FLAAT Documentation

Marcus Hardt

Dec 05, 2023

CONTENTS

Python support for OIDC Access Tokens – FLAAT. Use decorators for authorising access to OIDC authenticated REST APIs.

CHAPTER

ONE

DOCUMENTATION

If you want to protect a REST interface, that is used with OpenID Connect (OIDC), this documentation is what you are looking for.

Flaat supports to limit access to any REST endpoint you have. We do support this by making use of decorators. Three different decorators are currently supported:

1.1 Installation

The recommended way to install is with `pip`:

```
pip install flaat
```

This way you will ensure that the stable version is fetched from pip, rather than development or unstable version.

1.1.1 Installing the development version

The development version of flaat can be installed from the `master` branch of the GitHub repository and can be installed as follows (note the `-e` switch to install it in editable or "develop mode"):

```
git clone https://github.com/indigo-dc/flaat
pip install -e ./flaat
```

1.2 Example: aiohttp

The following is an example of using flaat with **aiohttp**. Using flaat with Flask or fastapi is basically the same. You can use the example using:

```
# run the server:
python ./examples/example_aio.py


# Do the following commands in a different shell

# With oidc agent:
curl http://localhost:8080/info -H "Authorization:Bearer $(oidc-token <account name>)"
```

(continues on next page)

(continued from previous page)

```
# With access token:
curl http://localhost:8080/info -H "Authorization:Bearer eyJraWQ..."
```

```
import logging

from aiohttp import web
from flaat import AuthWorkflow
from flaat.aio import Flaat
from flaat.exceptions import FlaatException
from flaat.requirements import CheckResult
from flaat.requirements import get_claim_requirement
from flaat.requirements import get_vo_requirement
from flaat.user_infos import UserInfos

# do some log setup so we can see something
logging.basicConfig(level="WARNING")
logging.getLogger("flaat").setLevel("DEBUG")


##########
# aio application
app = web.Application()
routes = web.RouteTableDef()

# Our FLAAT instance
flaat = Flaat()

# Insert OPs that you trust here
flaat.set_trusted_OP_list(
    [
        "https://b2access.eudat.eu/oauth2/",
        "https://b2access-integration.fz-juelich.de/oauth2",
        "https://unity.helmholtz-data-federation.de/oauth2/",
        "https://login.helmholtz-data-federation.de/oauth2/",
        "https://login-dev.helmholtz.de/oauth2/",
        "https://login.helmholtz.de/oauth2/",
        "https://unity.eudat-aai.fz-juelich.de/oauth2/",
        "https://services.humanbrainproject.eu/oidc/",
        "https://accounts.google.com/",
        "https://login.elixir-czech.org/oidc/",
        "https://iam-test.indigo-datacloud.eu/",
        "https://iam.deep-hybrid-datacloud.eu/",
        "https://iam.extreme-datacloud.eu/",
        "https://aai.egi.eu/oidc/",
        "https://aai.egi.eu/auth/realms/egi",
        "https://aai-demo.egi.eu/auth/realms/egi",
        "https://aai-demo.egi.eu/oidc",
        "https://aai-dev.egi.eu/oidc",
        "https://oidc.scc.kit.edu/auth/realms/kit/",
        "https://proxy.demo.eduteams.org",
        "https://wlcg.cloud.cnaf.infn.it/",
        "https://proxy.eduteams.org/",
```

(continues on next page)

(continued from previous page)

```
    ]
)

# verbosity:
#     0: Errors
#     1: Warnings
#     2: Infos
#     3: Debug output
flaat.set_verbosity(3)


# # Required for using token introspection endpoint:
# flaat.set_client_id('')
# flaat.set_client_secret('')


# this will customize error responses for the user (used down below)
def my_on_failure(exception: FlaatException, user_infos: UserInfos = None):
    user = "no auth"
    if user_infos is not None:
        user = str(user_infos)
    return web.Response(
        text=f"Custom my_on_failure invoked:\nError Message: {exception}\nUser: {user}"
    )


@routes.get("/")
async def root(request):
    text = """This is an example for useing flaat with AIO. These endpoints are␣
↪available:
    /info                       General info about the access_token (if provided)
    /authenticated              Requires a valid user
    /authenticated_callback     Requires a valid user, uses a custom callback on error
    /authorized_claim           Requires user to have one of two claims
    /authorized_vo              Requires user to have an entitlement
    /full_custom                Fully custom auth handling
        """
    return web.Response(text=text)


@routes.get("/info")
@flaat.inject_user_infos(
    strict=False,  # we don't fail if there is no user
)
async def info(
    request,
    user_infos: UserInfos = None,  # here is the variable that gets injected, this␣
↪should have a default value
):
    message = "No userinfo"
    if user_infos is not None:
        message = user_infos.toJSON()
```

(continues on next page)

(continued from previous page)

```
    return web.Response(text=message)


@routes.get("/authenticated")
@flaat.is_authenticated()
async def authenticated_user(request):
    return web.Response(text="This worked: there was a valid login")


@routes.get("/authenticated_callback")
@flaat.is_authenticated(
    on_failure=my_on_failure,  # called if there is no authentication
)
async def valid_user_own_callback(request):
    """instead of giving an error this will return the custom error response from `my_on_
↪failure`"""
    return web.Response(text="This worked: there was a valid login")


@routes.get("/authorized_claim")
@flaat.requires(
    get_claim_requirement(  # the user needs to satisfy this requirement (having one of␣
↪the email claims)
        ["admin@foo.org", "dev@foo.org"],
        claim="email",
        match=1,
    ),
)
async def authorized_claim(request):
    return web.Response(text="This worked: User has the claim")


@routes.get("/authorized_vo")
@flaat.requires(
    get_vo_requirement(
        [
            "urn:geant:h-df.de:group:m-team:feudal-developers",
            "urn:geant:h-df.de:group:MyExampleColab#unity.helmholtz.de",
        ],
        "eduperson_entitlement",
        match=1,
    )
)
async def authorized_vo(request):
    return web.Response(text="This worked: user has the required entitlement")


# If you need maxmimum customizability, there is AuthWorkflow:

# User requirements
user_reqs = get_claim_requirement("bar", "foo")
```

(continues on next page)

(continued from previous page)

```
def my_request_check(user_infos: UserInfos, *args, **kwargs):
    if len(args) != 1:
        return CheckResult(False, "Missing request object")

    return CheckResult(True, "The request is allowed")


def my_process_args(user_infos: UserInfos, *args, **kwargs):
    """
    We can manipulate the view functions arguments here
    The user is already authenticated at this point, therefore we have `user_infos`,
    therefore we can base our manipulations on the users identity.
    """
    kwargs["email"] = user_infos.get("email", "")
    return (args, kwargs)


custom = AuthWorkflow(
    flaat,  # needs the flaat instance
    user_requirements=user_reqs,
    request_requirements=my_request_check,
    process_arguments=my_process_args,
    on_failure=my_on_failure,
    ignore_no_authn=True,  # Don't fail if there is no authentication
)


@routes.get("/full_custom")
@custom.decorate_view_func  # invoke the workflow here
async def full_custom(request, email=""):
    return web.Response(
        text=f"This worked: The custom workflow did succeed\nThe users email is: {email}"
    )


app.add_routes(routes)

##########
# Main

if __name__ == "__main__":
    web.run_app(app, host="0.0.0.0", port=8080)
```

1.3 Example: flask

The following is an example of using flaat with **flask**.

```
# Flaat example with Flask
import logging
from flaat import AuthWorkflow
from flaat.config import AccessLevel
from flaat.flask import Flaat
from flaat.requirements import CheckResult, HasSubIss, IsTrue
from flaat.requirements import get_claim_requirement
from flaat.requirements import get_vo_requirement

from flask import Blueprint, Flask, abort, current_app
from werkzeug import Response


# -----------------------------------------------------------------------
# Basic configuration example -------------------------------------------
logging.basicConfig(level="WARNING")
logging.getLogger("flaat").setLevel("DEBUG")
logging.getLogger("werkzeug").setLevel("DEBUG")

# Standard flask blueprint snippet, source:
# https://flask.palletsprojects.com/en/2.1.x/blueprints
frontend = Blueprint("frontend", "frontend")


# Set a list of access levels to use
def is_admin(user_infos):
    return user_infos.user_info['email'] in current_app.config['ADMIN_EMAILS']


flaat = Flaat([
    AccessLevel("user", HasSubIss()),
    AccessLevel("admin", IsTrue(is_admin)),
])


class Config(object):

    # Defines the list of Flaat trusted OIDC providers
    TRUSTED_OP_LIST = [
        "https://aai-demo.egi.eu/oidc",
        "https://aai-demo.egi.eu/auth/realms/egi",
        "https://aai-dev.egi.eu/oidc",
        "https://aai.egi.eu/oidc/",
        "https://aai.egi.eu/auth/realms/egi",
        "https://accounts.google.com/",
        "https://b2access-integration.fz-juelich.de/oauth2",
        "https://b2access.eudat.eu/oauth2/",
        "https://iam-test.indigo-datacloud.eu/",
        "https://iam.deep-hybrid-datacloud.eu/",
```

(continues on next page)

(continued from previous page)

```
        "https://iam.extreme-datacloud.eu/",
        "https://login-dev.helmholtz.de/oauth2/",
        "https://login.elixir-czech.org/oidc/",
        "https://login.helmholtz-data-federation.de/oauth2/",
        "https://login.helmholtz.de/oauth2/",
        "https://oidc.scc.kit.edu/auth/realms/kit/",
        "https://orcid.org/",
        "https://proxy.demo.eduteams.org",
        "https://services.humanbrainproject.eu/oidc/",
        "https://unity.eudat-aai.fz-juelich.de/oauth2/",
        "https://unity.helmholtz-data-federation.de/oauth2/",
        "https://wlcg.cloud.cnaf.infn.it/",
        "https://proxy.eduteams.org/",
    ]

    # Additional example configuration:
    ADMIN_EMAILS = ["admin@foo.org", "dev@foo.org"]


# -----------------------------------------------------------------
# Production configuration example ---------------------------------
class ProductionConfig(Config):

    # In production you might want to reduce the number of OP
    TRUSTED_OP_LIST = ["https://aai.egi.eu/oidc/"]
    FLAAT_ISS = "https://aai.egi.eu/oidc/"

    # Define your request timeout for production
    FLAAT_REQUEST_TIMEOUT = 1.2

    # Required for using token introspection endpoint:
    FLAAT_CLIENT_ID = "oidc-agent"
    FLAAT_CLIENT_SECRET = ""


# -----------------------------------------------------------------
# Development configuration example --------------------------------
class DevelopmentConfig(Config):

    # High timeouts might simplify debugging
    FLAAT_REQUEST_TIMEOUT = 30

    # On development certificate verification might not be needed
    FLAAT_VERIFY_TLS = False
    FLAAT_VERIFY_JWT = False


# -----------------------------------------------------------------
# Testing configuration example ------------------------------------
class TestingConfig(Config):

    # Set TESTING to True to run all Flask plugins on testing mode
```

(continues on next page)

(continued from previous page)

```
    TESTING = True

    # When testing to run requirements as close as possible to production
    FLAAT_REQUEST_TIMEOUT = 1.2


# ------------------------------------------------------------------
# Standard flask Application Factories snippet, source -------------
# https://flask.palletsprojects.com/en/2.1.x/patterns/appfactories
def create_app(config=f"{__name__}.ProductionConfig"):
    app = Flask(__name__)
    app.config.from_object(config)

    # Init application plugins
    flaat.init_app(app)
    # db.init_app(app)
    # mail.init_app(app)

    # Register blueprints
    app.register_blueprint(frontend)
    # app.register_blueprint(admin)
    # app.register_blueprint(other)

    return app


# ------------------------------------------------------------------
# Routes definition ------------------------------------------------
@frontend.route("/", methods=["GET"])
def root():
    text = """This is an example for using flaat with Flask.
    The following endpoints are available:
        /info                       General info about the access_token
        /info_no_strict             General info without token validation
        /authenticated              Requires a valid user
        /authenticated_callback     Requires a valid user, uses a custom callback on␣
↪error
        /authorized_level           Requires user to fit the specified access level
        /authorized_claim           Requires user to have one of two claims
        /authorized_vo              Requires user to have an entitlement
        /full_custom                Fully custom auth handling
    """
    return Response(text, mimetype="text/plain")


# ------------------------------------------------------------------
# Call with user information ---------------------------------------
@frontend.route("/info", methods=["GET"])
@flaat.inject_user_infos()  # Fail if no valid authentication is provided
def info_strict_mode(user_infos):
    return user_infos.toJSON()
```

(continues on next page)

(continued from previous page)

```
@frontend.route("/info_no_strict", methods=["GET"])
@flaat.inject_user_infos(strict=False)  # Pass with invalid authentication
def info(user_infos=None):
    return user_infos.toJSON() if user_infos else "No userinfo"


# ----------------------------------------------------------------
# Endpoint which requires of an authenticated user ----------------
@frontend.route("/authenticated", methods=["GET"])
@flaat.is_authenticated()
def authenticated():
    return "This worked: there was a valid login"


# ----------------------------------------------------------------
# Instead of giving an error this will return the custom error
# response from `my_on_failure` -----------------------------------
def my_on_failure(exception, user_infos=None):
    text = f"""Custom callback 'my_on_failure' invoked:
        Error Message: {exception}
        User: {user_infos if user_infos else "No Auth"}
    """
    abort(401, description=text)


@frontend.route("/authenticated_callback", methods=["GET"])
@flaat.is_authenticated(on_failure=my_on_failure)
def authenticated_callback():
    return "This worked: there was a valid login"


# ----------------------------------------------------------------
# Endpoint which requires an access level -------------------------
@frontend.route("/authorized_level", methods=["GET"])
@flaat.access_level("admin")
def authorized_level():
    return "This worked: user has the required rights"


# ----------------------------------------------------------------
# The user needs to satisfy a certain requirement -----------------
email_requirement = get_claim_requirement(
    ["admin@foo.org", "dev@foo.org"],
    claim="email",
    match=1,
)


@frontend.route("/authorized_claim", methods=["GET"])
@flaat.requires(email_requirement)
def authorized_claim():
```

(continues on next page)

(continued from previous page)

```
    return "This worked: User has the claim"


# ---------------------------------------------------------------------
# The user needs belong to a certain virtual organization -----------
vo_requirement = get_vo_requirement(
    [
        "urn:mace:egi.eu:group:test:foo",
        "urn:mace:egi.eu:group:test:bar",
    ],
    "mock_entitlements",
    match=2,
)


@frontend.route("/authorized_vo", methods=["GET"])
@flaat.requires(vo_requirement)
def authorized_vo():
    return "This worked: user has the required entitlement"


# ---------------------------------------------------------------------
# For maximum customization use AuthWorkflow ------------------------
def my_request_check(user_infos, *args, **kwargs):
    if len(args) != 1:
        return CheckResult(False, "Missing request object")
    return CheckResult(True, "The request is allowed")


def my_process_args(user_infos, *args, **kwargs):
    """We can manipulate the view functions arguments here The user is
    already authenticated at this point, therefore we have `user_infos`,
    therefore we can base our manipulations on the users identity.
    """
    kwargs["email"] = user_infos.get("email", "")
    return (args, kwargs)


custom = AuthWorkflow(
    flaat,  # needs the flaat instance
    user_requirements=get_claim_requirement("bar", "foo"),
    request_requirements=my_request_check,
    process_arguments=my_process_args,
    on_failure=my_on_failure,
    ignore_no_authn=True,  # Don't fail if there is no authentication
)


@frontend.route("/full_custom", methods=["GET"])
@custom.decorate_view_func  # invoke the workflow here
def full_custom(email=""):
    text = f"""This worked: The custom workflow did succeed:
```

(continues on next page)

(continued from previous page)

```
        The users email is: {email}
    """
    return Response(text, mimetype="text/plain")


# -------------------------------------------------------------------
# Main function -----------------------------------------------------
if __name__ == "__main__":
    app = create_app("ProductionConfig")
    app.run(host="0.0.0.0", port=8081)
```

1.4 On the CLI: flaat-userinfo

1.4.1 Installation

flaat-userinfo is included with the *flaat installation*.

1.4.2 Description

flaat-userinfo is a simple tool to gather all oidc user info based on access tokens.

The tool can be used in multiple ways:

- Directly pass an access token.
- Name an *oidc-agent* account, which is used to retrieve an access token.
- Use an access token from environment variables (e.g. *ACCESS_TOKEN*).

1.4.3 Options

```
usage: flaat-userinfo [-h] [--my-config MY_CONFIG] [--client_id CLIENT_ID] [--client_
↪secret CLIENT_SECRET] [--oidc-agent-account OIDC_AGENT_ACCOUNT] [--issuer ISSUER] [--
↪audience AUDIENCE]
                      [--skip_tls_verify] [--skip_jwt_verify] [--accesstoken] [--
↪userinfo] [--introspection] [--all] [--quiet] [--verbose] [--machine-readable]
                      [access_token ...]

flaat-userinfo

positional arguments:
  access_token          An access token (without 'Bearer ')

options:
  -h, --help            show this help message and exit
  --my-config MY_CONFIG, -c MY_CONFIG
                        config file path
  --client_id CLIENT_ID
                        Specify the client_id of an oidc client. This is needed for␣
```

(continues on next page)

(continued from previous page)

```
↪token introspection.
  --client_secret CLIENT_SECRET
                        Specify the client_secret of an oidc client. This is may be␣
↪needed for token introspection.
  --oidc-agent-account OIDC_AGENT_ACCOUNT, -o OIDC_AGENT_ACCOUNT
                        Name of oidc-agent account for access token retrieval
  --issuer ISSUER, -i ISSUER
                        Specify issuer (OIDC Provider)
  --audience AUDIENCE, --aud AUDIENCE
                        Specify an intended audience for the requested access
                        token. Multiple audiences can be provided as a space
                        separated list. Only used when token is retrieved via
                        the oidc-agent. Ignored if OP does not support
                        audience setting.
  --skip_tls_verify     Disable TLS verification
  --skip_jwt_verify     Disable JWT verification
  --accesstoken, -at    Show access token info (default)
  --userinfo, -ui       Show user info (default)
  --introspection, -in  Show introspection info (default)
  --all, -a
  --quiet, -q           Enable quiet mode. This will only show requested information, no␣
↪explanatory text
  --verbose, -v         Enable verbose mode. This will also print debug messages.
  --machine-readable, -m
                        Make stdout machine readable
```

1.4.4 Quick examples

To use a raw access token with **flaat-userinfo**, just pass it as an argument:

```
flaat-userinfo eyJraWQ...
```

If you have a loaded *oidc-agent* account called "foo", you can use **flaat-userinfo** using:

```
flaat-userinfo -o foo
```

1.5 Development

1.5.1 Repository

Clone the repository to start developing:

```
git clone https://github.com/indigo-dc/flaat
cd ./flaat
```

1.5.2 Testing

We need access token(s) to run tests. We use oidc-agent for handling access tokens. The test suite uses environment variables for configuration. You can configure the test suite using a dotenv file:

```
cp .env-template .env   # use the template
<editor> .env           # set the correct values in the dotenv file
```

You should preferably configure two oidc agent accounts: One for an OIDC provider that issues JWTs and one that does not. The following file is the environment template. You will almost certainly need to change *OIDC_AGENT_ACCOUNT* and *NON_JWT_OIDC_AGENT_ACCOUNT*:

```
### JWT ACCESS TOKEN
# the shortname depends on how you setup your oidc agent
export OIDC_AGENT_ACCOUNT="egi"

# the issuer of the oidc agent account
export FLAAT_ISS="https://aai.egi.eu/auth/realms/egi"

# These claims must point to two lists of at least two elements in the userinfo
export FLAAT_CLAIM_ENTITLEMENT="eduperson_entitlement"
export FLAAT_CLAIM_GROUP="eduperson_scoped_affiliation"

# To test token introspection we need client id / secret
export FLAAT_CLIENT_ID="oidc-agent"
export FLAAT_CLIENT_SECRET="" # oidc agent needs no secret
### END JWT ACCESS TOKEN

### OPTIONAL NON-JWT ACCESS TOKEN
export NON_JWT_OIDC_AGENT_ACCOUNT="google"
export NON_JWT_FLAAT_ISS="https://accounts.google.com"
### END OPTIONAL NON-JWT ACCESS TOKEN


### OPTIONAL AUD ACCESS TOKEN; OP must support setting AT audience claim
export AUD_OIDC_AGENT_ACCOUNT="wlcg"
export AUD_FLAAT_ISS="https://wlcg.cloud.cnaf.infn.it/"
### END OPTIONAL AUD ACCESS TOKEN
```

1.5.3 Tox

We use tox to run the tests for supported python versions, lint the code using pylint and build this beautiful documentation:

```
tox             # Do everything
tox -e docs     # Only build the docs
tox -e pylint   # Only lint the code
tox -e py310    # Run a test for a specific python version
```

1.5.4 Code conventions

We use pyright for static type checking. Code is formatted using black.

1.5.5 Override auth using environment variables

Important: Be careful with these variables and never use them in production.

You may find setting the following environment variable useful:

- ***export DISABLE_AUTHORIZATION_AND_ASSUME_AUTHORIZED_USER=YES***
  Bypasses user authorization done by the decorators.
- ***export DISABLE_AUTHENTICATION_AND_ASSUME_AUTHENTICATED_USER=YES***
  Bypasses user authentication done by the decorators. This also bypasses the authorization.

1.5.6 Releasing to PyPI

To build a new version use:

```
git tag <new version>   # Tag the release version
git push                # Push the tag

make dist               # build the release
make upload             # upload it to PyPI (needs a valid PyPI account configured in ~/.
↪pypirc)
```

Read the Docs will automatically update the documentation for the git tag.

CHAPTER

TWO

REPORTING BUGS

Issues are managed at github.

CHAPTER

THREE

WHY THIS NAME?

Previously FLAAT only supported Flask, hence the name FLAsk support for Access Tokens (FLAAT).

CHAPTER

FOUR

API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

4.1 API

Important: Do not use the base class *flaat.BaseFlaat* directly. Better use the framework specific classes `flaat.flask.Flaat`, `flaat.aio.Flaat` and `flaat.fastapi.Flaat`.

Here is the documentation of the most important modules of *flaat* with regard to developers, that want to use flaat in the application:

4.1.1 flaat

Python support for OIDC Access Tokens – FLAAT. Use decorators for authorising access to OIDC authenticated REST APIs.

class BaseFlaat

Uses OIDC access tokens to provide authentication and authorization for multiple webframe works. This is the base class. Use the framework specific classes `flaat.flask.Flaat`, `flaat.aio.Flaat` and `flaat.fastapi.Flaat` directly.

You usually use a global instance of Flaat, configure it (see *flaat.config.FlaatConfig*) and then access its decorators (e.g. *is_authenticated()*, *requires()*, *inject_object()* and *access_level()*).

get_user_infos_from_access_token(*access_token*, *issuer_hint=''*)

This method is used to retrieve all infos about an user. You don't need to call this manually, as the decorators will automatically do it for you.

Parameters

access_token (`str`) – The access token of the user. The token must not start with 'Bearer '.

Return type

`Optional[`*UserInfos*`]`

Returns

A *flaat.user_infos.UserInfos* instance with all the infos that could be retrieved. If no info could be retrieved, then *None* is returned.

authenticate_user(**args*, ***kwargs*)

authenticate user needs the same arguments as the view_func it is called from.

Return type
Optional[*UserInfos*]

inject_object(*infos_to_object=None*, *key='object'*, *strict=True*)

Injects an object into a view function given a method to translate a UserInfos instance into the object. This is useful for injecting user model instances.

Parameters

- **infos_to_object** (Optional[Callable[[*UserInfos*], Any]]) – A function that translates a *flaat.user_infos.UserInfos* instance to a custom object.
- **key** – The key with which the generated object is injected into the *kwargs* of the view function.
- **strict** – If set to *True* this decorator if fail when there is nothing to inject.

Return type
Callable

Returns
A decorator for a view function.

inject_user_infos(*key='user_infos'*, *strict=True*)

A decorator which injects the current users *flaat.user_infos.UserInfos* into the view function.

Parameters

- **key** – The key with which the user info is injected.
- **strict** – If set to *True*, an unauthenticated user will not be able to use the view functions and cause an error instead.

Return type
Callable

Returns
A decorator for a view function.

requires(*requirements*, *on_failure=None*)

This returns a decorator that will make sure, that the user fits the requirements before the view function is called. If the user does not, an exception
for the respective web framework will be thrown, so the user sees the correct error.

Parameters

- **requirements** (Union[*Requirement*, Callable[[], *Requirement*], List[Union[*Requirement*, Callable[[], *Requirement*]]]]) – One *flaat.requirements.Requirement* instance or a list of requirements the user needs to fit to have access to the decorated function. If the requirements are wrapped in a callable, it will be lazy evaluated once the view_func is called.
- **on_failure** (Optional[Callable[[*FlaatException*, Optional[*UserInfos*]], Union[Any, NoReturn]]]) – Optional function to customize the handling of an error. This function can either raise an exception or return a response which should be returned in place of the response from the view function.

Return type
Callable

Returns
A decorator for a view function.

access_level(*access_level_name*, *on_failure=None*)

Parameters

- **access_level_name** (str) – The name of the access_level that the user needs to use the view function.
- **on_failure** (Optional[Callable[[*FlaatException*, Optional[*UserInfos*]], Union[Any, NoReturn]]]) – Can be used to either deliver an error response to the user, or raise a specific exception.

Return type
Callable

Returns
A decorator, that can be used to decorate a view function.

is_authenticated(*on_failure=None*)

This can be used to make sure that users are identified (as in they have a subject and an issuer). If you actually want to access the users infos we recommend using *inject_user_infos()* or *inject_object()* instead.

Parameters
on_failure (Optional[Callable[[*FlaatException*, Optional[*UserInfos*]], Union[Any, NoReturn]]]) – Can be used to either deliver an error response to the user, or raise a specific exception.

Return type
Callable

Returns
A decorator for a view function

class AuthWorkflow(*flaat*, *user_requirements=None*, *request_requirements=None*, *process_arguments=None*, *on_failure=None*, *ignore_no_authn=False*)

This class can be used if you need maximum customizability for your decorator. It encapsulates the complete workflow of a decorator.

Parameters

- **flaat** (*BaseFlaat*) – The flaat instance that is currently in use.
- **user_requirements** (Union[*Requirement*, Callable[[], *Requirement*], List[Union[*Requirement*, Callable[[], *Requirement*]]], None]) – Requirement which the user all needs to match, like with requires().
- **request_requirements** (Union[Callable[[*UserInfos*, tuple, dict], *CheckResult*], List[Callable[[*UserInfos*, tuple, dict], *CheckResult*]], None]) – A callable which determines if a users request is allowed to proceed. This function is handy if you want to evaluate the arguments for the view function before against the users permissions.
- **process_arguments** (Optional[Callable[[*UserInfos*, tuple, dict], Tuple[tuple, dict]]]) – As with inject_object(), this can be used to inject data into the view function.
- **on_failure** (Optional[Callable[[*FlaatException*, Optional[*UserInfos*]], Union[Any, NoReturn]]]) – Can be used to either deliver an error response to the user, or raise a specific exception.
- **ignore_no_authn** – If set to *True* a failing authentication of the user will not cause exceptions.

Returns
A class instance, which is used by decorating view functions with its *decorate_view_func()* method.

`decorate_view_func`(*view_func*)

Parameters
view_func (Callable) – The view function to decorate.

Return type
Callable

Returns
The decorated view function.

4.1.2 flaat.user_infos

`class UserInfos`(*access_token_info*, *user_info*, *introspection_info*)

Infos about an access token and the user it belongs to. This class acts like a dictionary with regards to claims. So *infos["foo"]* will give you the claim if it does exist in one of the underlying dicts.

`access_token_info: Optional[Any] = None`

Is set to *AccessTokenInfo* if the respective access token was a JWT.

`user_info: dict`

user_info is the user info dictionary from the user info endpoint of the issuer.

`introspection_info: Optional[dict] = None`

Is the data returned from the token intropsection endpoint of the issuer.

`post_process_dictionaries`()

post_process_dictionaries can be used to do post processing on the raw dictionaries after initialization. Extend this class and overwrite this method to do custom post processing. Make sure to call *super().post_process_dictionaries()*, so the post processing done here is picked up.

`property valid_for_secs: int | None`

Is set if we now about the expiry of these user infos.

`property issuer: str`

The issuer of the access token

`property subject: str`

The users subject at the issuer

`toJSON`()

Render these infos to JSON

Return type
str

4.1.3 flaat.access_tokens

class AccessTokenInfo(*complete_decode*, *verification*)

Infos from a JWT access token

header: dict

The JWTs JOSE header

body: dict

The JWTs data payload

signature: str

The JWTs JWS signature

verification: Optional[dict]

Infos about the verification of the JWT. If set to *None*, then the JWT data is unverified.

class FlaatPyJWKClient(*uri*, *cache_keys=False*, *max_cached_keys=16*, *cache_jwk_set=True*, *lifespan=300*, *headers=None*, *timeout=30*, *ssl_context=None*)

Fixes the jwt.PyJWKClient class:

- **get_signing_keys**
  - does not call self.get_jwk_set(), since it fails when "enc" keys are present
  - returns only keys used for signing (e.g. filters out keys with "use" == "enc")
- **get_signing_key_from_jwt**
  - tries to retrieve keys by id only if "kid" is specified in token header
  - otherwise, it tries to infer the key type ("kty") from the algorithm used to sign the token ("alg")
  - "alg" is always present in JWT header
- an additional method get_signing_key_by_alg

4.1.4 flaat.requirements

This module contains classes to express diverse requirements which a user needs to satisfy in order to use a view function.

The convenience functions *get_claim_requirement()* and *get_vo_requirement()* are recommended to construct individual requirements.

If you want to combine multiple requirements use the "meta requirements" *AllOf* and *N_Of*.

class CheckResult(*is_satisfied*, *message*, *data=None*)

CheckResult is the result of an *is_satisfied_by* check

is_satisfied: bool

Is True if the requirement was satisfied by the user info

message: str

Message describing the check result. This could be an error message.

class Requirement

Requirement is the base class of all requirements. The have a method *is_satisfied_by* which returns a *CheckResult* instance.

class Satisfied

Satisfied is always satisfied

class Unsatisfiable

Unsatisfiable is never satisfied

class IsTrue(*func*)

IsTrue is satisfied if the provided func evaluates to True

Parameters

func (Callable[[*UserInfos*], bool]) – A function that is used to determine if a user info satisfies custom requirements.

class MetaRequirement(**reqs*)

MetaRequirement is a requirements consisting of multiple sub-requirements Use the childs AllOf or N_Of directly.

property requirements: List[*Requirement*]

do the lazy loading of callables

class AllOf(**reqs*)

AllOf is satisfied if all of its sub-requirements are satisfied. If there are no sub-requirements, this class is never satisfied.

class N_Of(*n*, **reqs*)

N_Of is satisfied if at least *n* of its sub-requirements are satisfied. If there are no sub-requirements, this class is never satisfied.

class OneOf(**reqs*)

OneOf is satisfied if at least one of its sub-requirements are satisfied. If there are no sub-requirements, this class is never satisfied.

class HasSubIss

HasSubIss is satisfied if the user has a subject and an issuer

class HasClaim(*required*, *claim*)

HasClaim is satisfied if the user has the specified claim value

class HasAudience(*required*, *claim*)

HasAudience is satisfied if the user's access token was issued for a specific audience

class HasAARCEntitlement(*required*, *claim*)

HasAARCEntitlement is satisfies if the user has the provided AARC-G002/G069 entitlement If the argument *required* is not a parseable AARC entitlement, we revert to equals comparisons.

get_claim_requirement(*required*, *claim*, *match='all'*)

Parameters

- **required** (Union[str, List[str]]) – The claim values that the user needs to have.
- **claim** (str) – The claim of the value in *required*, e.g. *eduperson_entitlement*.
- **match** (Union[str, int]) – May be "all" if all required values need to be matched, or "one" or an integer if a specific amount needs to be matched.

Return type

Requirement

Returns

A requirement that is satisfied if the user has the claim value(s) of *required*.

get_vo_requirement(*required*, *claim*, *match='all'*)

Equivalent to *get_claim_requirement()*, but works for both groups and AARC entitlements.

Return type
Requirement

get_audience_requirement(*required*)

Equivalent to *get_claim_requirement()*, but specific to audience claim.

Return type
Requirement

4.1.5 flaat.exceptions

exception FlaatException

An error occured inside flaat. The cause can be misconfiguration or other not user related errors. This exception will cause a response with status 500 if unhandled.

exception FlaatForbidden

The user is forbidden from using the service. This exception will cause a response with status 403 if unhandled.

exception FlaatUnauthenticated

The users identity could not be determined. Probably there was no access token or the access tokens issuer could not be determined.

This exception will cause a response with status 401 if unhandled.

4.2 Configuration Options

Flaat instances are configured like `FlaatConfig` below.

class FlaatConfig

The configuration for Flaat instances.

set_access_levels(*access_levels*)

Set the list of access levels for use with *flaat.BaseFlaat.access_level()*. This list will overwrite the default access levels.

Parameters
access_level – List of *AccessLevel* instances.

set_verbosity(*verbosity*, *set_global=False*)

Set the logging verbosity.

Parameters

- **verbosity** (int) – Verbosity integer from 0 (= error) to 3 (= debug)
- **set_global** – If set to *True* the logging level will be set for all loggers

set_issuer(*issuer*)

Pins the given issuer. Only users of this issuer will be able to use services.

Parameters
issuer (str) – Issuer URL of the pinned issuer.

set_trusted_OP_list(*trusted_op_list*)

Sets a list of OIDC providers that you trust. This means that users of these OPs will be able to use your services.

Parameters
trusted_op_list (List[str]) – A list of the issuer URLs that you trust. An example issuer is: 'https://iam.deep-hybrid-datacloud.eu/'.

set_verify_tls(*verify_tls=True*)

Only use for development and debugging. Set to *False* to skip TLS certificate verification while processing requests.

set_verify_jwt(*verify_jwt=True*)

Set to *False* to skip JWT verification while processing requests.

set_client_id(*client_id=''*)

Set a client id for token introspection

set_client_secret(*client_secret=''*)

Set a client secret for token introspection

set_request_timeout(*timeout=1.2*)

Set the timeout for individual requests (retrieving issuer configs, user infos and introspection infos). Note that the total runtime of a decorator could be significantly more, based on your *trusted_op_list*.

Parameters
timeout (float) – Request timeout in seconds.

class AccessLevel(*name*, *requirement*)

Access levels are basically named requirements. An example would be two access levels 'user' and 'admin', which have requirements for the respective level.

In order to use an access level with a flaat instance, you need to use `flaat.BaseFlaat.set_access_levels()` to add them to the list.

name: str

The name of the access level. This is used in *flaat.BaseFlaat.access_level()* to identify this access level.

requirement: Union[*Requirement*, Callable[[], *Requirement*]]

The requirement that users of this access level need to satisfy. If this is a callable, then the requirement is lazily loaded at runtime using the callable.

CHAPTER

FIVE

INDICES

- genindex
- modindex

PYTHON MODULE INDEX

INDEX

R

S

T

U

V