aio-sf-streaming Documentation

Release 0.3.7

Christophe GABARD / papernest.com

Dec 10, 2020

CONTENTS

aio-sf-streaming is a simple Python 3.6 asyncio library allowing to connect and receive live notifications from Salesforce. This library is provided to you by papernest.

See The Force.com streaming API developer guide for more information about the different uses cases and how configure your Salesforce organization.

Simple example

```
import asyncio
from aio_sf_streaming import SimpleSalesforceStreaming

async def print_event():
    # Create client and connect
    async with SimpleSalesforceStreaming(
                    username='my-username',
                    password='my-password',
                    client_id='my-client-id',
                    client_secret='my-client-secret') as client:
        # Subscribe to some push topics
        await client.subscribe('/topic/Foo')
        await client.subscribe('/topic/Bar')
        # Print received message infinitely
        async for message in client.events():
            channel = message['channel']
            print(f"Message received on {channel} : {message}")

loop = asyncio.get_event_loop()
loop.run_until_complete(print_event())
```

Main features

- asyncio compatible library
- Authentication with username/password or refresh token
- Subscribe to push topics and custom events
- Receive events pushed by Salesforce
- Auto-reconnect after too many time of inactivity
- Replay support: replay events missed while your client is disconnected (see Force.com documentation for more information).

aio-sf-streaming only support Python 3.6 for now.

CHAPTER

ONE

USER GUIDE

1.1 Introduction

aio-sf-streaming is a simple client library allowing to connect to the Force.com Streaming API and receive push event from Salesforce.

Salesforce can push two kind of events:

- PushTopics allow you to monitor object and receive notification when the provided SOQL query match after an object creation, update or deletion.
- Generic streaming allow to create custom events not linked to Saleforce object. Apex or API call allow you to trigger the event and receive notification in your streaming client.

aio-sf-streaming allow you to connect to Salesforce, subscribe for some events and receive event when a change in a `PushTopics` is detected or when a generic streaming event is trigged.

1.1.1 License

aio-sf-streaming was created at papernest and is distribued under the MIT license.

1.2 Quickstart

1.2.1 Code organization

aio-sf-streaming is designed in a modular way:

- *BaseSalesforceStreaming* is the base class of the package. It implement the main logic of the streaming API flow. It is an abstract class, you can not use it directly, the class lake of connection capability. You must use one of the connector implementation.
- *Connectors* add connection capabilities to *BaseSalesforceStreaming* allowing to connect to Salesforce. *PasswordSalesforceStreaming* allow to connect on Salesforce with password flow. *RefreshTokenSalesforceStreaming* allow to connect on Salesforce with refresh token flow.
- *Mixins* are provided and can be added to concrete implementation to provide additional capabilities like replay support or use the provided timeout advice. This functionalities can be added by sub-classing a connector and add mixin to your concrete implementation.
- Helper class like *SimpleSalesforceStreaming* simplify implementation with an "all-in-one" class implementation.

1.2.2 Asyncronous and Asyncio

1.2.3 Salesforce connection

BaseSalesforceStreaming allow you to connect with user name and password of the user and client id and secret from the connected app.

Constructor does not establish any connection, you needs to call *BaseSalesforceStreaming.start()* to connect to Salesforce and start Bayeux/CometD protocol. Call :*BaseSalesforceStreaming.stop()* to disconnect and stop connection.

```
client = SimpleSalesforceStreaming(
            username='username',
            password='password',
            client_id='client_id',
            client_secret='client_secret')
await client.start()
# process events
await client.stop()
```

Most of the time, you should not call theses methods directly, you should use the asynchronous context manager interface that will call all of these for you:

```
async with SimpleSalesforceStreaming(
                username='username',
                password='password',
                client_id='client_id',
                client_secret='client_secret') as client:
    # process events
```

1.2.4 Subscribe to events

Two methods *BaseSalesforceStreaming.subscribe()* and *BaseSalesforceStreaming.unsubscribe()* allow you to start receiving new events from a push topic or a generic streaming event and stop when you does not want to receive event anymore.

```
async with SimpleSalesforceStreaming(**credentials) as client:
    # Subscribe to push topic
    await client.subscribe('/topic/Foo')
    # Subscribe to generic event
    await client.subscribe('/u/MyEvent')

    # Process events

    # Unsubscribe from push topic
    await client.unsubscribe('/topic/Foo')
    # Unsubscribe from generic event
    await client.unsubscribe('/u/MyEvent')
```

You can subscribe and unsubscribe at any moment and on other coroutine as soon as the connection is established. You can even start to process without waiting the response:

```
async def process(loop):
    async with SimpleSalesforceStreaming(**credentials, loop=loop) as client:
        loop.create_task(client.subscribe('/topic/Foo'))
        loop.create_task(client.subscribe('/topic/Bar'))

        # Process events

loop = asyncio.get_event_loop()
loop.run_until_complete(process(loop))
```

1.2.5 Receive events

BaseSalesforceStreaming.messages() and *BaseSalesforceStreaming.events()* are used to iterate over events when their are received. The main difference is that *BaseSalesforceStreaming.messages()* provide all events, whereas *BaseSalesforceStreaming.events()* filter internal messages and provide only the events for channel you subscribed.

Both methods are asynchronous generator and should be iterate with *async for*:

```
async with SimpleSalesforceStreaming(**credentials) as client:
    await client.subscribe('/topic/Foo')
    await client.subscribe('/topic/Bar')

    async for event in client.events():
        channel = event['channel']
        print(f"Received an event from {channel} : {event}")
```

Warning: Linked to the underlying protocol, long-pooling based, the client should reconnect as soon as possible. Practically, client have 40 seconds to reconnect. If your processing take a longer time, a new connection should be made. You should avoid doing long processing between each iteration or launch this processing into a background task.

The processing loop is infinite by default. Inside the loop, you can stop easily with a *break*:

```
async with SimpleSalesforceStreaming(**credentials) as client:
    await client.subscribe('/topic/Foo')
    await client.subscribe('/topic/Bar')

    async for event in client.events():
        channel = event['channel']
        if channel == '/topic/Foo':
            break
        else:
            print(event)
```

Outside the main loop, you can call *BaseSalesforceStreaming.ask_stop()* to stop the loop as soon as is possible, even if your loop is waiting for a new message. Please note that, due to the underlying protocol, this can take some time to really happen (the code must wait a timeout from the server, can be as long as 2min).

1.2.6 Replay support

ReplayMixin add support of 24 hours events replay. Each event is associated with an unique id by channel. To support replay, you must override two methods: *ReplayMixin.store_replay_id()* and *ReplayMixin.get_last_replay_id()*.

ReplayMixin.store_replay_id() is called for each received event. The method is called with three arguments:

- the channel,
- the replay id,
- the object creation time (the string provided by SF).

For each channel, this function should store the replay id of the last created object.

ReplayMixin.get_last_replay_id() will be called to retrieve the last replay id for a specific channel. In addition of a specific id, this function can return two special values from the *ReplayType* enum to replay all available events (24 hours history) or only new events after subscription.

The next example will store replay id in memory. In real world application you should store this id in a persistent way:

```
class MyClient(SimpleSalesforceStreaming):
    def __init__(*args, **kwargs):
        self.replays = {}
        super().__init__(*args, **kwargs)

    async def store_replay_id(self, channel, replay_id, creation_time):
        # we does not want to store the replay id if a most recent one is
        # already stored
        last_storage = self.replays.get(channel, None)
        creation_time = parse_time(creation_time)       # Custom function to implement
        if last_storage and last_storage[0] > creation_time:
            return
        self.replays[channel] = (creation_time, replay_id)

    async def get_last_replay_id(self, channel):
        # Retrieve last replay
        last_storage = self.replays.get(channel, None)
        # If we have not any stored replay id, we can either replay all
        # events or only subscribe to new ones.
```

(continues on next page)

(continued from previous page)

```
        if not last_storage:
            return ReplayType.NEW_EVENTS
        return last_storage[1]
```

CHAPTER

TWO

REFERENCE DOCUMENTATION

2.1 Developer Interface

This part of the documentation covers all the interfaces of *aio_sf_streaming*.

2.1.1 Code organization

The package is separated in 5 mains modules:

- High-level classes and helpers are provided to have a quickly functional client. See *Main Interface* section.
- The *Base class* section describe the low-level base class that implement the main client logic.
- To authenticate on Salesforce, you must use one connector that add authentication capability to `BaseSalesforceStreaming`. See *Connectors* section for a list of available connectors.
- Finally, *Mixins* extend `BaseSalesforceStreaming` capabilities and can be added easily as opt-in option by sub classing.

2.1.2 Main Interface

class aio_sf_streaming.**SimpleSalesforceStreaming**(*username*, *password*, *client_id*, *client_secret*, ***, *sandbox=False*, *version='42.0'*, *loop=None*, *connector=None*, *login_connector=None*, *retry_sub_duration=0.1*, *retry_factor=1.0*, *retry_max_duration=30.0*, *retry_max_count=20*)

A simple helper class providing all-in-one functionalities.

Parameters

- **username** (str) – User login name
- **password** (str) – User password
- **client_id** (str) – OAuth2 client Id
- **client_secret** (str) – Oauth2 client secret
- **sandbox** (bool) – If True, the connexion will be made on a sandbox, from https://test.salesforce.com instead of the main login route at https://login.salesforce.com.

- **version** (str) – The API version to use. For example '42.0'.
- **loop** (Optional[AbstractEventLoop]) – Asyncio loop used
- **connector** (Optional[BaseConnector]) – aiohttp connector used for main session. Mainly used for test purpose.
- **login_connector** (Optional[BaseConnector]) – aiohttp connector used during connection. Mainly used for test purpose.
- **retry_sub_duration** (float) – Duration between subscribe retry if server is too buzy.
- **retry_factor** (float) – Factor amplification between each successive retry
- **retry_max_duration** (float) – Maximum value of the retry duration
- **retry_max_count** (int) – Maximum count of retry, after this count is reach, response or exception are propagated.

Usage example:

```
class MyClient(SimpleSalesforceStreaming):
    def __init__(self):
        self.replays = []
        super().__init__(username='my-username',
                         password='my-password',
                         client_id='my-client-id',
                         client_secret='my-client-secret')

    async def store_replay_id(self, channel, replay_id, creation_time):
        # We only store replay id without any use
        self.replays.append((channel, replay_id, creation_time))

    async def get_last_replay_id(self, channel):
        # We ask for only use new events
        return EventType.NEW_EVENTS

async def print_events():
    async with MyClient() as client:
        await client.subscribe('/topic/Foo')
        async for message in client.events():
            channel = message['channel']
            print(f"Message received on {channel} : {message}")

loop = asyncio.get_event_loop()
loop.run_until_complete(print_event())
```

SimpleSalesforceStreaming inherit all members from their base class. Only main one, for external use, are listed here.

coroutine start(*self*)
See *BaseSalesforceStreaming.start()*

Return type None

coroutine subscribe(*self*, *channel*)
See *BaseSalesforceStreaming.subscribe()*

Return type List[Dict[str, Any]]

async-for messages(*self*)
See *BaseSalesforceStreaming.messages()*

Return type Dict[str, Any]

async-for events(*self*)
Asynchronous generator that fetch new events and return one as soon as one is available:

```
async for message in client.events():
    channel = message['channel']
    print(channel, ':', message)
```

This method is different from *BaseSalesforceStreaming.messages()* because it filter messages and provide only those related to the channels you subscribed.

Return type Dict[str, Any]

coroutine store_replay_id(*self*, *channel*, *replay_id*, *creation_time*)
Callback called to store a replay id. You should override this method to implement your custom logic.

Parameters

- **channel** (str) – Channel name
- **replay_id** (int) – replay id to store
- **creation_time** (str) – Creation time. You should store only the last created object but you can not know if you received event in order without this. This value is the string provided by SF.

Return type None

coroutine get_last_replay_id(*self*, *channel*)
Callback called to retrieve a replay id. You should override this method to implement your custom logic.

Parameters **channel** (str) – Channel name

Return type Union[ReplayType, int]

coroutine ask_stop(*self*)
Ask client to stop receiving event:

```
async for event in client.events():
    ...
    if ...:
        await client.ask_stop()
```

This call will eventually stop *BaseSalesforceStreaming.messages()* and *BaseSalesforceStreaming.events()* async generator but this can take some time if not called inside the loop body: the generator will wait a timeout response from Salesforce server.

Return type None

coroutine unsubscribe(*self*, *channel*)
See *BaseSalesforceStreaming.unsubscribe()*

Return type List[Dict[str, Any]]

coroutine stop(*self*)
See *BaseSalesforceStreaming.stop()*

Return type None

class aio_sf_streaming.**SimpleRefreshTokenSalesforceStreaming**(*refresh_token*, *client_id*, *client_secret*, ***, *sandbox=False*, *version='42.0'*, *loop=None*, *connector=None*, *login_connector=None*, *retry_sub_duration=0.1*, *retry_factor=1.0*, *retry_max_duration=30.0*, *retry_max_count=20*)

A simple helper class providing all-in-one functionalities.

Parameters

- **refresh_token** (str) – Refresh token
- **client_id** (str) – OAuth2 client Id
- **client_secret** (str) – Oauth2 client secret
- **sandbox** (bool) – If True, the connexion will be made on a sandbox, from https://test.salesforce.com instead of the main login route at https://login.salesforce.com.
- **version** (str) – The API version to use. For example '42.0'.
- **loop** (Optional[AbstractEventLoop]) – Asyncio loop used
- **connector** (Optional[BaseConnector]) – aiohttp connector used for main session. Mainly used for test purpose.
- **login_connector** (Optional[BaseConnector]) – aiohttp connector used during connection. Mainly used for test purpose.
- **retry_sub_duration** (float) – Duration between subscribe retry if server is too buzy.
- **retry_factor** (float) – Factor amplification between each successive retry
- **retry_max_duration** (float) – Maximum value of the retry duration
- **retry_max_count** (int) – Maximum count of retry, after this count is reach, response or exception are propagated.

Usage example:

```
class MyClient(SimpleRefreshTokenSalesforceStreaming):
    def __init__(self):
        self.replays = []
        super().__init__(refresh_token='refresh_token',
                         client_id='my-client-id',
                         client_secret='my-client-secret')

    async def store_replay_id(self, channel, replay_id, creation_time):
        # We only store replay id without any use
        self.replays.append((channel, replay_id, creation_time))

    async def get_last_replay_id(self, channel):
        # We ask for only use new events
        return EventType.NEW_EVENTS
```

(continues on next page)

(continued from previous page)

```
async def print_events():
    async with MyClient() as client:
        await client.subscribe('/topic/Foo')
        async for message in client.events():
            channel = message['channel']
            print(f"Message received on {channel} : {message}")

loop = asyncio.get_event_loop()
loop.run_until_complete(print_event())
```

SimpleSalesforceStreaming inherit all members from their base class. Only main one, for external use, are listed here. See *SimpleRefreshTokenSalesforceStreaming* for method description.

2.1.3 Base class

class aio_sf_streaming.**BaseSalesforceStreaming**(*, *sandbox=False*, *version='42.0'*, *loop=None*, *connector=None*)

Base low-level *aio-sf-streaming* class.

Can not be used directly: must be sub-classed with at least one connector implementation. The class provide basic functionalities. Additional functionalities can be added with provided mixins.

The main logic is implemented here but you should not use it directly.

Parameters

- **sandbox** (bool) – If True, the connexion will be made on a sandbox, from https://test.salesforce.com instead of the main login route at https://login.salesforce.com.
- **version** (str) – The API version to use. For example '42.0'.
- **loop** (Optional[AbstractEventLoop]) – Asyncio loop used
- **connector** (Optional[BaseConnector]) – aiohttp connector used for main session. Mainly used for test purpose.

This class supports the context manager protocol for self closing.

All main members are coroutine, even if default implementation does do any asynchronous call. With this convention, sub classes and mixins can easily override this members and do complex call.

See *SimpleSalesforceStreaming* for an usage example.

High level api

coroutine start(*self*)

Connect to Salesforce, authenticate and init CometD connexion.

A best practice is to use async context manager interface that will call this method directly.

Return type None

coroutine subscribe(*self*, *channel*)

Subscribe to a channel. Can be used directly:

```
await client.subscribe('/topic/Foo')
```

This method, and the underlying protocol, are safe to be started as an background task:

```
loop.create_task(client.subscribe('/topic/Foo'))
```

Return type List[Dict[str, Any]]

async-for messages(*self*)

Asynchronous generator that fetch new messages and return one as soon as one is available:

```
async for message in client.messages():
    channel = message['channel']
    print(channel, ':', message)
```

This method iterate over **all** messages, even on internal/meta one. If you want to only iterate over messages from channels you subscribed, you should use *BaseSalesforceStreaming.events()*.

Warning: Linked to the underlying protocol, long-pooling based, the client should reconnect as soon as possible. Practically, client have 40 seconds to reconnect. If your processing take a longer time, a new connection should be made. You should avoid doing long processing between each iteration or launch this processing into a background task.

Return type Dict[str, Any]

async-for events(*self*)

Asynchronous generator that fetch new events and return one as soon as one is available:

```
async for message in client.events():
    channel = message['channel']
    print(channel, ':', message)
```

This method is different from *BaseSalesforceStreaming.messages()* because it filter messages and provide only those related to the channels you subscribed.

Return type Dict[str, Any]

coroutine ask_stop(*self*)

Ask client to stop receiving event:

```
async for event in client.events():
    ...
    if ...:
        await client.ask_stop()
```

This call will eventually stop *BaseSalesforceStreaming.messages()* and *BaseSalesforceStreaming.events()* async generator but this can take some time if not called inside the loop body: the generator will wait a timeout response from Salesforce server.

Return type None

coroutine unsubscribe(*self*, *channel*)

Unsubscribe to a channel. Can be used directly:

```
await client.unsubscribe('/topic/Foo')
```

This method, and the underlying protocol, are safe to be started as an background task:

```
loop.create_task(client.unsubscribe('/topic/Foo'))
```

Return type List[Dict[str, Any]]

coroutine stop(*self*)
Disconnect to Salesforce and close underlying connection.

A best practice is to use async context manager interface that will call this method directly.

Return type None

Connection logic

token_url
The url that should be used to fetch an access token.

Return type str

coroutine fetch_token(*self*)
Abstract coroutine method of connector that must provide an access token and the instance url linked.

Return type Tuple[str, str]

coroutine create_connected_session(*self*)
This coroutine create an aiohttp.ClientSession using fetched token

Return type ClientSession

coroutine close_session(*self*)
Close the underlying aiohttp.ClientSession connection

Return type None

Bayeux/CometD logic layer

end_point
Cometd endpoint

Return type str

coroutine get_handshake_payload(*self*)
Provide the handshake payload

Return type Dict[str, Any]

coroutine get_subscribe_payload(*self*, *channel*)
Provide the subscription payload for a specific channel

Return type Dict[str, Any]

coroutine get_unsubscribe_payload(*self*, *channel*)
Provide the unsubscription payload for a specific channel

Return type Dict[str, Any]

coroutine send(*self*, *data*)
Send data to CometD server when the connection is established:

```
# Manually disconnect
await client.send({'channel': '/meta/disconnect'})
```

Return type Union[Dict[str, Any], List[Dict[str, Any]]]

coroutine handshake(*self*)
Coroutine that perform an handshake (mandatory before any other action)

Return type List[Dict[str, Any]]

coroutine disconnect(*self*)
Disconnect from the SF streaming server

Return type List[Dict[str, Any]]

I/O layer helpers

coroutine get(*self*, *sub_url*, ***kwargs*)
Perform a simple json get request from an internal url:

```
response = await.client.get('/myendpoint/')
```

Return type Union[Dict[str, Any], List[Dict[str, Any]]]

coroutine post(*self*, *sub_url*, ***kwargs*)
Perform a simple json post request from an internal url:

```
response = await.client.post('/myendpoint/', json={'data': 'foo'})
```

Return type Union[Dict[str, Any], List[Dict[str, Any]]]

coroutine request(*self*, *method*, *sub_url*, ***kwargs*)
Perform a simple json request from an internal url

Return type Union[Dict[str, Any], List[Dict[str, Any]]]

Other attributes

loop
Running event loop

Return type AbstractEventLoop

2.1.4 Connectors

class aio_sf_streaming.**BaseConnector**(*, *client_id=None*, *client_secret=None*, *login_connector=None*, ***kwargs*)
Base class for all sf connectors.

Parameters

- **client_id** (Optional[str]) – OAuth2 client Id (mandatory)
- **client_secret** (Optional[str]) – Oauth2 client secret (mandatory)
- **login_connector** (Optional[BaseConnector]) – aiohttp connector used during connection. Mainly used for test purpose.

See *BaseSalesforceStreaming* for other keywords arguments.

class aio_sf_streaming.**PasswordSalesforceStreaming**(*, *username=None*, *password=None*, ***kwargs*)
Create a SF streaming manager with password flow connection.

Main arguments are connection credentials:

Parameters

- **username** (Optional[str]) – User login name
- **password** (Optional[str]) – User password

See *BaseConnector* for other keywords arguments.

class aio_sf_streaming.**RefreshTokenSalesforceStreaming**(*, *refresh_token=None*, ***kwargs*)

Create a SF streaming manager with password refresh token connection.

Main arguments are connection credentials:

Parameters **refresh_token** (Optional[str]) – Refresh token

See *BaseConnector* for other keywords arguments.

2.1.5 Mixins

class aio_sf_streaming.**AllMixin**(**args*, ***kwargs*)

Helper class to add all mixin with one class

class aio_sf_streaming.**TimeoutAdviceMixin**

Simple mixin that automatically set timeout setting according to SF advice, if provided.

class aio_sf_streaming.**ReplayType**(*value*)

Enumeration with special replay values

ALL_EVENTS = -2

Replay all events available.

NEW_EVENTS = -1

No replay, retrieve only new events.

class aio_sf_streaming.**ReplayMixin**

Mixing adding replay support to the streaming client.

This mixin is not enough, you must implement *ReplayMixin.store_replay_id()* and *:py:func:`ReplayMixin.get_last_replay_id* in a subclass in order to have a working replay.

coroutine get_last_replay_id(*self*, *channel*)

Callback called to retrieve a replay id. You should override this method to implement your custom logic.

Parameters **channel** (str) – Channel name

Return type Union[ReplayType, int]

coroutine store_replay_id(*self*, *channel*, *replay_id*, *creation_time*)

Callback called to store a replay id. You should override this method to implement your custom logic.

Parameters

- **channel** (str) – Channel name
- **replay_id** (int) – replay id to store
- **creation_time** (str) – Creation time. You should store only the last created object but you can not know if you received event in order without this. This value is the string provided by SF.

Return type None

class aio_sf_streaming.**AutoVersionMixin**

Simple mixin that fetch last api version before connect.

class aio_sf_streaming.**AutoReconnectMixin**(**args*, ***kwargs*)
Mixin that will automatically reconnect when asked by Salesforce

class aio_sf_streaming.**ReSubscribeMixin**(*retry_sub_duration=0.1*, *retry_factor=1.0*, *retry_max_duration=30.0*, *retry_max_count=20*, ***kwargs*)
Mixin that handle subscription error, will try again after a short delay

Parameters

- **retry_sub_duration** (float) – Duration between subscribe retry if server is too buzy (initial value).
- **retry_factor** (float) – Factor amplification between each successive retry
- **retry_max_duration** (float) – Maximum value of the retry duration
- **retry_max_count** (int) – Maximum count of retry, after this count is reach, response or exception are propagated.

coroutine should_retry_on_error_response(*self*, *channel*, *response*)
Callback called to process a response with and error message. Return a boolean if we must retry. If False is returned, the response will be returned to caller.

By-default, retry on known 'server unavailable' response.

Parameters

- **channel** (str) – Channel name
- **response** (Dict[str, Any]) – The response received

Return type bool

coroutine should_retry_on_exception(*self*, *channel*, *exception*)
Callback called to process an exception raised during subscription. Return a boolean if we must retry. If False is returned, the exception will be propagated to caller.

By-default, do return always False.

Parameters

- **channel** (str) – Channel name
- **exception** (Exception) – The exception raised

Return type bool

2.2 Release notes

v. **0.3.0**

- Remove date-time parsing of the replay mixin.
- Allow client to customize ReSubscribeMixin retry conditions.
- Allow client to customize ReSubscribeMixin retry duration evolution.

v. **0.2.0** Minor release : Add refresh token authentication

v. **0.1.1** Minor release : Add documentation and start adding typing annotations.

v. **0.1.0** Initial release : Basic Salesforce streaming client with password flow support.

PYTHON MODULE INDEX

a

INDEX

R

S

T

U