- `validate()`: Checks if the token has expired based on JWT claims.
- `can_refresh()`: Returns FALSE since manual tokens cannot be refreshed.
- `cache()`: No-op; manual tokens are not cached.

See Also

`get_manual_token`, `AzureToken`, `decode_jwt`

Examples

```
## Not run:
# Get a token externally (e.g., from Azure CLI)
# az account get-access-token --resource https://graph.microsoft.com
raw_token <- "eyJ0eXAiOiJKV1QiLC..."

# Create a manual token object
token <- get_manual_token(raw_token)

# Check if metadata was parsed
print(token$tenant)
print(token$resource)

# Use with AzureGraph
library(AzureGraph)
gr <- ms_graph$new(token = token)
me <- gr$get_user("me")

# Check token validity
token$validate()

## End(Not run)
```

`AzureR_dir` *Data directory for AzureR packages*

Description

Data directory for AzureR packages

Usage

```
AzureR_dir()

create_AzureR_dir()
```

Details

AzureAuth can save your authentication credentials in a user-specific directory, using the rappdirs package. On recent Windows versions, this will usually be in the location `C:\\Users\\(username)\\AppData\\Local\\Az`
On Unix/Linux, it will be in `~/.local/share/AzureR`, and on MacOS, it will be in `~/Library/Application Support/Azur`
you can specify the location of the directory in the environment variable `R_AZURE_DATA_DIR`. AzureAuth does not modify R's working directory, which significantly lessens the risk of accidentally introducing cached tokens into source control.

On package startup, if this directory does not exist, AzureAuth will prompt you for permission to create it. It's recommended that you allow the directory to be created, as otherwise you will have to reauthenticate with Azure every time. Note that many cloud engineering tools, including the Azure CLI, save authentication credentials in this way. The prompt only appears in an interactive session (in the sense that `interactive()` returns TRUE); if AzureAuth is loaded in a batch script, the directory is not created if it doesn't already exist.

`create_AzureR_dir` is a utility function to create the caching directory manually. This can be useful not just for non-interactive sessions, but also Jupyter and R notebooks, which are not *technically* interactive in that `interactive()` returns FALSE.

The caching directory is also used by other AzureR packages, notably AzureRMR (for storing Resource Manager logins) and AzureGraph (for Microsoft Graph logins). You should not save your own files in it; instead, treat it as something internal to the AzureR packages.

Value

A string containing the data directory.

See Also

get_azure_token

rappdirs::user_data_dir

AzureToken — *Azure OAuth authentication*

Description

Azure OAuth 2.0 token classes, with an interface based on the Token2.0 class in httr. Rather than calling the initialization methods directly, tokens should be created via `get_azure_token()`.

Format

An R6 object representing an Azure Active Directory token and its associated credentials. `AzureToken` is the base class, and the others inherit from it.

Methods

- `refresh`: Refreshes the token. For expired tokens without an associated refresh token, refreshing really means requesting a new token.
- `validate`: Checks if the token has not yet expired. Note that a token may be invalid for reasons other than having expired, eg if it is revoked on the server.
- `hash`: Computes an MD5 hash on the input fields of the object. Used internally for identification purposes when caching.
- `cache`: Stores the token on disk for use in future sessions.

See Also

get_azure_token, httr::Token

build_authorization_uri — *Standalone OAuth authorization functions*

Description

Standalone OAuth authorization functions

Usage

```
build_authorization_uri(
  resource,
  tenant,
  app,
  username = NULL,
  ...,
  aad_host = "https://login.microsoftonline.com/",
  version = 1
)

get_device_creds(
  resource,
  tenant,
  app,
  aad_host = "https://login.microsoftonline.com/",
  version = 1
)
```

Arguments

resource, tenant, app, aad_host, version	See the corresponding arguments for get_azure_token.
username	For build_authorization_uri, an optional login hint to be sent to the authorization endpoint.
...	Named arguments that will be added to the authorization URI as query parameters.

Details

These functions are mainly for use in embedded scenarios, such as within a Shiny web app. In this case, the interactive authentication flows (authorization code and device code) need to be split up so that the authorization step is handled separately from the token acquisition step. You should not need to use these functions inside a regular R session, or when executing an R batch script.

Value

For build_authorization_uri, the authorization URI as a string. This can be set as a redirect from within a Shiny app's UI component.

For get_device_creds, a list containing the following components:

- user_code: A short string to be shown to the user
- device_code: A long string to verify the session with the AAD server
- verification_uri: The URI the user should browse to in order to login
- expires_in: The duration in seconds for which the user and device codes are valid
- interval: The interval between polling requests to the AAD token endpoint
- message: A string with login instructions for the user

Examples

```
build_authorization_uri("https://myresource", "mytenant", "app_id",
                        redirect_uri="http://localhost:8100")

## Not run:

## obtaining an authorization code separately to acquiring the token
# first, get the authorization URI
```

```
auth_uri <- build_authorization_uri("https://management.azure.com/", "mytenant", "app_id")
# browsing to the URI will log you in and redirect to another URI containing the auth code
browseURL(auth_uri)
# use the code to acquire the token
get_azure_token("https://management.azure.com/", "mytenant", "app_id",
    auth_code="code-from-redirect")


## obtaining device credentials separately to acquiring the token
# first, contact the authorization endpoint to get the user and device codes
creds <- get_device_creds("https://management.azure.com/", "mytenant", "app_id")
# print the login instructions
creds$message
# use the creds to acquire the token
get_azure_token("https://management.azure.com/", "mytenant", "app_id",
    auth_type="device_code", device_creds=creds)


## End(Not run)
```

`cert_assertion` *Create a client assertion for certificate authentication*

Description

Create a client assertion for certificate authentication

Usage

```
cert_assertion(certificate, duration = 3600, signature_size = 256, ...)
```

Arguments

`certificate`	An Azure Key Vault certificate object, or the name of a PEM or PFX file containing *both* a private key and a public certificate.
`duration`	The requested validity period of the token, in seconds. The default is 1 hour.
`signature_size`	The size of the SHA2 signature.
`...`	Other named arguments which will be treated as custom claims.

Details

Use this function to customise a client assertion for authenticating with a certificate.

Value

An object of S3 class `cert_assertion`, which is a list representing the assertion.

See Also

get_azure_token

Examples

```
## Not run:

cert_assertion("mycert.pem", duration=2*3600)
cert_assertion("mycert.pem", custom_data="some text")

# using a cert stored in Azure Key Vault
cert <- AzureKeyVault::key_vault("myvault")$certificates$get("mycert")
cert_assertion(cert, duration=2*3600)


## End(Not run)
```

`decode_jwt` *Get raw access token (which is a JWT object)*

Description

Get raw access token (which is a JWT object)

Usage

```
decode_jwt(token, ...)

## S3 method for class 'AzureToken'
decode_jwt(token, type = c("access", "id"), ...)

## S3 method for class 'Token'
decode_jwt(token, type = c("access", "id"), ...)

## S3 method for class 'character'
decode_jwt(token, ...)

extract_jwt(token, ...)

## S3 method for class 'AzureToken'
extract_jwt(token, type = c("access", "id"), ...)

## S3 method for class 'Token'
extract_jwt(token, type = c("access", "id"), ...)

## S3 method for class 'character'
extract_jwt(token, ...)
```

Arguments

`token`	A token object. This can be an object of class `AzureToken`, of class `httr::Token`, or a character string containing the encoded token.
`...`	Other arguments passed to methods.
`type`	For the `AzureToken` and `httr::Token` methods, the token to decode/retrieve: either the access token or ID token.

Details

An OAuth token is a *JSON Web Token*, which is a set of base64URL-encoded JSON objects containing the token credentials along with an optional (opaque) verification signature. `decode_jwt` decodes the credentials into an R object so they can be viewed. `extract_jwt` extracts the credentials from an R object of class `AzureToken` or `httr::Token`.

Note that `decode_jwt` does not touch the token signature or attempt to verify the credentials. You should not rely on the decoded information without verifying it independently. Passing the token itself to Azure is safe, as Azure will carry out its own verification procedure.

Value

For `extract_jwt`, the character string containing the encoded token, suitable for including in a HTTP query. For `decode_jwt`, a list containing up to 3 components: `header`, `payload` and `signature`.

See Also

jwt.io, the main JWT informational site

jwt.ms, Microsoft site to decode and explain JWTs

JWT Wikipedia entry

`format_auth_header` *Format an AzureToken object*

Description

Format an AzureToken object

Usage

```
format_auth_header(token)
```

Arguments

`token`	An Azure OAuth token.

get_managed_token *Manage Azure Active Directory OAuth 2.0 tokens*

Description

Use these functions to authenticate with Azure Active Directory (AAD).

Usage

```
get_managed_token(resource, token_args = list(), use_cache = NULL)

get_azure_token(
  resource,
  tenant,
  app,
  password = NULL,
  username = NULL,
  certificate = NULL,
  auth_type = NULL,
  aad_host = "https://login.microsoftonline.com/",
  version = 1,
  authorize_args = list(),
  token_args = list(),
  use_cache = NULL,
  on_behalf_of = NULL,
  auth_code = NULL,
  device_creds = NULL
)

delete_azure_token(
  resource,
  tenant,
  app,
  password = NULL,
  username = NULL,
  certificate = NULL,
  auth_type = NULL,
  aad_host = "https://login.microsoftonline.com/",
  version = 1,
  authorize_args = list(),
  token_args = list(),
  on_behalf_of = NULL,
  hash = NULL,
  confirm = TRUE
)

load_azure_token(hash)
```

```
clean_token_directory(confirm = TRUE)

list_azure_tokens()

token_hash(
  resource,
  tenant,
  app,
  password = NULL,
  username = NULL,
  certificate = NULL,
  auth_type = NULL,
  aad_host = "https://login.microsoftonline.com/",
  version = 1,
  authorize_args = list(),
  token_args = list(),
  on_behalf_of = NULL
)

is_azure_token(object)

is_azure_v1_token(object)

is_azure_v2_token(object)
```

Arguments

`resource`	For AAD v1.0, the URL of your resource host, or a GUID. For AAD v2.0, a character vector of scopes, each consisting of a URL or GUID along with a path designating the access scope. See 'Details' below.
`token_args`	An optional list of further parameters for the token endpoint. These will be included in the body of the request for `get_azure_token`, or as URI query parameters for `get_managed_token`.
`use_cache`	If TRUE and cached credentials exist, use them instead of obtaining a new token. The default value of NULL means to use the cache only if AzureAuth is not running inside a Shiny app.
`tenant`	Your tenant. This can be a name ("myaadtenant"), a fully qualified domain name ("myaadtenant.onmicrosoft.com" or "mycompanyname.com"), or a GUID. It can also be one of the generic tenants "common", "organizations" or "consumers"; see 'Generic tenants' below.
`app`	The client/app ID to use to authenticate with.
`password`	For most authentication flows, this is the password for the *app* where needed, also known as the client secret. For the resource owner grant, this is your personal account password. See 'Details' below.
`username`	Your AAD username, if using the resource owner grant. See 'Details' below.

certificate	A file containing the certificate for authenticating with (including the private key), an Azure Key Vault certificate object, or a call to the `cert_assertion` function to build a client assertion with a certificate. See 'Certificate authentication' below.
auth_type	The authentication type. See 'Details' below.
aad_host	URL for your AAD host. For the public Azure cloud, this is `https://login.microsoftonline.com/`. Change this if you are using a government or private cloud. Can also be a full URL, eg `https://mydomain.b2clogin.com/mydomain/other/path/names/oauth2` (this is relevant mainly for Azure B2C logins).
version	The AAD version, either 1 or 2. Authenticating with a personal account as opposed to a work or school account requires AAD 2.0. The default is AAD 1.0 for compatibility reasons, but you should use AAD 2.0 if possible.
authorize_args	An optional list of further parameters for the AAD authorization endpoint. These will be included in the request URI as query parameters. Only used if `auth_type="authorization_code`
on_behalf_of	For the on-behalf-of authentication type, a token. This should be either an AzureToken object, or a string containing the JWT-encoded token itself.
auth_code	For the `authorization_code` flow, the code. Only used if `auth_type == "authorization_code"`.
device_creds	For the `device_code` flow, the device credentials used to verify the session between the client and the server. Only used if `auth_type == "device_code"`.
hash	The MD5 hash of this token, computed from the above inputs. Used by `load_azure_token` and `delete_azure_token` to identify a cached token to load and delete, respectively.
confirm	For `delete_azure_token`, whether to prompt for confirmation before deleting a token.
object	For `is_azure_token`, `is_azure_v1_token` and `is_azure_v2_token`, an R object.

Details

`get_azure_token` does much the same thing as `httr::oauth2.0_token()`, but customised for Azure. It obtains an OAuth token, first by checking if a cached value exists on disk, and if not, acquiring it from the AAD server. `load_azure_token` loads a token given its hash, `delete_azure_token` deletes a cached token given either the credentials or the hash, and `list_azure_tokens` lists currently cached tokens.

`get_managed_token` is a specialised function to acquire tokens for a *managed identity*. This is an Azure service, such as a VM or container, that has been assigned its own identity and can be granted access permissions like a regular user. The advantage of managed identities over the other authentication methods (see below) is that you don't have to store a secret password, which improves security. Note that `get_managed_token` can only be used from within the managed identity itself.

By default `get_managed_token` retrieves a token using the system-assigned identity for the resource. To obtain a token with a user-assigned identity, pass either the client, object or Azure resource ID in the `token_args` argument. See the examples below.

The `resource` arg should be a single URL or GUID for AAD v1.0. For AAD v2.0, it should be a vector of *scopes*, where each scope consists of a URL or GUID along with a path that designates the type of access requested. If a v2.0 scope doesn't have a path, `get_azure_token` will append the `/.default` path with a warning. A special scope is `offline_access`, which requests a refresh token from AAD along with the access token: without this scope, you will have to reauthenticate if you want to refresh the token.

The `auth_code` and `device_creds` arguments are intended for use in embedded scenarios, eg when AzureAuth is loaded from within a Shiny web app. They enable the flow authorization step to be separated from the token acquisition step, which is necessary within an app; you can generally ignore these arguments when using AzureAuth interactively or as part of an R script. See the help for build_authorization_uri for examples on their use.

`token_hash` computes the MD5 hash of its arguments. This is used by AzureAuth to identify tokens for caching purposes. Note that tokens are only cached if you allowed AzureAuth to create a data directory at package startup.

One particular use of the `authorize_args` argument is to specify a different redirect URI to the default; see the examples below.

Authentication methods

1. Using the **authorization_code** method is a multi-step process. First, `get_azure_token` opens a login window in your browser, where you can enter your AAD credentials. In the background, it loads the httpuv package to listen on a local port. Once you have logged in, the AAD server redirects your browser to a local URL that contains an authorization code. `get_azure_token` retrieves this authorization code and sends it to the AAD access endpoint, which returns the OAuth token.
2. The **device_code** method is similar in concept to authorization_code, but is meant for situations where you are unable to browse the Internet – for example if you don't have a browser installed or your computer has input constraints. First, `get_azure_token` contacts the AAD devicecode endpoint, which responds with a login URL and an access code. You then visit the URL and enter the code, possibly using a different computer. Meanwhile, `get_azure_token` polls the AAD access endpoint for a token, which is provided once you have entered the code.
3. The **client_credentials** method is much simpler than the above methods, requiring only one step. `get_azure_token` contacts the access endpoint, passing it either the app secret or the certificate assertion (which you supply in the `password` or `certificate` argument respectively). Once the credentials are verified, the endpoint returns the token. This is the method typically used by service accounts.
4. The **resource_owner** method also requires only one step. In this method, `get_azure_token` passes your (personal) username and password to the AAD access endpoint, which validates your credentials and returns the token.
5. The **on_behalf_of** method is used to authenticate with an Azure resource by passing a token obtained beforehand. It is mostly used by intermediate apps to authenticate for users. In particular, you can use this method to obtain tokens for multiple resources, while only requiring the user to authenticate once: see the examples below.

If the authentication method is not specified, it is chosen based on the presence or absence of the other arguments, and whether httpuv is installed.

The httpuv package must be installed to use the authorization_code method, as this requires a web server to listen on the (local) redirect URI. See httr::oauth2.0_token for more information; note that Azure does not support the `use_oob` feature of the httr OAuth 2.0 token class.

Similarly, since the authorization_code method opens a browser to load the AAD authorization page, your machine must have an Internet browser installed that can be run from inside R. In particular, if you are using a Linux Data Science Virtual Machine in Azure, you may run into difficulties; use one of the other methods instead.

Certificate authentication

OAuth tokens can be authenticated via an SSL/TLS certificate, which is considered more secure than a client secret. To do this, use the `certificate` argument, which can contain any of the following:

- The name of a PEM or PFX file, containing *both* the private key and the public certificate.
- A certificate object from the AzureKeyVault package, representing a cert stored in the Key Vault service.
- A call to the `cert_assertion()` function to customise details of the requested token, eg the duration, expiry date, custom claims, etc. See the examples below.

Generic tenants

There are 3 generic values that can be used as tenants when authenticating:

Tenant	Description
`common`	Allows users with both personal Microsoft accounts and work/school accounts from Azure AD to sign into
`organizations`	Allows only users with work/school accounts from Azure AD to sign into the application.
`consumers`	Allows only users with personal Microsoft accounts (MSA) to sign into the application.

Authentication vs authorization

Azure Active Directory can be used for two purposes: *authentication* (verifying that a user is who they claim they are) and *authorization* (granting a user permission to access a resource). In AAD, a successful authorization process concludes with the granting of an OAuth 2.0 access token, as discussed above. Authentication uses the same process but concludes by granting an ID token, as defined in the OpenID Connect protocol.

`get_azure_token` can be used to obtain ID tokens along with regular OAuth access tokens, when using an interactive flow (authorization_code or device_code). The behaviour depends on the AAD version:

When retrieving ID tokens, the behaviour depends on the AAD version:

- AAD v1.0 will return an ID token as well as the access token by default; you don't have to do anything extra. However, AAD v1.0 will not *refresh* the ID token when it expires; you must reauthenticate to get a new one. To ensure you don't pull the cached version of the credentials, specify `use_cache=FALSE` in the calls to `get_azure_token`.
- Unlike AAD v1.0, AAD v2.0 does not return an ID token by default. To get a token, include `openid` as a scope. On the other hand it *does* refresh the ID token, so bypassing the cache is not needed. It's recommended to use AAD v2.0 if you only want an ID token.

If you *only* want to do authentication and not authorization (for example if your app does not use any Azure resources), specify the `resource` argument as follows:

- For AAD v1.0, use a blank resource (`resource=""`).
- For AAD v2.0, use `resource="openid"` without any other elements. Optionally you can add `"offline_access"` as a 2nd element if you want a refresh token as well.

See also the examples below.

Caching

AzureAuth caches tokens based on all the inputs to `get_azure_token` as listed above. Tokens are cached in a custom, user-specific directory, created with the rappdirs package. On recent Windows versions, this will usually be in the location `C:\\Users\\(username)\\AppData\\Local\\AzureR`. On Linux, it will be in `~/.config/AzureR`, and on MacOS, it will be in `~/Library/Application Support/AzureR`. Alternatively, you can specify the location of the directory in the environment variable `R_AZURE_DATA_DIR`. Note that a single directory is used for all tokens, and the working directory is not touched (which significantly lessens the risk of accidentally introducing cached tokens into source control).

To list all cached tokens on disk, use `list_azure_tokens`. This returns a list of token objects, named according to their MD5 hashes.

To delete a cached token, use `delete_azure_token`. This takes the same inputs as `get_azure_token`, or you can specify the MD5 hash directly in the `hash` argument.

To delete all files in the caching directory, use `clean_token_directory`.

Refreshing

A token object can be refreshed by calling its `refresh()` method. If the token's credentials contain a refresh token, this is used; otherwise a new access token is obtained by reauthenticating.

Note that in AAD, a refresh token can be used to obtain an access token for any resource or scope that you have permissions for. Thus, for example, you could use a refresh token issued on a request for Azure Resource Manager (`https://management.azure.com/`) to obtain a new access token for Microsoft Graph (`https://graph.microsoft.com/`).

To obtain an access token for a new resource, change the object's `resource` (for an AAD v1.0 token) or `scope` field (for an AAD v2.0 token) before calling `refresh()`. If you *also* want to retain the token for the old resource, you should call the `clone()` method first to create a copy. See the examples below.

Value

For `get_azure_token`, an object inheriting from `AzureToken`. The specific class depends on the authentication flow: `AzureTokenAuthCode`, `AzureTokenDeviceCode`, `AzureTokenClientCreds`, `AzureTokenOnBehalfOf`, `AzureTokenResOwner`. For `get_managed_token`, a similar object of class `AzureTokenManaged`.

For `list_azure_tokens`, a list of such objects retrieved from disk.

The actual credentials that are returned from the authorization endpoint can be found in the `credentials` field, the same as with a `httr::Token` object. The access token (if present) will be `credentials$access_token`, and the ID token (if present) will be `credentials$id_token`. Use these if you are manually constructing a HTTP request and need to insert an "Authorization" header, for example.

See Also

AzureToken, httr::oauth2.0_token, httr::Token, cert_assertion, build_authorization_uri, get_device_creds

Azure Active Directory for developers, Managed identities overview Device code flow on OAuth.com, OAuth 2.0 RFC for the gory details on how OAuth works

Examples

```
## Not run:

# authenticate with Azure Resource Manager:
# no user credentials are supplied, so this will use the authorization_code
# method if httpuv is installed, and device_code if not
get_azure_token("https://management.azure.com/", tenant="mytenant", app="app_id")

# you can force a specific authentication method with the auth_type argument
get_azure_token("https://management.azure.com/", tenant="mytenant", app="app_id",
    auth_type="device_code")

# to default to the client_credentials method, supply the app secret as the password
get_azure_token("https://management.azure.com/", tenant="mytenant", app="app_id",
    password="app_secret")

# authenticate to your resource with the resource_owner method: provide your username and password
get_azure_token("https://myresource/", tenant="mytenant", app="app_id",
    username="user", password="abcdefg")

# obtaining multiple tokens: authenticate (interactively) once...
tok0 <- get_azure_token("serviceapp_id", tenant="mytenant", app="clientapp_id",
    auth_type="authorization_code")
# ...then get tokens for each resource (Resource Manager and MS Graph) with on_behalf_of
tok1 <- get_azure_token("https://management.azure.com/", tenant="mytenant", app="serviceapp_id",
    password="serviceapp_secret", on_behalf_of=tok0)
tok2 <- get_azure_token("https://graph.microsoft.com/", tenant="mytenant", app="serviceapp_id",
    password="serviceapp_secret", on_behalf_of=tok0)


# authorization_code flow with app registered in AAD as a web rather than a native client:
# supply the client secret in the password arg
get_azure_token("https://management.azure.com/", "mytenant", "app_id",
    password="app_secret", auth_type="authorization_code")


# use a different redirect URI to the default localhost:1410
get_azure_token("https://management.azure.com/", tenant="mytenant", app="app_id",
    authorize_args=list(redirect_uri="http://localhost:8000"))


# request an AAD v1.0 token for Resource Manager (the default)
token1 <- get_azure_token("https://management.azure.com/", "mytenant", "app_id")

# same request to AAD v2.0, along with a refresh token
token2 <- get_azure_token(c("https://management.azure.com/.default", "offline_access"),
    "mytenant", "app_id", version=2)

# requesting multiple scopes (Microsoft Graph) with AAD 2.0
get_azure_token(c("https://graph.microsoft.com/User.Read.All",
                  "https://graph.microsoft.com/User.ReadWrite.All",
                  "https://graph.microsoft.com/Directory.ReadWrite.All",
                  "offline_access"),
    "mytenant", "app_id", version=2)


# list saved tokens
list_azure_tokens()

# delete a saved token from disk
delete_azure_token(resource="https://myresource/", tenant="mytenant", app="app_id",
    username="user", password="abcdefg")

# delete a saved token by specifying its MD5 hash
delete_azure_token(hash="7ea491716e5b10a77a673106f3f53bfd")


# authenticating for B2C logins (custom AAD host)
get_azure_token("https://mydomain.com", "mytenant", "app_id", "password",
    aad_host="https://mytenant.b2clogin.com/tfp/mytenant.onmicrosoft.com/custom/oauth2")


# authenticating with a certificate
get_azure_token("https://management.azure.com/", "mytenant", "app_id",
    certificate="mycert.pem")

# authenticating with a certificate stored in Azure Key Vault
cert <- AzureKeyVault::key_vault("myvault")$certificates$get("mycert")
get_azure_token("https://management.azure.com/", "mytenant", "app_id",
    certificate=cert)

# get a token valid for 2 hours (default is 1 hour)
get_azure_token("https://management.azure.com/", "mytenant", "app_id",
    certificate=cert_assertion("mycert.pem", duration=2*3600))


# ID token with AAD v1.0
# if you only want an ID token, set the resource to blank ("")
tok <- get_azure_token("", "mytenant", "app_id", use_cache=FALSE)
extract_jwt(tok, "id")

# ID token with AAD v2.0 (recommended)
tok2 <- get_azure_token(c("openid", "offline_access"), "mytenant", "app_id", version=2)
extract_jwt(tok2, "id")


# get a token from within a managed identity (VM, container or service)
get_managed_token("https://management.azure.com/")
```

```
# get a token from a managed identity, with a user-defined identity:
# specify one of the identity's object_id, client_id and mi_res_id (Azure resource ID)
# you can get these values via the Azure Portal or Azure CLI
get_managed_token("https://management.azure.com/", token_args=list(
    mi_res_id="/subscriptions/zzzz-zzzz/resourceGroups/resgroupname/..."
))

# use a refresh token from one resource to get an access token for another resource
tok <- get_azure_token("https://myresource", "mytenant", "app_id")
tok2 <- tok$clone()
tok2$resource <- "https://anotherresource"
tok2$refresh()

# same for AAD v2.0
tok <- get_azure_token(c("https://myresource/.default", "offline_access"),
    "mytenant", "app_id", version=2)
tok2 <- tok$clone()
tok2$scope <- c("https://anotherresource/.default", "offline_access")
tok2$refresh()


# manually adding auth header for a HTTP request
tok <- get_azure_token("https://myresource", "mytenant", "app_id")
header <- httr::add_headers(Authorization=paste("Bearer", tok$credentials$access_token))
httr::GET("https://myresource/path/for/call", header, ...)


## End(Not run)
```

get_manual_token *Get a manual Azure token*

Description

Create an Azure token object from a pre-existing access token string. This is useful when you have obtained a token externally (e.g., via Azure CLI, Python, or another authentication mechanism) and want to use it with the AzureR ecosystem.

Usage

```
get_manual_token(token, type = "Bearer", tenant = NULL, resource = NULL)
```

Arguments

`token`	A character string containing the access token.
`type`	The token type, usually "Bearer".
`tenant`	Optional tenant ID. If NULL, will be extracted from JWT claims if possible.
`resource`	Optional resource/audience URL or GUID. If NULL, will be extracted from JWT claims if possible.

Details

This function creates an AzureManualToken object that wraps an externally-obtained access token. The token object can then be used with packages like AzureGraph, AzureRMR, and other AzureR family packages.

If the provided token is a JWT (JSON Web Token), the function will attempt to parse it to extract metadata like tenant ID, resource, and expiration time. For opaque tokens or tokens that cannot be parsed, you can provide the tenant and resource parameters manually.

Value

An object of class AzureManualToken, inheriting from AzureToken.

Token sources

Common ways to obtain tokens externally include:

- Azure CLI: az account get-access-token --resource <resource>
- Azure PowerShell: Get-AzAccessToken -ResourceUrl <resource>
- Python (azure-identity): DefaultAzureCredential().get_token(<scope>)
- MSAL libraries in various languages

Limitations

Manual tokens have the following limitations compared to tokens obtained via get_azure_token:

- Cannot be automatically refreshed when they expire
- Are not cached to disk
- May have incomplete metadata if JWT parsing fails

See Also

AzureManualToken, get_azure_token, decode_jwt

Examples

```
## Not run:
# Example: Use a token from Azure CLI
# First, get the token from command line:
# az account get-access-token --resource https://graph.microsoft.com --query accessToken -o tsv

raw_token <- "eyJ0eXAiOiJKV1QiLC..."
token <- get_manual_token(raw_token)

# Check token properties
print(token)
token$validate()

# Use with AzureGraph
library(AzureGraph)
```

```
gr <- ms_graph$new(token = token)

# For opaque tokens, provide metadata explicitly
token2 <- get_manual_token(
    token = "opaque_token_string",
    tenant = "your-tenant-id",
    resource = "https://management.azure.com/"
)

## End(Not run)
```

`normalize_tenant` *Normalize GUID and tenant values*

Description

These functions are used by `get_azure_token` to recognise and properly format tenant and app IDs. `is_guid` can also be used generically for identifying GUIDs/UUIDs in any context.

Usage

```
normalize_tenant(tenant)

normalize_guid(x)

is_guid(x)
```

Arguments

`tenant`	For `normalize_tenant`, a string containing an Azure Active Directory tenant. This can be a name ("myaadtenant"), a fully qualified domain name ("myaad-tenant.onmicrosoft.com" or "mycompanyname.com"), or a valid GUID.
`x`	For `is_guid`, a character string; for `normalize_guid`, a string containing a *validly formatted* GUID.

Details

A tenant can be identified either by a GUID, or its name, or a fully-qualified domain name (FQDN). The rules for normalizing a tenant are:

1. If `tenant` is recognised as a valid GUID, return its canonically formatted value
2. Otherwise, if it is a FQDN, return it
3. Otherwise, if it is one of the generic tenants "common", "organizations" or "consumers", return it
4. Otherwise, append ".onmicrosoft.com" to it

These functions are vectorised. See the link below for the GUID formats they accept.

Value

For `is_guid`, a logical vector indicating which values of `x` are validly formatted GUIDs.

For `normalize_guid`, a vector of GUIDs in canonical format. If any values of `x` are not recognised as GUIDs, it throws an error.

For `normalize_tenant`, the normalized tenant IDs or names.

See Also

get_azure_token

Parsing rules for GUIDs in .NET. `is_guid` and `normalize_guid` recognise the "N", "D", "B" and "P" formats.

Examples

```
is_guid("72f988bf-86f1-41af-91ab-2d7cd011db47")    # TRUE
is_guid("{72f988bf-86f1-41af-91ab-2d7cd011db47}")  # TRUE
is_guid("72f988bf-86f1-41af-91ab-2d7cd011db47}")   # FALSE (unmatched brace)
is_guid("microsoft")                               # FALSE

# all of these return the same value
normalize_guid("72f988bf-86f1-41af-91ab-2d7cd011db47")
normalize_guid("{72f988bf-86f1-41af-91ab-2d7cd011db47}")
normalize_guid("(72f988bf-86f1-41af-91ab-2d7cd011db47)")
normalize_guid("72f988bf86f141af91ab2d7cd011db47")

normalize_tenant("microsoft")     # returns 'microsoft.onmicrosoft.com'
normalize_tenant("microsoft.com") # returns 'microsoft.com'
normalize_tenant("72f988bf-86f1-41af-91ab-2d7cd011db47") # returns the GUID

# vector arguments are accepted
ids <- c("72f988bf-86f1-41af-91ab-2d7cd011db47", "72f988bf86f141af91ab2d7cd011db47")
is_guid(ids)
normalize_guid(ids)
normalize_tenant(c("microsoft", ids))
```

Index