sanic_session Documentation

Release 0.5.0

Suby Raman

Dec 26, 2022

CONTENTS

CHAPTER

ONE

USING THE INTERFACES

For now project has set of different interfaces. You can install each manually or using the extra parameters:

`pip install sanic_session[aioredis]`

Other supported backend keywords:

- `aioredis` (dependency 'aioredis'),
- `redis` ('asyncio_redis'),
- `mongo` ('sanic_motor' and 'pymongo'),
- `aiomcache` ('aiomcache')

1.1 Redis (asyncio_redis)

Redis is a popular and widely supported key-value store. In order to interface with redis, you will need to add `asyncio_redis` to your project. Do so with pip:

`pip install asyncio_redis` or `pip install sanic_session[redis]`

To integrate Redis with `sanic_session` you need to pass a getter method into the `RedisSessionInterface` which returns a connection pool. This is required since there is no way to synchronously create a connection pool. An example is below:

```python
import asyncio_redis

from sanic import Sanic
from sanic.response import text
from sanic_session import Session, RedisSessionInterface

app = Sanic()


class Redis:
    """
    A simple wrapper class that allows you to share a connection
    pool across your application.
    """
    _pool = None

    async def get_redis_pool(self):
        if not self._pool:
```

(continues on next page)

(continued from previous page)

```
            self._pool = await asyncio_redis.Pool.create(
                host='localhost', port=6379, poolsize=10
            )

        return self._pool


redis = Redis()

Session(app, interface=RedisSessionInterface(redis.get_redis_pool))


@app.route("/")
async def test(request):
    # interact with the session like a normal dict
    if not request.ctx.session.get('foo'):
        request.ctx.session['foo'] = 0

    request.ctx.session['foo'] += 1

    response = text(request.ctx.session['foo'])

    return response

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

1.2 Redis (aioredis)

aioredis have little better syntax and more popular since it supported by *aiohttp* team.

`pip install asyncio_redis` or `pip install sanic_session[aioredis]`

This example shows little different approach. You can use classic Flask extensions approach with factory based initialization process. You can use it with different backends also.

```
import aioredis

from sanic import Sanic
from sanic.response import text
from sanic_session import Session, AIORedisSessionInterface

app = Sanic(__name__, load_env=False)
# init extensions
session = Session()

@app.listener('before_server_start')
async def server_init(app, loop):
    # For aioredis 1.x and older
    # app.redis = await aioredis.create_redis_pool(app.config['redis'])
    # For aioredis 2.x
```

(continues on next page)

(continued from previous page)

```
    app.redis = aioredis.from_url(app.config['redis'], decode_responses=True)
    # init extensions fabrics
    session.init_app(app, interface=AIORedisSessionInterface(app.redis))


@app.route("/")
async def test(request):
    # interact with the session like a normal dict
    if not request.ctx.session.get('foo'):
        request.ctx.session['foo'] = 0

    request.ctx.session['foo'] += 1

    response = text(request.ctx.session['foo'])

    return response

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

1.3 Memcache

Memcache is another popular key-value storage system. In order to interface with memcache, you will need to add `aiomcache` to your project. Do so with pip:

`pip install aiomcache` or `pip install sanic_session[aiomcache]`

To integrate memcache with `sanic_session` you need to pass an `aiomcache.Client` into the session interface, as follows:

```
import aiomcache
import uvloop

from sanic import Sanic
from sanic.response import text
from sanic_session import Session, MemcacheSessionInterface

app = Sanic()

# create a uvloop to pass into the memcache client and sanic
loop = uvloop.new_event_loop()

# create a memcache client
client = aiomcache.Client("127.0.0.1", 11211, loop=loop)

# pass the memcache client into the session
session = Session(app, interface=MemcacheSessionInterface(client))

@app.route("/")
async def test(request):
    # interact with the session like a normal dict
```

(continues on next page)

(continued from previous page)

```
    if not request.ctx.session.get('foo'):
        request.ctx.session['foo'] = 0

    request.ctx.session['foo'] += 1

    response = text(request.ctx.session['foo'])

    return response

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True, loop=loop)
```

1.4 In-Memory

`sanic_session` comes with an in-memory interface which stores sessions in a Python dictionary available at `session_interface.session_store`. This interface is meant for testing and development purposes only. **This interface is not suitable for production**.

```
from sanic import Sanic
from sanic.response import text
from sanic_session import Session


app = Sanic()

Session(app)  # because InMemorySessionInterface used by default

# of full syntax:
#   from sanic_session import InMemorySessionInterface
#   session = Session(app, interface=InMemorySessionInterface())

@app.route("/")
async def index(request):
    # interact with the session like a normal dict
    if not request.ctx.session.get('foo'):
        request.ctx.session['foo'] = 0

    request.ctx.session['foo'] += 1

    return text(request.ctx.session['foo'])

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

CHAPTER TWO

API DOCUMENTATION

2.1 InMemorySessionInterface

class sanic_session.**InMemorySessionInterface**(*domain: Optional[str] = None*, *expiry: int = 2592000*, *httponly: bool = True*, *cookie_name: str = 'session'*, *prefix: str = 'session:'*, *sessioncookie: bool = False*, *samesite: Optional[str] = None*, *session_name='session'*, *secure: bool = False*)

__init__(*domain: Optional[str] = None*, *expiry: int = 2592000*, *httponly: bool = True*, *cookie_name: str = 'session'*, *prefix: str = 'session:'*, *sessioncookie: bool = False*, *samesite: Optional[str] = None*, *session_name='session'*, *secure: bool = False*)

2.2 MemcacheSessionInterface

class sanic_session.**MemcacheSessionInterface**(*memcache_connection*, *domain: Optional[str] = None*, *expiry: int = 2592000*, *httponly: bool = True*, *cookie_name: str = 'session'*, *prefix: str = 'session:'*, *sessioncookie: bool = False*, *samesite: Optional[str] = None*, *session_name: str = 'session'*, *secure: bool = False*)

__init__(*memcache_connection*, *domain: Optional[str] = None*, *expiry: int = 2592000*, *httponly: bool = True*, *cookie_name: str = 'session'*, *prefix: str = 'session:'*, *sessioncookie: bool = False*, *samesite: Optional[str] = None*, *session_name: str = 'session'*, *secure: bool = False*)

Initializes the interface for storing client sessions in memcache. Requires a client object establised with *asyncio_memcache*.

Args:

memcache_connection (aiomccache.Client): The memcache client used for interfacing with memcache.

domain (str, optional): Optional domain which will be attached to the cookie.

expiry (int, optional): Seconds until the session should expire.

httponly (bool, optional): Adds the *httponly* flag to the session cookie.

cookie_name (str, optional): Name used for the client cookie.

prefix (str, optional): Memcache keys will take the format of *prefix+session_id*; specify the prefix here.

sessioncookie (bool, optional): Specifies if the sent cookie should be a 'session cookie', i.e no Expires or Max-age headers are included. Expiry is still fully tracked on the server side. Default setting is False.

samesite (str, optional): Will prevent the cookie from being sent by the browser to the target site in all cross-site browsing context, even when following a regular link. One of ('lax', 'strict') Default: None

session_name (str, optional): Name of the session that will be accessible through the request. e.g. If `session_name` is `alt_session`, it should be accessed like that: `request.ctx.alt_session` e.g. And if `session_name` is left to default, it should be accessed like that: `request.ctx.session` Default: 'session'

secure (bool, optional): Adds the *Secure* flag to the session cookie.

2.3 RedisSessionInterface

`class sanic_session.RedisSessionInterface`(*redis_getter: Callable*, *domain: Optional[str] = None*, *expiry: int = 2592000*, *httponly: bool = True*, *cookie_name: str = 'session'*, *prefix: str = 'session:'*, *sessioncookie: bool = False*, *samesite: Optional[str] = None*, *session_name: str = 'session'*, *secure: bool = False*)

`__init__`(*redis_getter: Callable*, *domain: Optional[str] = None*, *expiry: int = 2592000*, *httponly: bool = True*, *cookie_name: str = 'session'*, *prefix: str = 'session:'*, *sessioncookie: bool = False*, *samesite: Optional[str] = None*, *session_name: str = 'session'*, *secure: bool = False*)

Initializes a session interface backed by Redis.

Args:

redis_getter (Callable): Coroutine which should return an asyncio_redis connection pool (suggested) or an asyncio_redis Redis connection.

domain (str, optional): Optional domain which will be attached to the cookie.

expiry (int, optional): Seconds until the session should expire.

httponly (bool, optional): Adds the *httponly* flag to the session cookie.

cookie_name (str, optional): Name used for the client cookie.

prefix (str, optional): Memcache keys will take the format of *prefix+session_id*; specify the prefix here.

sessioncookie (bool, optional): Specifies if the sent cookie should be a 'session cookie', i.e no Expires or Max-age headers are included. Expiry is still fully tracked on the server side. Default setting is False.

samesite (str, optional): Will prevent the cookie from being sent by the browser to the target site in all cross-site browsing context, even when following a regular link. One of ('lax', 'strict') Default: None

session_name (str, optional): Name of the session that will be accessible through the request. e.g. If `session_name` is `alt_session`, it should be accessed like that: `request.ctx.alt_session` e.g. And if `session_name` is left to default, it should be accessed like that: `request.ctx.session` Default: 'session'

secure (bool, optional): Adds the *Secure* flag to the session cookie.

2.4 AIORedisSessionInterface

`class sanic_session.AIORedisSessionInterface`(*redis*, *domain: Optional[str] = None*, *expiry: int = 2592000*, *httponly: bool = True*, *cookie_name: str = 'session'*, *prefix: str = 'session:'*, *sessioncookie: bool = False*, *samesite: Optional[str] = None*, *session_name: str = 'session'*, *secure: bool = False*)

`__init__`(*redis*, *domain: Optional[str] = None*, *expiry: int = 2592000*, *httponly: bool = True*, *cookie_name: str = 'session'*, *prefix: str = 'session:'*, *sessioncookie: bool = False*, *samesite: Optional[str] = None*, *session_name: str = 'session'*, *secure: bool = False*)

Initializes a session interface backed by Redis.

Args:

redis (Callable): aioredis connection or connection pool instance.

domain (str, optional): Optional domain which will be attached to the cookie.

expiry (int, optional): Seconds until the session should expire.

httponly (bool, optional): Adds the *httponly* flag to the session cookie.

cookie_name (str, optional): Name used for the client cookie.

prefix (str, optional): Memcache keys will take the format of *prefix+session_id*; specify the prefix here.

sessioncookie (bool, optional): Specifies if the sent cookie should be a 'session cookie', i.e no Expires or Max-age headers are included. Expiry is still fully tracked on the server side. Default setting is False.

samesite (str, optional): Will prevent the cookie from being sent by the browser to the target site in all cross-site browsing context, even when following a regular link. One of ('lax', 'strict') Default: None

session_name (str, optional): Name of the session that will be accessible through the request. e.g. If `session_name` is `alt_session`, it should be accessed like that: `request.ctx.alt_session` e.g. And if `session_name` is left to default, it should be accessed like that: `request.ctx.session` Default: 'session'

secure (bool, optional): Adds the *Secure* flag to the session cookie.

CHAPTER

THREE

CONFIGURATION

When initializing a session interface, you have a number of optional arguments for configuring your session.

domain (str, optional): Optional domain which will be attached to the cookie. Defaults to None.

expiry (int, optional): Seconds until the session should expire. Defaults to *2592000* (30 days). Setting this to 0 or None will set the session as permanent.

httponly (bool, optional): Adds the *httponly* flag to the session cookie. Defaults to True.

cookie_name (str, optional): Name used for the client cookie. Defaults to "session".

prefix (str, optional): Storage keys will take the format of *prefix+<session_id>*. Specify the prefix here.

sessioncookie (bool, optional): If enabled the browser will be instructed to delete the cookie when the browser is closed. This is done by omitting the *max-age* and *expires* headers when sending the cookie. The *expiry* configuration option will still be honored on the server side. This is option is disabled by default.

samesite (str, optional): One of 'strict' or 'lax'. Defaults to None https://www.owasp.org/index.php/SameSite

session_name (str, optional):

Name of the session that will be accessible through the request.
e.g. If `session_name` is `alt_session`, it should be accessed like that: `request.ctx.alt_session`
e.g. And if `session_name` is left to default, it should be accessed like that: `request.ctx.session`

Note: If you choose to build your application using more than one session object, make sure that they have different:

1. `cookie_name`
2. `prefix` (Only if the two cookies share the same store)
3. And obviously, different: `session_name`

Example 1:

```
session_interface = InMemorySessionInterface(
    domain='.example.com', expiry=0,
    httponly=False, cookie_name="cookie", prefix="sessionprefix:",  samesite="strict")
```

Will result in a session that:

- Will be valid only on *example.com*.
- Will never expire.
- Will be accessible by Javascript.

- Will be named "cookie" on the client.
- Will be named "sessionprefix:<sid>" in the session store.
- Will prevent the cookie from being sent by the browser to the target site in all cross-site browsing context, even when following a regular link.

Example 2:

```
session_interface = InMemorySessionInterface(
    domain='.example.com', expiry=3600, sessioncookie=True,
    httponly=True, cookie_name="myapp", prefix="session:")
```

Will result in a session that:

- Will be valid only on *example.com*.
- Will expire on the server side after 1 hour.
- Will be deleted on the client when the user closes the browser.
- Will *not* be accessible by Javascript.
- Will be named "myapp" on the client.
- Will be named "session:<sid>" in the session store.

CHAPTER

FOUR

TESTING

When building your application you'll eventually want to test that your sessions are behaving as expected. You can use the `InMemorySessionInterface` for testing purposes. You'll want to insert some logic in your application so that in a testing environment, your application uses the `InMemorySessionInterface`. An example is like follows:

main.py

```
import asyncio_redis
import os

from sanic import Sanic
from sanic.response import text
from sanic_session import (
    RedisSessionInterface,
    InMemorySessionInterface
)


app = Sanic()


class Redis:
    _pool = None

    async def get_redis_pool(self):
        if not self._pool:
            self._pool = await asyncio_redis.Pool.create(
                host='localhost', port=6379, poolsize=10
            )

        return self._pool


redis = Redis()

# If we are in the testing environment, use the in-memory session interface
if os.environ.get('TESTING'):
    Session(app, interface = InMemorySessionInterface())
else:
    Session(app, interface = RedisSessionInterface(redis.get_redis_pool))
```

(continues on next page)

(continued from previous page)

```
@app.route("/")
async def index(request):
    if not request.ctx.session.get('foo'):
        request.ctx.session['foo'] = 0

    request.ctx.session['foo'] += 1

    response = text(request.ctx.session['foo'])

    return response

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

Let's say we want to test that the route / does in fact increment a counter on subsequent requests. There's a few things to remember:

- When a session is saved, a `session` parameter is included in the response cookie.
- Use this session ID to retrieve the server-stored session data from the `session_interface`.
- You can also use this session ID on future requests to reuse the same client session.

An example is like follows:

```
import os
os.environ['TESTING'] = 'True'

from main import app, session_interface

import pytest
import aiohttp
from sanic.utils import sanic_endpoint_test


def test_session_increments_counter():
    request, response = sanic_endpoint_test(app, uri='/')

    # A session ID is passed in the response cookies, save that
    session_id = response.cookies['session'].value

    # retrieve the session data using the session_id
    session = session_interface.get_session(session_id)

    assert session['foo'] == 1, 'foo should initially equal 1'

    # use the session ID to test the endpoint against the same session
    request, response = sanic_endpoint_test(
        app, uri='/', cookies={'session': session_id})

    # again retrieve the session data using the session_id
    session = session_interface.get_session(session_id)

    assert session['foo'] == 2, 'foo should increment on subsequent requests'
```

sanic_session is an extension for sanic that integrates server-backed sessions with a Flask-like API.

Install it with pip: `pip install sanic_session`

sanic_session provides a number of *session interfaces* for you to store a client's session data. The interfaces available right now are:

- Redis
- Memcache
- In-Memory (suitable for testing and development environments)

See *Using the interfaces* for instructions on using each.

A simple example uses the in-memory session interface.

```
from sanic import Sanic
from sanic.response import text
from sanic_session import Session

app = Sanic()
Session(app)

@app.route("/")
async def index(request):
    # interact with the session like a normal dict
    if not request.ctx.session.get('foo'):
        request.ctx.session['foo'] = 0

    request.ctx.session['foo'] += 1

    return text(request.ctx.session['foo'])

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

INDEX

Symbols

A

I

M

R