Sanic-JWT-Extended Documentation

Release 0.1.0 beta

Lewis "devArtoria" Kim

Dec 23, 2019

Contents

Warning: Using 0.x version is highly **not** recommended. 0.x version is not maintained anymore and documentation for new version doesn't use readthedocs anymore. for the new documentation, visit https://sanic-jwt-extended.seonghyeon.dev/

Contents:

CHAPTER 1

Installation

The easiest way to start working with this extension with pip:

```
$ pip install sanic-jwt-extended
```

CHAPTER 2

Basic Usage

In its simplest form, there is not much to using flask_jwt_extended. You use *create_access_token()* to make new access JWTs, the *jwt_required()* decorator to protect endpoints, and `jwt_identity` in the given token argument to get the identity of a JWT in a protected endpoint.

```
from sanic import Sanic
from sanic.response import json
from sanic.request import Request
from sanic_jwt_extended import (
    JWTManager,
    jwt_required,
    create_access_token,
    create_refresh_token,
)
import uuid
from sanic_jwt_extended.tokens import Token

app = Sanic(__name__)

# Setup the Sanic-JWT-Extended extension
app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
JWTManager(app)


# Provide a method to create access tokens. The create_access_token()
# function is used to actually generate the token, and you can return
# it to the caller however you choose.
@app.route("/login", methods=["POST"])
async def login(request: Request):
    if not request.json:
        return json({"msg": "Missing JSON in request"}, status=400)

    username = request.json.get("username", None)
    password = request.json.get("password", None)
    if not username:
```

(continues on next page)

(continued from previous page)

```
        return json({"msg": "Missing username parameter"}, status=400)
    if not password:
        return json({"msg": "Missing password parameter"}, status=400)

    if username != "test" or password != "test":
        return json({"msg": "Bad username or password"}, status=403)

    # Identity can be any data that is json serializable
    access_token = await create_access_token(identity=username, app=request.app)
    refresh_token = await create_refresh_token(
        identity=str(uuid.uuid4()), app=request.app
    )
    return json(
        dict(access_token=access_token, refresh_token=refresh_token), status=200
    )


# Protect a view with jwt_required, which requires a valid access token
# in the request to access.
@app.route("/protected", methods=["GET"])
@jwt_required
async def protected(request: Request, token: Token):
    # Access the identity of the current user with get_jwt_identity
    current_user = token.jwt_identity
    return json(dict(logined_as=current_user))


if __name__ == "__main__":
    app.run()
```

To access a jwt_required protected view, all we have to do is send in the JWT with the request. By default, this is done with an authorization header that looks like:

```
Authorization: Bearer <access_token>
```

We can see this in action using CURL:

```
$ curl http://localhost:5000/protected
{
  "msg": "Missing Authorization Header"
}

$ curl -H "Content-Type: application/json" -X POST \
  -d '{"username":"test","password":"test"}' http://localhost:8000/login
{
  "access_token": <ACCESS TOKEN>"
}

$ curl -H "Authorization: Bearer <ACCESS TOKEN>" http://localhost:5000/protected
{
  "logined_as": "test"
}
```

NOTE: Remember to change the secret key of your application, and insure that no one is able to view it. The JSON Web Tokens are signed with the secret key, so if someone gets that, they can create arbitrary tokens, and in essence log in as any user.

CHAPTER 3

Insert User Claims

You may want to store additional information in the access token which you could later access in the protected views. This can be done with the fill the *user_claims* parameter in the *create_access_token()* and *create_refresh_token()* and the data can be accessed later in a protected endpoint with `jwt_user_claims` in the given token argument.

Storing data in an access token can be good for performance. If you store data in the token, you wont need to look it up from disk next time you need it in a protected endpoint. However, you should take care what data you put in the token. Any data in the access token can be trivially viewed by anyone who can read the token. **Do not** store sensitive information in access tokens!

```
from sanic import Sanic
from sanic.response import json
from sanic.request import Request
from sanic_jwt_extended import (
    JWTManager,
    jwt_required,
    create_access_token,
    create_refresh_token,
)
import uuid
from sanic_jwt_extended.tokens import Token

app = Sanic(__name__)

# Setup the Sanic-JWT-Extended extension
app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
JWTManager(app)

user_claim = {"VERI TAS": "LUX MEA"}


# Provide a method to create access tokens. The create_access_token()
# function is used to actually generate the token, and you can return
# it to the caller however you choose.
```

(continues on next page)

(continued from previous page)

```
@app.route("/login", methods=["POST"])
async def login(request: Request):
    if not request.json:
        return json({"msg": "Missing JSON in request"}, status=400)

    username = request.json.get("username", None)
    password = request.json.get("password", None)
    if not username:
        return json({"msg": "Missing username parameter"}, status=400)
    if not password:
        return json({"msg": "Missing password parameter"}, status=400)

    if username != "test" or password != "test":
        return json({"msg": "Bad username or password"}, status=403)

    # Identity can be any data that is json serializable
    access_token = await create_access_token(
        identity=username, app=request.app, user_claims=user_claim
    )
    return json(dict(access_token=access_token), status=200)


# Protect a view with jwt_required, which requires a valid access token
# in the request to access.
@app.route("/protected", methods=["GET"])
@jwt_required
async def protected(request: Request, token: Token):
    # Access the identity of the current user with get_jwt_identity
    user_claims = token.jwt_user_claims
    return json(dict(data=user_claims))


if __name__ == "__main__":
    app.run()
```

CHAPTER 4

Configuration Options

You can change many options for how this extension works via

```
app.config[OPTION_NAME] = new_options
```

4.1 General Options:

JWT_TOKEN_LOCATION	Where to look for a JWT when processing a request. The options are 'headers', 'cookies', 'query_string', or 'json'. You can pass in a list to check more then one location, such as: ['headers', 'cookies']. Defaults to 'headers'
JWT_ACCESS_TOKEN_EXPIRES	How long an access token should live before it expires. This takes a datetime.timedelta, and defaults to 15 minutes. Can be set to False to disable expiration.
JWT_REFRESH_TOKEN_EXPIRES	How long a refresh token should live before it expires. This takes a datetime.timedelta, and defaults to 30 days. Can be set to False to disable expiration.
JWT_ALGORITHM	Which algorithm to sign the JWT with. See here for the options. Defaults to 'HS256'.
JWT_SECRET_KEY	The secret key needed for symmetric based signing algorithms, such as HS*. If this is not set, we use the sanic SECRET_KEY value instead.
JWT_PUBLIC_KEY	The public key needed for asymmetric based signing algorithms, such as RS* or ES*. PEM format expected.
JWT_PRIVATE_KEY	The private key needed for asymmetric based signing algorithms, such as RS* or ES*. PEM format expected.
JWT_IDENTITY_CLAIM	Claim in the tokens that is used as source of identity. For interoperability, the JWT RFC recommends using 'sub'. Defaults to 'identity' for legacy reasons.
JWT_USER_CLAIMS	Claim in the tokens that is used to store user claims. Defaults to 'user_claims'.
JWT_CLAIMS_IN_REFRESH_TOKEN	If user claims should be included in refresh tokens. Defaults to False.
JWT_ERROR_MESSAGE_KEY	The key of the error message in a JSON error response when using the default error handlers. Defaults to 'msg'.
RBAC_ENABLE	Role-based access control (RBAC) enable option. Defaults to False

4.2 Header Options:

These are only applicable if JWT_TOKEN_LOCATION is set to use headers.

JWT_HEADER_NAME	What header to look for the JWT in a request. Defaults to 'Authorization'
JWT_HEADER_TYPE	What type of header the JWT is in. Defaults to 'Bearer'. This can be an empty string, in which case the header contains only the JWT (insead of something like HeaderName: Bearer <JWT>)

CHAPTER 5

Role-based access control(RBAC)

Sanic-JWT-Extended supports RBAC.

all you have to do is just make `'RBAC_ENABLE'` option to `'True'`, give role to jwt with `'role'` option in *create_access_token()*. and specify role to allow or deny when using *jwt_required()* and *fresh_jwt_required()*

Warning: `'deny'` and `'allow'` option **can not** be used together.

```
from sanic import Sanic
from sanic.response import json
from sanic.request import Request
from sanic_jwt_extended import (
    JWTManager,
    jwt_required,
    create_access_token,
    create_refresh_token,
)
import uuid
from sanic_jwt_extended.tokens import Token

app = Sanic(__name__)

# Setup the Sanic-JWT-Extended extension
app.config["JWT_SECRET_KEY"] = "super-secret"  # Change this!
app.config["RBAC_ENABLE"] = True
JWTManager(app)


# Provide a method to create access tokens. The create_access_token()
# function is used to actually generate the token, and you can return
# it to the caller however you choose.
@app.route("/login", methods=["POST"])
```

(continues on next page)

(continued from previous page)

```
async def login(request: Request):
    username = request.json.get("username", None)

    # Identity can be any data that is json serializable
    access_token = await create_access_token(
        identity=username, role="ADMIN", app=request.app
    )
    return json(dict(access_token=access_token), status=200)


# Protect a view with jwt_required, which requires a valid access token
# in the request to access.
@app.route("/protected", methods=["GET"])
@jwt_required(allow=["ADMIN"])  # default to whitelist mode
async def protected(request: Request, token: Token):
    # Access the identity of the current user with get_jwt_identity
    current_user = token.jwt_identity
    return json(dict(logined_as=current_user))


if __name__ == "__main__":
    app.run(port=9000)
```

CHAPTER 6

API Documentation

In here you will find the API for everything exposed in this extension.

6.1 Sanic-JWT-Extended

class sanic_jwt_extended.**JWTManager**(*app: sanic.app.Sanic*)

An object used to hold JWT settings for the Sanic-JWT-Extended extension. Instances of `JWTManger` are *not* bound to specific apps, so you can create one in the main body of your code and then bind it to your app in a factory function.

__init__(*app: sanic.app.Sanic*)

Create the JWTManager instance. You can either pass a sanic application in directly here to register this extension with the sanic app, or you can call init_app after creating this object (in a factory pattern). :param app: A sanic application

init_app(*app: sanic.app.Sanic*)

Register this extension with the sanic app. :param app: A sanic application

6.2 Protected endpoint decorators

sanic_jwt_extended.**jwt_required**(*function=None*, *allow=None*, *deny=None*)

A decorator to protect a Sanic endpoint. If you decorate an endpoint with this, it will ensure that the requester has a valid access token before allowing the endpoint to be called. and if token check passed this will insert Token object to kwargs, This does not check the freshness of the access token. See also: *fresh_jwt_required()*

sanic_jwt_extended.**jwt_refresh_token_required**(*fn*)

A decorator to protect a Sanic endpoint. If you decorate an endpoint with this, it will ensure that the requester has a valid refresh token before allowing the endpoint to be called.

sanic_jwt_extended.**fresh_jwt_required**(*function=None*, *allow=None*, *deny=None*)

A decorator to protect a Sanic endpoint. If you decorate an endpoint with this, it will ensure that the requester has a valid and fresh access token before allowing the endpoint to be called. See also: *jwt_required()*

sanic_jwt_extended.**jwt_optional**(*fn*)

A decorator to optionally protect a Sanic endpoint If an access token in present in the request, this will insert filled Token object to kwargs. If no access token is present in the request, this will insert Empty Token object to kwargs If there is an invalid access token in the request (expired, tampered with, etc), this will still call the appropriate error handler instead of allowing the endpoint to be called as if there is no access token in the request. and also does not check role

6.3 Verify Tokens in Request

These perform the same actions as the protected endpoint decorators, without actually decorating a function. These are very useful if you want to create your own decorators on top of sanic jwt extended (such as role_required), or

sanic_jwt_extended.decorators.**get_jwt_data_in_request_header**(*app: sanic.app.Sanic*, *request: sanic.request.Request*) → Dict

Get JWT token data from request header with configuration. raise NoAuthorizationHeaderError when no jwt header. also raise InvalidHeaderError when malformed jwt header detected.

Parameters

- **app** – A Sanic application
- **request** – Sanic request object that contains app

Returns Dictionary containing contents of the JWT

sanic_jwt_extended.decorators.**verify_jwt_data_type**(*token_data: dict*, *token_type: str*) → None

Check jwt type with given argument. raise WrongTokenError if token type is not expected type,

Parameters

- **token_data** – Dictionary containing contents of the JWT
- **token_type** – Token type that want to check (ex: access)

6.4 Utilities

sanic_jwt_extended.**create_access_token**(*app*, *identity*, *user_claims=None*, *role=None*, *fresh=False*, *expires_delta=None*)

Create a new access token.

Parameters

- **app** – A Sanic application from request object
- **identity** – The identity of this token, which can be any data that is json serializable. It can also be a python object
- **user_claims** – User made claims that will be added to this token. it should be dictionary.
- **role** – A role field for RBAC
- **fresh** – If this token should be marked as fresh, and can thus access *fresh_jwt_required()* endpoints. Defaults to *False*. This value can also be a *datetime.timedelta* in which case it will indicate how long this token will be considered fresh.

- **expires_delta** – A *datetime.timedelta* for how long this token should last before it expires. Set to False to disable expiration. If this is None, it will use the 'JWT_ACCESS_TOKEN_EXPIRES' config value

Returns An encoded access token

sanic_jwt_extended.**create_refresh_token**(*app*, *identity*, *user_claims=None*, *expires_delta=None*)

Create a new refresh token.

Parameters

- **app** – A Sanic application from request object
- **identity** – The identity of this token, which can be any data that is json serializable. It can also be a python object
- **user_claims** – User made claims that will be added to this token. it should be dictionary.
- **expires_delta** – A *datetime.timedelta* for how long this token should last before it expires. Set to False to disable expiration. If this is None, it will use the 'JWT_REFRESH_TOKEN_EXPIRES' config value

Returns An encoded access token

sanic_jwt_extended.tokens.**encode_access_token**(*identity: str, secret: str, algorithm: str, expires_delta: datetime.timedelta, fresh: Union[datetime.timedelta, bool], user_claims: dict, role: str, identity_claim_key: str, user_claims_key: str, json_encoder: Callable[..., str] = None*) → str

Creates a new encoded (utf-8) access token. :param identity: Identifier for who this token is for (ex, username). This

data must be json serializable

Parameters

- **secret** – Secret key to encode the JWT with
- **algorithm** – Which algorithm to encode this JWT with
- **expires_delta** (*datetime.timedelta or False*) – How far in the future this token should expire (set to False to disable expiration)
- **fresh** – If this should be a 'fresh' token or not. If a datetime.timedelta is given this will indicate how long this token will remain fresh.
- **user_claims** – Custom claims to include in this token. This data must be json serializable
- **role** – A role field for RBAC
- **identity_claim_key** – Which key should be used to store the identity
- **user_claims_key** – Which key should be used to store the user claims
- **json_encoder** – json encoder

Returns Encoded access token

sanic_jwt_extended.tokens.**encode_refresh_token**(*identity*, *secret*, *algorithm*, *expires_delta*, *user_claims*, *identity_claim_key*, *user_claims_key*, *json_encoder=None*)

Creates a new encoded (utf-8) refresh token.

Parameters

- **identity** – Some identifier used to identify the owner of this token
- **secret** – Secret key to encode the JWT with
- **algorithm** – Which algorithm to use for the toek
- **expires_delta** (*datetime.timedelta or False*) – How far in the future this token should expire (set to False to disable expiration)
- **user_claims** – Custom claims to include in this token. This data must be json serializable
- **identity_claim_key** – Which key should be used to store the identity
- **user_claims_key** – Which key should be used to store the user claims
- **json_encoder** – json encoder

Returns Encoded refresh token

sanic_jwt_extended.tokens.**decode_jwt**(*encoded_token: str*, *secret: str*, *algorithm: str*, *identity_claim_key: str*, *user_claims_key: str*) → Dict

Decodes an encoded JWT

Parameters

- **encoded_token** – The encoded JWT string to decode
- **secret** – Secret key used to encode the JWT
- **algorithm** – Algorithm used to encode the JWT
- **identity_claim_key** – expected key that contains the identity
- **user_claims_key** – expected key that contains the user claims

Returns Dictionary containing contents of the JWT

6.5 Token Object

class sanic_jwt_extended.tokens.**Token**(*app: sanic.app.Sanic*, *token: dict*)

Token object that contains decoded token data and passed with kwargs to endpoint function

raw_jwt

Returns full jwt data in dictionary form

jwt_identity

Returns jwt identity claim data (or this can be None if data does not exist)

jwt_user_claims

Returns user claim data

jti

Returns jti data

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

Symbols

C

D

E

F

G

I

J

R

S

T

V