nickw-flask-utils Documentation

Release 0.0.0.dev0

Nick Whyte

Nov 23, 2021

Contents

A set of utilities to make working with flask web applications easier.

This repository contains a collection of web utilities that are common across many of our projects.

It makes the most sense to keep these in a common repository to promote code re-use and modularity.

View the Full Documentation at ReadTheDocs

CHAPTER 1

Contents:

1.1 Celery

1.1.1 API

flask_utils.celery.**create_celery**(*name*, *config_obj*, *inject_version=True*, ***kwargs*)

Creates a celery app.

Parameters

- **config_obj** – The configuration object to initiaze with
- **inject_version** –

  bool: Whether or not to inject the application's version number. Attempts to get version number

  using *flask_utils.deployment_release.get_release()*
- **kwargs** – Other arguments to pass to the `Celery` instantiation.

Returns An initialized celery application.

flask_utils.celery.**create_db_session**(*celery*)

Creates an SQLA Database scoped session. Requires SQLAlchemy.

Uses `SQLALCHEMY_DATABASE_URI` and `SQLALCHEMY_POOL_RECYCLE` to create an apropriate engine.

If you are using MySQL and `SQLALCHEMY_POOL_RECYCLE` is not specified, you'll have a bad time - this is required, as MySQL kills old sessions.

Parameters **celery** – The celery application to create the app on

Returns A SQLAlchemy scoped_session instance.

1.2 Config

1.2.1 API

flask_utils.config.**build_url**(*url*, *scheme=None*, *username=None*, *password=None*, *hostname=None*, *port=None*, *path=None*)

Parse a URL and override specific segments of it.

Parameters

- **url** – The url to parse/build upon
- **scheme** –
- **username** –
- **password** –
- **hostname** –
- **port** –
- **path** –

Returns A URL with overridden components

1.3 Deployment Release

1.3.1 API

flask_utils.deployment_release.**get_release**()

Opens a file `version.txt` and returns it's stripped contents.

Returns The stripped file contents

1.4 Pagination

1.4.1 API

flask_utils.pagination.**paginated**(*basequery*, *schema_type*, *offset=None*, *limit=None*)

Paginate a sqlalchemy query

Parameters

- **basequery** – The base query to be iterated upon
- **schema_type** – The `Marshmallow` schema to dump data with
- **offset** – (Optional) The offset into the data. If omitted it will be read from the query string in the `?offset=` argument. If not query string, defaults to 0.
- **limit** – (Optional) The maximum results per page. If omitted it will be read from the query string in the `?limit=` argument. If not query string, defaults to 20.

Returns The page's data in a namedtuple form `(data=, errors=)`

1.5 Restful

1.5.1 API

exception flask_utils.restful.**ExpectedJSONException**
Thrown when JSON was expected in a flask request but was not provided

classmethod handle(*exc*)
A handler for this type of exception.

Usage:

```
app.errorhandler(ExpectedJSONException)(ExpectedJSONException.handle)
```

flask_utils.restful.**get_and_expect_json**()
Returns the ‘‘flask.request.get_json()‘, however if no JSON data was decoded, will raise a *ExpectedJSONException*

flask_utils.restful.**output_json**(*data*, *code*, *headers=None*)

```
api.representations['application/json'] = output_json
```

Generates better formatted responses for RESTFul APIs.

If the restful resource responds with a string, with a non 200 error, the response will look like

```
{
    "_errors": ["String the user responded with"]
}
```

Likewise, a string return value with a 200 response will look like:

```
{
    "message": "String the user returned with."
}
```

If a Non-200 response occured, and flask-restful added it’s own error message in the “message” field of the response data, this data is moved into “_errors”:

```
{
    "_errors": ["You don't have the permission to access the requested resource...
↪"]
}
```

All data is returned using flask’s jsonify method. This means you can use simplejson to return decimal objects from your flask restful resources.

1.6 Sentry

1.6.1 API

1.7 Testing

1.7.1 API

class flask_utils.testing.**AppReqTestHelper**
Adds convenience request methods on the testcase object.

Assumes a flask app client is defined on `self.client`

delete(**args*, ***kwargs*)
Perform a delete request to the application.

get(**args*, ***kwargs*)
Perform a get request to the application.

patch(**args*, ***kwargs*)
Perform a patch request to the application.

post(**args*, ***kwargs*)
Perform a post request to the application.

put(**args*, ***kwargs*)
Perform a put request to the application.

class flask_utils.testing.**CRUDTestHelper**
A helper to test generic CRUD operations on an endpoint.

do_crud_test(*endpoint*, *data_1=None*, *data_2=None*, *key='id'*, *check_keys=[]*, *keys_from_prev=[]*, *create=True*, *delete=True*, *update=True*, *read=True*, *initial_count=0*)
Begins the CRUD test.

Parameters

- **endpoint** – `string`: The endpoint to test
- **data1** – `dict`: Data to create the initial entity with (POST)
- **data2** – `dict`: Data to update the entity with (PUT)
- **key** – `string`: The key field in the response returned when performing a create.
- **check_keys** – `list`: A list of keys to compare `data_1` and `data_2` to returned API responses. (To ensure expected response data)
- **keys_from_prev** – `list`: A list of keys to check that they persisted after a create/update.
- **create** – `bool`: Should create a new object and test it's existence
- **delete** – `bool`: Should delete the newly created object and test that it has been deleted.
- **update** – `bool`: Should performs PUT (update)
- **read** – `bool`: Should perform a plural read
- **initial_count** – `int`: The initial number of entities in the endpoint's dataset

class flask_utils.testing.**PrivilegeTestHelper**
Adds a helper to test endpoint privileges in an application.

do_test_privileges(*endpoint*, *data*, *object_id*, *expected_codes*)
Test privileges on a specific endpoint.

Parameters

- **endpoint** – The endpoint to test. e.g. */api/v1/classes*
- **data** – The data to use when performing a *PUT/POST* (dict)
- **object_id** – The id of the singular object to test permissions on *PUT/DELETE/GET*
- **expected_codes** – The expected response codes when performing each endpoint request. E.g.

```
{
    'plural-get': 200,
    'get': 200,
    'delete': 403,
    'post': 200,
    'put': 200,
}
```

1.8 Token Auth

1.8.1 Motivation

Building applications that require authentication and protection is almost always necessary.

We've found the most practical token exchange implementation exists with 2 different kinds of tokens and the following authentication flow:

1. A user performs a `POST` to a login endpoint `/api/v1/login` with their credentials.
2. The API validates these credentials, and begins the token exchange process. The API generates a **refresh token**, and stores it. This token must have all the information required to generate a **shortlived token**. A **shortlived token** is then generated. A possible response from this endpoint could be:

```
{
    "refreshToken": "abcdefg",
    "token": "shortlived-token-could-be-a-jwt"
}
```

3. The client stores both these tokens. (Possibly in `localstorage`).
4. Before every request, the client checks if their shortlived token has expired. If it hasn't expired, they can just send the shortlived token as usual. However, if it is expired, a renewal must occur.
5. **The client performs a `POST` to the renew endpoint, providing the** `refresh token` they recieved at login.
6. The server checks if the refresh token is still valid (ie, hasn't been revoked, or inactive for too long), and using this returns a new **shortlived token**. A possible response from this endpoint could be:

```
{
    "token": "shortlived-token-could-be-a-jwt"
}
```

7. The user stores the new token. If it was found that the **refresh token** had been revoked, then the user will be prompted to log back into the app.
8. When the user wishes to log out, a `POST` request to a logout endpoint occurs, sending the user's shortlived token.
9. The server revokes the shortlived token's associated refresh token, thus, invalidating any further renewals.

There are 4 main advantages to using an authentication flow like this:

1. Users never have to log in again as long as they stay active.
2. Tokens are stateless (Unless you need to check for revocation on every request, but this is still a very lightweight operation.)
3. As long as shortlived tokens have a short enough expiry, a compromised shortlived token can have little impact.
4. Heavy token renewal/stateful operations can be minimised to happening ONLY when the token has expired.

Thus, this package exposes helpers to assist in implementing such an authentication workflow. It makes no assumptions about your refresh token objects. It does assume that your prefered short lived token type is a JWT.

1.8.2 Usage

Within your project, create class for your short lived token. Within this class, we need to define a few things. Let's store a user_id, a list of scopes, and the associated refresh token on the short lived token. We can use a marshmallow schema for this:

```
import logging
import random
import string

log = logging.getLogger(__name__)


class ShortlivedToken(ShortlivedTokenMixin):
    class TokenSchema(ShortlivedTokenMixin.TokenSchema):
        rfid = fields.String(attribute='refresh_token_id')
        user_id = fields.String(attribute='user_id')
        scopes = fields.List(fields.String(), attribute='scopes')

    def __init__(self, refresh_token_id, user_id, scopes, *args, **kwargs):
        super(ShortlivedToken, self).__init__(*args, **kwargs)

        self.refresh_token_id = refresh_token_id
        self.user_id = user_id
        self.scopes = scopes
```

I used shortend names like `rfid` in the schema to cut down on bytes transfered with the token. JWT's are meant to be lightweight.

Finally, we need to define a way to load a `ShortlivedToken` from a refresh token. We define the `classmethod`, `from_refresh_token` to do this:

```
import logging
import random
import string

log = logging.getLogger(__name__)
```

(continues on next page)

(continued from previous page)

```
class ShortlivedToken(ShortlivedTokenMixin):
    class TokenSchema(ShortlivedTokenMixin.TokenSchema):
        rfid = fields.String(attribute='refresh_token_id')
        user_id = fields.String(attribute='user_id')
        scopes = fields.List(fields.String(), attribute='scopes')

    def __init__(self, refresh_token_id, user_id, scopes, *args, **kwargs):
        super(ShortlivedToken, self).__init__(*args, **kwargs)

        self.refresh_token_id = refresh_token_id
        self.user_id = user_id
        self.scopes = scopes

    @classmethod
    def from_refresh_token(Cls, refresh_token):
        # Fetch the token expiry from the application configuration
        shortlived_expiry = current_app.config['SHORT_LIVED_TOKEN_EXPIRY']

        return Cls(
            refresh_token_id=refresh_token.id,
            user_id=refresh_token.user_id,
            scopes=refresh_token.scopes,
            expiry=datetime.datetime.now(pytz.UTC) + shortlived_expiry,
```

As mentioned above, we use information from the refresh token to build a ShortlivedToken. To complete the example, we'll add an in-memory refresh token store and a basic implementation:

```
import uuid

import pytz

random_alpha = string.digits + string.ascii_letters


class RefreshToken():
    def __init__(self, user_id, scopes):
        self.id = uuid.uuid4().hex
        self.user_id = user_id
        self.scopes = scopes
        self.token = ''.join(random.choice(random_alpha) for _ in range(80))
```

Finally, we will create a flask app and implement the 3 endpoints for authentication. `/login`, `/logout`, and `/renew`. This is now the final implementation:

```
import datetime
import logging
import random
import string
import uuid

import pytz
from flask import current_app, Flask, request, abort, jsonify, g
```

(continues on next page)

(continued from previous page)

```
from marshmallow import fields

from flask_utils.token_auth import (
    ShortlivedTokenMixin, parse_auth_header, auth_required)

log = logging.getLogger(__name__)


class ShortlivedToken(ShortlivedTokenMixin):
    class TokenSchema(ShortlivedTokenMixin.TokenSchema):
        rfid = fields.String(attribute='refresh_token_id')
        user_id = fields.String(attribute='user_id')
        scopes = fields.List(fields.String(), attribute='scopes')

    def __init__(self, refresh_token_id, user_id, scopes, *args, **kwargs):
        super(ShortlivedToken, self).__init__(*args, **kwargs)

        self.refresh_token_id = refresh_token_id
        self.user_id = user_id
        self.scopes = scopes

    @classmethod
    def from_refresh_token(Cls, refresh_token):
        # Fetch the token expiry from the application configuration
        shortlived_expiry = current_app.config['SHORT_LIVED_TOKEN_EXPIRY']

        return Cls(
            refresh_token_id=refresh_token.id,
            user_id=refresh_token.user_id,
            scopes=refresh_token.scopes,
            expiry=datetime.datetime.now(pytz.UTC) + shortlived_expiry,
        )


random_alpha = string.digits + string.ascii_letters


class RefreshToken():
    def __init__(self, user_id, scopes):
        self.id = uuid.uuid4().hex
        self.user_id = user_id
        self.scopes = scopes
        self.token = ''.join(random.choice(random_alpha) for _ in range(80))


# Store the granted refresh tokens in memory.
refresh_tokens = []

app = Flask(__name__)
app.config.update({
    'SHORT_LIVED_TOKEN_EXPIRY': datetime.timedelta(hours=1),
    'SECRET_KEY': 'supersecret'
})


@app.route('/login', methods=['POST'])
def login():
```

(continues on next page)

(continued from previous page)

```
    # Check if the credentials were correct
    if request.form.get('username') != 'test' or \
            request.form.get('password') != 'test':
        abort(401)

    # Create a new refresh token
    refresh_token = RefreshToken(user_id=request.form.get('username'),
                                 scopes=['360noscope'])

    # Persist the refresh token so we can renew it later
    refresh_tokens.append(refresh_token)

    shortlived_token = ShortlivedToken.from_refresh_token(refresh_token)
    log.info("Generated token with payload: {}".format(shortlived_token))

    return jsonify({
        'token': shortlived_token.dump(),
        'refreshToken': refresh_token.token
    })


@app.route('/logout', methods=['POST'])
@parse_auth_header(ShortlivedToken)
@auth_required()
def logout():
    # Find the associated refresh token
    for refresh_token in refresh_tokens:
        if refresh_token.id == g.token.refresh_token_id:
            break  # Found the associated token

    else:  # nobreak
        # Couldn't find the token. Maybe it has been revoked.
        abort(401)

    # Remove the refresh token from the store. It has now been revoked.
    refresh_tokens.remove(refresh_token)

    return jsonify({
        'status': 'success'
    })


@app.route('/renew', methods=['POST'])
def renew():
    token_string = request.form.get('refreshToken')

    # Find the refresh token in the store
    for refresh_token in refresh_tokens:
        if refresh_token.token == token_string:
            break  # Found the token that we need.

    else:  # nobreak
        # Couldn't find the token. Oops
        abort(401)

    # Make a new shortlived token
    shortlived_token = ShortlivedToken.from_refresh_token(refresh_token)
```

(continues on next page)

(continued from previous page)

```
    log.info("Generated token with payload: {}".format(shortlived_token))

    # Respond to the client with the new token
    return jsonify({
        'token': shortlived_token.dump()
    })


@app.route('/protected', methods=['POST'])
@parse_auth_header(ShortlivedToken)
@auth_required()
def protected():
    return "Welcome, {}".format(g.token.user_id)


if __name__ == '__main__':
    app.run(debug=True, port=5005)
```

Using this example, you should be able to exchange credentials for a refresh token, a shortlived token, and perform subsequent renewalls and revokations.

In the above example, the functions *parse_auth_header()* and *auth_required()* are used on the protected endpoint and logout endpoint.

1.8.3 API

class flask_utils.token_auth.**ShortlivedTokenMixin**(*expiry=None*, *issuer=None*, *subject=None*, *audience=None*, *not_before=None*, *issued_at=None*)

A base class for implementing a short-lived token using JWTs

Parameters

- **expiry** – datetime: The expiry of this token
- **issuer** – string
- **subject** – string
- **audience** – string
- **not_before** – datetime: A datetime of when the token becomes valid for use
- **issued_at** – datetime: When the token was issued. This value is overwritten during dump()

class TokenSchema(**, only: Union[Sequence[str], Set[str], None] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Optional[Dict[KT, VT]] = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: Optional[str] = None*)

The schema to use to serialize/de-serialize JWT's with.

dump(*secret=None*)

Dump the token into a stringified JWT.

Parameters **secret** – The secret to sign the JWT with. If this is omitted, the secret will be sourced from current_app.config['SECRET_KEY']

Returns The stringified JWT.

classmethod from_refresh_token(*refresh_token*)
Given a refresh token, return an instance of ShortLivedTokenMixin configured using information from `refresh_token`.

Parameters **refresh_token** – A refresh token instance.

Returns A new ShortLivedToken instance.

> **Warning:** You must implement this method

classmethod load(*token_string*, *secret=None*, *issuer=None*, *audience=None*)
Load from a JWT (`token_string`)

Parameters

- **token_string** – The raw string to load from
- **secret** – The secret that the JWT was signed with to check validity. If this is omitted, the secret will be sourced from `current_app.config['SECRET_KEY']`
- **issuer** – The issuer the JWT decode should expect
- **audience** – The audience the JWT decode should expect

Returns A de-serialized ShortLivedToken instance.

flask_utils.token_auth.**auth_required**()
Force authentication on an endpoint. Checks if `g.token` is not None, and returns a `401` if it is.

Example:

```
@app.route('/auth-required')
@auth_required()
def my_endpoint():
    return "Hello World"
```

flask_utils.token_auth.**parse_auth_header**(*token_cls*, *auth_header=True*, *query_string=True*, *secret=None*)
A decorator to extract a token from either the `Authorization` header OR the query string parameter `?token=`.

Parameters

- **token_cls** – An instance of *ShortlivedTokenMixin* OR a class which implements the classmethod `load(token_string, secret)`. An instance will be available on *g.token*.
- **auth_header** – (optional, `bool`) Extract the token from the auth header (Default: `True`)
- **query_string** – (optional, `bool`) Extract the token from the query string (Default: `True`)
- **secret** – (optional) A secret to pass to `token_cls.load(raw, secret)`

Any wrapped function will be able to access both `g.token` and `g.raw_token` to read the `token_cls` instance and raw token string respectively.

Authorization header expects tokens in the format of `Bearer <token string>`

1.9 Webargs

1.9.1 API

class flask_utils.webargs.**BetterFlaskParser**(*location: Optional[str] = None, *, unknown: Optional[str] = '_default', error_handler: Optional[Callable[[...], NoReturn]] = None, schema_class: Optional[Type[CT_co]] = None*)

A Flask-Restful compatible parser for WebArgs.

handle_error(*error*)

Don't raise a `HTTPException` via `abort`. Instead we will throw the `ValidationError` and handle it with our flask Exception handler.

This allows a common code path for both Flask-Restful AND standard Flask Views.

flask_utils.webargs.**handle_validation_error**(*exc*)

When using *BetterFlaskParser*, if an exception occurs, it will throw the original `ValidationError`. This circumvents the capture inside Flask-Restful (if it is being used at all).

Instead of capturing all 422 `HTTPExceptions`, you register this error handler with `ValidationError`:

```
app.errorhandler(ValidationError)(handle_validation_error)
```

This function will produce a jsonified response with the field errors from the ValidationError.

> **Warning:** This handler is incompatible with the standard `FlaskParser`, since it throws `HTTPExceptions` (via `abort`). Registering this handler with `errorhandler(422)` will not work.

Python Module Index

f

Index

S