aiosfstream Documentation

Release 0.5.0

Robert Marki

Mar 08, 2019

Contents

aiosfstream is a Salesforce Streaming API client for asyncio. It can be used to receive push notifications about changes on Salesforce objects or notifications of general events sent through the Streaming API.

For detailed guidance on how to work with PushTopics or how to create Generic Streaming Channels please consult the Streaming API documentation. For working with Platform Events or Change Data Capture events check out the linked documentation.

CHAPTER 1

Features

- **Supported authentication types:**
  - using a username and password
  - using a refresh token
- **Subscribe to and receive messages on:**
  - PushTopics
  - Generic Streaming Channels
  - Platform Events
  - Change Data Capture events
- Support for durable messages and replay of events
- Automatic recovery from replay errors

CHAPTER 2

Usage

```python
import asyncio

from aiosfstream import SalesforceStreamingClient


async def stream_events():
    # connect to Streaming API
    async with SalesforceStreamingClient(
            consumer_key="<consumer key>",
            consumer_secret="<consumer secret>",
            username="<username>",
            password="<password>") as client:

        # subscribe to topics
        await client.subscribe("/topic/one")
        await client.subscribe("/topic/two")

        # listen for incoming messages
        async for message in client:
            topic = message["channel"]
            data = message["data"]
            print(f"{topic}: {data}")

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(stream_events())
```

CHAPTER 3

Documentation

http://aiosfstream.readthedocs.io/

CHAPTER 4

Contents

4.1 User's guide

4.1.1 Installation

```
pip install aiosfstream
```

Install extras

aiosfstream defines several groups of optional requirements:

- `tests` for running unit tests
- `docs` for building the documentation
- `dev` for creating a complete development enviroment

Any combination of these options can be specified during installation.

```
pip install aiosfstream[tests,docs,dev]
```

4.1.2 Quickstart

Authentication

To connect to the Salesforce Streaming API all clients must authenticate themselves. The library supports the username-password based OAuth2 authentication flow as well as the refresh token based authentication.

Whichever technique you end up using, you must first create a Connected App on Salesforce to acquire a Consumer Key and Consumer Secret value. Which are actually the client_id and client_secret parameters in OAuth2 terminology.

Username-Password authentication

For username-password based authentication you can use the *SalesforceStreamingClient* class, with the Salesforce user's username and password:

```
client = SalesforceStreamingClient(
    consumer_key="<consumer key>",
    consumer_secret="<consumer secret>",
    username="<username>",
    password="<password>"
)
```

SalesforceStreamingClient is actually just a convenience class, based on *Client*. It enables you to create a client object with the most common authentication technique, without having to create a separate *PasswordAuthenticator* object. You can actually use the *Client* class to create client that would be equivalent with the example above:

```
auth = PasswordAuthenticator(
    consumer_key="<consumer key>",
    consumer_secret="<consumer secret>",
    username="<username>",
    password="<password>"
)
client = Client(auth)
```

Refresh token authentication

The refresh token base authentication technique can be used by creating a *RefreshTokenAuthenticator* and passing it to the *Client* class:

```
auth = RefreshTokenAuthenticator(
    consumer_key="<consumer key>",
    consumer_secret="<consumer secret>",
    refresh_token="<refresh_token>"
)
client = Client(auth)
```

You can get a refresh token using several different authentication techniques supported by Salesforce, the most commonly used one is probably the web server authentication flow.

Authentication on sandbox orgs

If you're trying to connect to a sandbox org, then you have to assign `True` to the `sandbox` parameter when creating the *SalesforceStreamingClient*, *PasswordAuthenticator* or *RefreshTokenAuthenticator* object. Furthermore, the name of the sandbox should be appended to the username. For example, if a username for a production org is user1@acme.com, and the sandbox is named *test*, the modified username to log in to the sandbox is user1@acme.com.test.

```
client = SalesforceStreamingClient(
    consumer_key="<consumer key>",
    consumer_secret="<consumer secret>",
    username="<username>.<sandbox_name>",
    password="<password>",
```

(continues on next page)

(continued from previous page)

```
    sandbox=True
)
```

Connecting

After creating a *Client* object the *open()* method should be called to establish a connection with the server. The connection is closed and the session is terminated by calling the *close()* method.

```
client = SalesforceStreamingClient(
    consumer_key="<consumer key>",
    consumer_secret="<consumer secret>",
    username="<username>",
    password="<password>"
)
await client.open()
# subscribe and receive messsages...
await client.close()
```

Client objects can be also used as asynchronous context managers.

```
async with SalesforceStreamingClient(
        consumer_key="<consumer key>",
        consumer_secret="<consumer secret>",
        username="<username>",
        password="<password>") as client:
    # subscribe and receive messsages...
```

Channels

A channel is a string that looks like a URL path such as `/topic/foo` or `/topic/bar`.

For detailed guidance on how to work with PushTopics or how to create Generic Streaming Channels please consult the Streaming API documentation. For working with Platform Events or Change Data Capture events check out the linked documentation.

Subscriptions

To receive notification messages the client must subscribe to the channels it's interested in.

```
await client.subscribe("/topic/foo")
```

If you no longer want to receive messages from one of the channels you're subscribed to then you must unsubscribe from the channel.

```
await client.unsubscribe("/topic/foo")
```

The current set of subscriptions can be obtained from the *Client.subscriptions* attribute.

Receiving messages

To receive messages broadcasted by Salesforce after *subscribing* to these *channels* the *receive()* method should be used.

```
message = await client.receive()
```

The *receive()* method will wait until a message is received or it will raise a *TransportTimeoutError* in case the connection is lost with the server and the client can't re-establish the connection or a *ServerError* if the connection gets closed by the server.

The client can also be used as an asynchronous iterator in a for loop to wait for incoming messages.

```
async for message in client:
    # process message
```

Replay of events

The great thing about streaming is that the client gets instantly notified about events as they occur. The downside is that if the client becomes temporarily disconnected, due to hardware, software or network failure, then it might miss some of the messages emitted by the server. This is where Salesforce's message durability comes in handy.

Salesforce stores events for 24 hours. Events outside the 24-hour retention period are discarded. Salesforce extends the event messages with `replayId` and `createdDate` fields (called as *ReplayMarker* by aiosfstream). These fields can be used by the client to request the missed event messages from the server when it reconnects.

The default behavior of the client is to receive only the new events sent after subscribing. To take advantage of message durability, all you have to do is to pass an object capable of storing the most recent *ReplayMarker* objects, so the next time the client reconnects, it can continue to process event messages from the point where it left off. The most convenient choice is a `Shelf` object, which can store *ReplayMarkers* on the disk, between application restarts.

```
with shelve.open("replay.db") as replay:

    async with SalesforceStreamingClient(
        consumer_key="<consumer key>",
        consumer_secret="<consumer secret>",
        username="<username>",
        password="<password>",
        replay=replay) as client:

        await client.subscribe("/topic/foo")

        async for message in client:
            # process message
```

Besides `Shelf` objects you can pass a lot of different kind of objects to the replay parameter, and you can configure different aspects of replay behavior as well. For a full description of replay configuration options check out the *Replay configuration* section.

4.1.3 Advanced Usage

Replay configuration

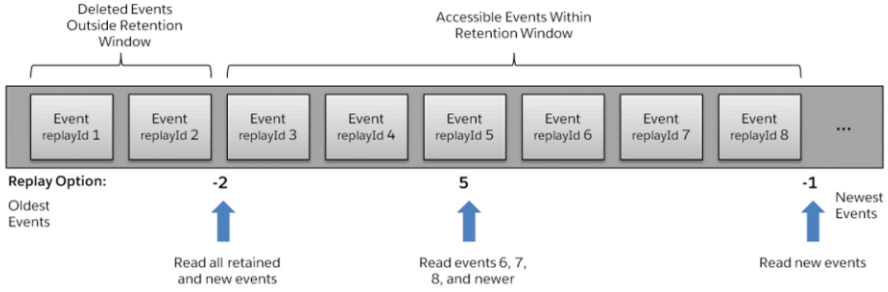
Salesforce stores events for 24 hours. Events outside the 24-hour retention period are discarded.

ReplayOption

A subscriber can choose which events to receive, such as all events within the retention window or starting after a particular event. In the *Client* object this can be specified with the `replay` parameter. The default is to receive

only the new events sent after subscribing, the default `replay` parameter is *`ReplayOption.NEW_EVENTS`*

This high-level diagram shows how event consumers can read a stream of events by using various replay options.



If you want to receive all events within the retention window every time the *`Client`* connects, before receiving new events, then the *`ReplayOption.ALL_EVENTS`* value should be passed to the *`Client`*.

```
async with SalesforceStreamingClient(
                consumer_key="<consumer key>",
                consumer_secret="<consumer secret>",
                username="<username>",
                password="<password>",
                replay=ReplayOption.ALL_EVENTS) as client:

    await client.subscribe("/topic/foo")

            async for message in client:
                # process message
```

ReplayMarkerStorage

Although using a fixed *`ReplayOption`* can be sometimes useful, the real advantage of using Salesforce's replay extension comes from being able to continue to process event messages from the point where the client left off. To take advantage of this feature, all you have to do is to pass an object capable of storing the most recent *`ReplayMarker`* for every channel.

Salesforce extends the event messages with `replayId` and `createdDate` fields (called as *`ReplayMarker`* by aiosfstream).

The simplest way is to pass an object for the `replay` parameter that inherits from `collections.abc.MutableMapping`. This can be a simple `dict`, `OrderedDict` or if you want to use persistent storage then a `Shelf` object, or maybe one of the key-value database drivers that inherit from `collections.abc.MutableMapping`.

```
with shelve.open("replay.db") as replay:

    async with SalesforceStreamingClient(
        consumer_key="<consumer key>",
        consumer_secret="<consumer secret>",
        username="<username>",
        password="<password>",
        replay=replay) as client:

        await client.subscribe("/topic/foo")
```

(continues on next page)

(continued from previous page)

```
        async for message in client:
            # process message
```

By using a `collections.abc.MutableMapping` object, the client on the first connection will receive only new events, and on reconnection will continue from the last unretrieved message. If you want to receive all events from the retention window before continuing with new events, combined with the advantage of continuation on the next reconnect, then you can pass a *DefaultMappingStorage* object to the `replay` parameter.

```
with shelve.open("replay.db") as replay:

    default_mapping = DefaultMappingStorage(
        replay,
        ReplayOption.ALL_EVENTS
    )

    async with SalesforceStreamingClient(
        consumer_key="<consumer key>",
        consumer_secret="<consumer secret>",
        username="<username>",
        password="<password>",
        replay=default_mapping) as client:

        await client.subscribe("/topic/foo")

        async for message in client:
            # process message
```

If you want complete control over how *ReplayMarkers* are stored and retrieved or you want to use your favorite database whose driver doesn't inherit from `collections.abc.MutableMapping` then you can provide your own *ReplayMarkerStorage* implementation.

```
class MyReplayMarkerStorage(ReplayMarkerStorage):
    async def set_replay_marker(self, subscription, replay_marker):
        # store *replay_marker* for the given *subscription*

    async def get_replay_marker(self, subscription):
        # retrieve the replay marker for the given *subscription*

replay = MyReplayMarkerStorage()

async with SalesforceStreamingClient(
    consumer_key="<consumer key>",
    consumer_secret="<consumer secret>",
    username="<username>",
    password="<password>",
    replay=replay) as client:

    await client.subscribe("/topic/foo")

    async for message in client:
        # process message
```

Subscription errors

Events outside the 24-hour retention period are discarded. If you're using some form of *ReplayMarkerStorage* or a MutableMapping object, and if you're client doesn't connects to the Streaming API for more then 24 hours, then it's possible that the client will try to continue retrieving messages from a very old message outside the retention window. Since Salesforce no longer has the event message that the client would try to retrieve, it would raise *ServerError*.

```
try:
    await client.subscribe("/topic/foo")
except ServerError as error:
    print(error.error_message)
```

The above code would print the following message, if the client would request and event outside the retention window:

```
The replayId {1} you provided was invalid.  Please provide a valid ID, -2
to replay all events, or -1 to replay only new events.
```

To recover from an error like the above, you would have to discard the *ReplayMarker* for the problematic channel, and try to subscribe again.

```
try:
    await client.subscribe("/topic/foo")
except ServerError as error:
    del replay["/topic/foo"]
    await client.subscribe(/topic/foo")
```

To spare you the hassle of recovering from errors like the one above, you can pass a *ReplayOption* for the replay_fallback parameter. If a subscription error occurs, then *Client* will try to resubscribe using the specified *ReplayOption*.

```
with shelve.open("replay.db") as replay:

    async with SalesforceStreamingClient(
        consumer_key="<consumer key>",
        consumer_secret="<consumer secret>",
        username="<username>",
        password="<password>",
        replay=replay,
        replay_fallback=ReplayOption.ALL_EVENTS) as client:

        await client.subscribe("/topic/foo")

        async for message in client:
            # process message
```

ReplayMarkerStoragePolicy

If you're using some form of ReplayMarkerStorage or a MutableMapping object, the *ReplayMarker* values for messages are stored just before they're returned by the *receive()* method, or yielded from the asynchronous iterator of the *Client* object if used as an iterable.

This behaviour might not be optimal in situations when there is a high probability that the processing of the message might fail on the side of the user of the *Client*. If the message processing code would raise an exception, the message wouldn't be processed completely, but since the message's *ReplayMarker* is already stored at that point, the failed message wouldn't be re-retrieved the next time the *Client* is opened.

In order to be able to retrieve unsuccessfully processed messages, users can take control at which point the *ReplayMarkers* are stored. Pass the *ReplayMarkerStoragePolicy.MANUAL* option to the `replay_storage_policy` parameter when creating the *Client* and call the *extract_replay_id()* method of *Client.replay_storage* once the message is successfully processed.

```
with shelve.open("replay.db") as replay:

        async with SalesforceStreamingClient(
            consumer_key="<consumer key>",
            consumer_secret="<consumer secret>",
            username="<username>",
            password="<password>",
            replay=replay,
            replay_storage_policy=ReplayMarkerStoragePolicy.MANUAL) as client:

            await client.subscribe("/topic/foo")

            async for message in client:
                # message processing code raising errors...
                await client.replay_storage.extract_replay_id(message)
```

The *Client.replay_storage* attribute can also be used as an asynchronous context manager, which will only store the *ReplayMarker* of the given message if no errors are raised in the runtime context.

```
with shelve.open("replay.db") as replay:

        async with SalesforceStreamingClient(
            consumer_key="<consumer key>",
            consumer_secret="<consumer secret>",
            username="<username>",
            password="<password>",
            replay=replay,
            replay_storage_policy=ReplayMarkerStoragePolicy.MANUAL) as␣
↪client:

            await client.subscribe("/topic/foo")

            async for message in client:
                async with client.replay_storage(message):
                    # message processing code raising errors...
```

Network failures

When a *Client* object is opened, it will try to maintain a continuous connection in the background with the server. If any network failures happen while waiting to *receive()* messages, the client will reconnect to the server transparently, it will resubscribe to the subscribed channels, and continue to wait for incoming messages.

To avoid waiting for a server which went offline permanently, or in case of a permanent network failure, a `connection_timeout` can be passed to the *Client*, to limit how many seconds the client object should wait before raising a *TransportTimeoutError* if it can't reconnect to the server.

```
client = SalesforceStreamingClient(
    consumer_key="<consumer key>",
    consumer_secret="<consumer secret>",
    username="<username>",
    password="<password>",
```

(continues on next page)

(continued from previous page)

```
    connection_timeout=60
)
await client.open()

try:
    message = await client.receive()
except TransportTimeoutError:
    print("Connection is lost with the server. "
          "Couldn't reconnect in 60 seconds.")
```

The defaul value is `10` seconds. If you pass `None` as the `connection_timeout` value, then the client will keep on trying indefinitely.

Prefetching

When a *`Client`* is opened it will start and maintain a connection in the background with the server. It will start to fetch messages from the server as soon as it's connected, even before *`receive()`* is called.

Prefetching messages has the advantage, that incoming messages will wait in a buffer for users to consume them when *`receive()`* is called, without any delay.

To avoid consuming all the available memory by the incoming messages, which are not consumed yet, the number of prefetched messages can be limited with the `max_pending_count` parameter of the *`Client`*. The default value is `100`.

```
client = SalesforceStreamingClient(
    consumer_key="<consumer key>",
    consumer_secret="<consumer secret>",
    username="<username>",
    password="<password>",
    max_pending_count=42
)
```

The current number of messages waiting to be consumed can be obtained from the *`Client.pending_count`* attribute.

JSON encoder/decoder

Besides the standard `json` module, many third party libraries offer JSON serialization/deserilization functionality. To use a different library for handling JSON data types, you can specify the callable to use for serialization with the `json_dumps` and the callable for deserialization with the `json_loads` parameters of the *`Client`*.

```
import ujson

client = SalesforceStreamingClient(
    consumer_key="<consumer key>",
    consumer_secret="<consumer secret>",
    username="<username>",
    password="<password>",
    json_dumps=ujson.dumps,
    json_loads=ujson.loads
)
```

4.2 API Reference

4.2.1 Client

class aiosfstream.**SalesforceStreamingClient**(*, *consumer_key*, *consumer_secret*, *username*, *password*, *replay=<ReplayOption.NEW_EVENTS: -1>*, *replay_fallback=None*, *replay_storage_policy=<ReplayMarkerStoragePolicy.AUTOMATIC: 1>*, *connection_timeout=10.0*, *max_pending_count=100*, *sandbox=False*, *json_dumps=<function dumps>*, *json_loads=<function loads>*, *loop=None*)

Salesforce Streaming API client with username/password authentication

This is a convenience class which is suitable for the most common use case. To use a different authentication method, use the general *Client* class with a different *Authenticator*

Parameters

- **consumer_key** (str) – Consumer key from the Salesforce connected app definition
- **consumer_secret** (str) – Consumer secret from the Salesforce connected app definition
- **username** (str) – Salesforce username
- **password** (str) – Salesforce password
- **replay** (Union[ReplayOption, ReplayMarkerStorage, Mutablemapping[str, ReplayMarker]]) – A ReplayOption or an object capable of storing replay ids if you want to take advantage of Salesforce's replay extension. You can use one of the *ReplayOptions*, or an object that supports the MutableMapping protocol like dict, defaultdict, Shelf etc. or a custom *ReplayMarkerStorage* implementation.
- **replay_fallback** (Optional[ReplayOption]) – Replay fallback policy, for when a subscribe operation fails because a replay id was specified for a message outside the retention window
- **replay_storage_policy** (ReplayMarkerStoragePolicy) – Defines at which point the replay marker of received messages will be stored
- **connection_timeout** (Union[int, float]) – The maximum amount of time to wait for the transport to re-establish a connection with the server when the connection fails.
- **max_pending_count** (int) – The maximum number of messages to prefetch from the server. If the number of prefetched messages reach this size then the connection will be suspended, until messages are consumed. If it is less than or equal to zero, the count is infinite.
- **sandbox** (bool) – Marks whether the connection has to be made with a sandbox org or with a production org
- **json_dumps** (Callable[[Dict[str, Any]], str]) – Function for JSON serialization, the default is json.dumps()
- **json_loads** (Callable[[str], Dict[str, Any]]) – Function for JSON deserialization, the default is json.loads()

- **loop** (Optional[AbstractEventLoop]) – Event `loop` used to schedule tasks. If *loop* is `None` then `asyncio.get_event_loop()` is used to get the default event loop.

class aiosfstream.**Client**(*authenticator*, ***, *replay=<ReplayOption.NEW_EVENTS: -1>*, *replay_fallback=None*, *replay_storage_policy=<ReplayMarkerStoragePolicy.AUTOMATIC: 1>*, *connection_timeout=10.0*, *max_pending_count=100*, *json_dumps=<function dumps>*, *json_loads=<function loads>*, *loop=None*)

Salesforce Streaming API client

Parameters

- **authenticator** (*AuthenticatorBase*) – An authenticator object
- **replay** (Union[ReplayOption, ReplayMarkerStorage, Mutablemapping[str, ReplayMarker]]) – A ReplayOption or an object capable of storing replay ids if you want to take advantage of Salesforce's replay extension. You can use one of the *ReplayOptions*, or an object that supports the MutableMapping protocol like `dict`, `defaultdict`, `Shelf` etc. or a custom *ReplayMarkerStorage* implementation.
- **replay_fallback** (Optional[ReplayOption]) – Replay fallback policy, for when a subscribe operation fails because a replay id was specified for a message outside the retention window
- **replay_storage_policy** (ReplayMarkerStoragePolicy) – Defines at which point the replay marker of received messages will be stored
- **connection_timeout** (Union[int, float]) – The maximum amount of time to wait for the transport to re-establish a connection with the server when the connection fails.
- **max_pending_count** (int) – The maximum number of messages to prefetch from the server. If the number of prefetched messages reach this size then the connection will be suspended, until messages are consumed. If it is less than or equal to zero, the count is infinite.
- **json_dumps** (Callable[[Dict[str, Any]], str]) – Function for JSON serialization, the default is `json.dumps()`
- **json_loads** (Callable[[str], Dict[str, Any]]) – Function for JSON deserialization, the default is `json.loads()`
- **loop** (Optional[AbstractEventLoop]) – Event `loop` used to schedule tasks. If *loop* is `None` then `asyncio.get_event_loop()` is used to get the default event loop.

coroutine open(*self*)

Establish a connection with the Streaming API endpoint

Raises

- ***ClientError*** – If none of the connection types offered by the server are supported
- ***ClientInvalidOperation*** – If the client is already open, or in other words if it isn't *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the handshake or the first connect request gets rejected by the server.
- ***AuthenticationError*** – If the server rejects the authentication request or if a network failure occurs during the authentication

Return type `None`

coroutine close(*self*)
Disconnect from the CometD server

Return type None

coroutine publish(*self*, *channel*, *data*)
Publish *data* to the given *channel*

> **Warning:** The Streaming API is implemented on top of CometD. The publish operation is a CometD operation. While it's still a legal operation, Salesforce chose not to implement the publishing of Generic Streaming and Platform events with CometD.
>
> You should use the REST API to generate Generic Streaming events, or use the REST or SOAP API to publish Platform events.

Parameters

- **channel** (str) – Name of the channel
- **data** (Dict[str, Any]) – Data to send to the server

Return type Dict[str, Any]

Returns Publish response

Raises

- ***ClientInvalidOperation*** – If the client is *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the publish request gets rejected by the server

coroutine subscribe(*self*, *channel*)
Subscribe to *channel*

Parameters **channel** (str) – Name of the channel

Raises

- ***ClientInvalidOperation*** – If the client is *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the subscribe request gets rejected by the server

Return type None

coroutine unsubscribe(*self*, *channel*)
Unsubscribe from *channel*

Parameters **channel** (str) – Name of the channel

Raises

- ***ClientInvalidOperation*** – If the client is *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the unsubscribe request gets rejected by the server

Return type None

coroutine receive(*self*)
Wait for incoming messages from the server

Return type Dict[str, Any]

Returns Incoming message

Raises

- ***ClientInvalidOperation*** – If the client is closed, and has no more pending incoming messages
- ***ServerError*** – If the client receives a confirmation message which is not `successful`
- ***TransportTimeoutError*** – If the transport can't re-establish connection with the server in `connection_timeout` time.
- ***ReplayError*** – On a message replay or replay marker storage related error

closed
Marks whether the client is open or closed

Return type bool

subscriptions
Set of subscribed channels

Return type Set[str]

connection_type
The current connection type in use if the client is open, otherwise `None`

Return type Optional[ConnectionType]

pending_count
The number of pending incoming messages

Once *open* is called the client starts listening for messages from the server. The incoming messages are retrieved and stored in an internal queue until they get consumed by calling *receive*.

Return type int

has_pending_messages
Marks whether the client has any pending incoming messages

Return type bool

coroutine close(*self*)
Disconnect from the CometD server

Return type None

static create_replay_storage(*replay_param*)
Create a *ReplayMarkerStorage* object based from *replay_param*

Parameters **replay_param** (Union[ReplayOption, ReplayMarkerStorage, Mutablemapping[str, ReplayMarker]]) – One of the supported *replay_param* type objects

Return type Optional[ReplayMarkerStorage]

Returns A new *ReplayMarkerStorage* object or *replay_param* if it's already an instance of *ReplayMarkerStorage* object, or None if *replay_param* is None

static get_cometd_url(*instance_url*)
Get the CometD URL associated with the *instance_url*

Parameters **instance_url** (str) – Salesforce instance URL

Return type str

Returns CometD URL associated with the *instance_url*

coroutine open(*self*)
Establish a connection with the Streaming API endpoint

Raises

- ***ClientError*** – If none of the connection types offered by the server are supported
- ***ClientInvalidOperation*** – If the client is already open, or in other words if it isn't *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the handshake or the first connect request gets rejected by the server.
- ***AuthenticationError*** – If the server rejects the authentication request or if a network failure occurs during the authentication

Return type None

coroutine publish(*self*, *channel*, *data*)
Publish *data* to the given *channel*

> **Warning:** The Streaming API is implemented on top of CometD. The publish operation is a CometD operation. While it's still a legal operation, Salesforce chose not to implement the publishing of Generic Streaming and Platform events with CometD.
>
> You should use the REST API to generate Generic Streaming events, or use the REST or SOAP API to publish Platform events.

Parameters

- **channel** (str) – Name of the channel
- **data** (Dict[str, Any]) – Data to send to the server

Return type Dict[str, Any]

Returns Publish response

Raises

- ***ClientInvalidOperation*** – If the client is *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the publish request gets rejected by the server

coroutine receive(*self*)
Wait for incoming messages from the server

Return type Dict[str, Any]

Returns Incoming message

Raises

- ***ClientInvalidOperation*** – If the client is closed, and has no more pending incoming messages

- ***ServerError*** – If the client receives a confirmation message which is not `successful`
- ***TransportTimeoutError*** – If the transport can't re-establish connection with the server in `connection_timeout` time.
- ***ReplayError*** – On a message replay or replay marker storage related error

replay_fallback = None
Replay fallback policy, for when a subscribe operation fails because a replay id was specified for a message outside the retention window

replay_storage = None
ReplayMarkerStorage instance capable of storing *ReplayMarker* objects

replay_storage_policy = None
Defines at which point the *ReplayMarker* of received messages will be stored

coroutine subscribe(*self*, *channel*)
Subscribe to *channel*

Parameters **channel** (`str`) – Name of the channel

Raises

- ***ClientInvalidOperation*** – If the client is *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the subscribe request gets rejected by the server

Return type `None`

coroutine unsubscribe(*self*, *channel*)
Unsubscribe from *channel*

Parameters **channel** (`str`) – Name of the channel

Raises

- ***ClientInvalidOperation*** – If the client is *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the unsubscribe request gets rejected by the server

Return type `None`

class aiosfstream.**ReplayMarkerStoragePolicy**
Defines the available replay marker storage policies

AUTOMATIC = 1
Store the replay marker of messages automatically, as soon as they're received. The downside of this approach is that the replay marker of a message will be stored (thus marking it as successfully consumed) even if the processing of the message fails in the client side code

MANUAL = 2
Store the replay marker of messages manually, after the message has been successfully processed in client side code

4.2.2 Authenticators

class aiosfstream.auth.**AuthenticatorBase**(*sandbox=False*, *json_dumps=<function dumps>*, *json_loads=<function loads>*)

Abstract base class to serve as a base for implementing concrete authenticators

Parameters

- **sandbox** (bool) – Marks whether the authentication has to be done for a sandbox org or for a production org
- **json_dumps** (Callable[[Dict[str, Any]], str]) – Function for JSON serialization, the default is json.dumps()
- **json_loads** (Callable[[str], Dict[str, Any]]) – Function for JSON deserialization, the default is json.loads()

class aiosfstream.**PasswordAuthenticator**(*consumer_key*, *consumer_secret*, *username*, *password*, *sandbox=False*, *json_dumps=<function dumps>*, *json_loads=<function loads>*)

Authenticator for using the OAuth 2.0 Username-Password Flow

Parameters

- **consumer_key** (str) – Consumer key from the Salesforce connected app definition
- **consumer_secret** (str) – Consumer secret from the Salesforce connected app definition
- **username** (str) – Salesforce username
- **password** (str) – Salesforce password
- **sandbox** (bool) – Marks whether the authentication has to be done for a sandbox org or for a production org
- **json_dumps** (Callable[[Dict[str, Any]], str]) – Function for JSON serialization, the default is json.dumps()
- **json_loads** (Callable[[str], Dict[str, Any]]) – Function for JSON deserialization, the default is json.loads()

client_id = None
OAuth2 client id

client_secret = None
OAuth2 client secret

password = None
Salesforce password

username = None
Salesforce username

class aiosfstream.**RefreshTokenAuthenticator**(*consumer_key*, *consumer_secret*, *refresh_token*, *sandbox=False*, *json_dumps=<function dumps>*, *json_loads=<function loads>*)

Authenticator for using the OAuth 2.0 Refresh Token Flow

Parameters

- **consumer_key** (str) – Consumer key from the Salesforce connected app definition

- **consumer_secret** (str) – Consumer secret from the Salesforce connected app definition
- **refresh_token** (str) – A refresh token obtained from Salesforce by using one of its authentication methods (for example with the OAuth 2.0 Web Server Authentication Flow)
- **sandbox** (bool) – Marks whether the authentication has to be done for a sandbox org or for a production org
- **json_dumps** (Callable[[Dict[str, Any]], str]) – Function for JSON serialization, the default is `json.dumps()`
- **json_loads** (Callable[[str], Dict[str, Any]]) – Function for JSON deserialization, the default is `json.loads()`

client_id = None
OAuth2 client id

client_secret = None
OAuth2 client secret

refresh_token = None
Salesforce refresh token

4.2.3 Replay

class aiosfstream.**ReplayOption**
Replay options supported by Salesforce

ALL_EVENTS = -2
Receive all events, including past events that are within the 24-hour retention window and new events sent after subscription

NEW_EVENTS = -1
Receive new events that are broadcast after the client subscribes

class aiosfstream.**ReplayMarker**
Bases: tuple

Class for storing a message replay id and its creation date

Create new instance of ReplayMarker(date, replay_id)

date
Creation date of a message, as a ISO 8601 formatted datetime string or a unix timestamp as a string

replay_id
Replay id of a message

class aiosfstream.**ReplayMarkerStorage**
Abstract base class for replay marker storage implementations

coroutine get_replay_marker(*self*, *subscription*)
Retrieve a stored replay marker for the given *subscription*

Parameters **subscription** (str) – Name of the subscribed channel

Return type Optional[ReplayMarker]

Returns A replay marker or None if there is nothing stored for the given *subscription*

coroutine set_replay_marker(*self*, *subscription*, *replay_marker*)
Store the *replay_marker* for the given *subscription*

Parameters

- **subscription** (str) – Name of the subscribed channel
- **replay_marker** (ReplayMarker) – A replay marker

Return type None

coroutine extract_replay_id(*self*, *message*)
Extract and store the replay id present int the *message*

Parameters **message** (Dict[str, Any]) – An incoming broadcast message

Raises ***ReplayError*** – If no creation date can be found in the *message*

Return type None

coroutine __call__(*self*, *message*)
Return an asynchronous context manager instance for extracting the replay id from the *message* if no exceptions occur inside the runtime context

Parameters **message** (Dict[str, Any]) – An incoming message

Return type Asynccontextmanager[None]

Returns An asynchronous context manager

class aiosfstream.**MappingStorage**(*mapping*)
Mapping based replay marker storage

Parameters **mapping** (Mutablemapping[str, ReplayMarker]) – A MutableMapping object for storing replay markers

class aiosfstream.**DefaultMappingStorage**(*mapping*, *default_id*)
Mapping based replay marker storage which will return a defualt replay id if there is not replay marker for the given subscription

Parameters

- **mapping** (Mutablemapping[str, ReplayMarker]) – A MutableMapping object for storing replay markers
- **default_id** (int) – A replay id

class aiosfstream.**ConstantReplayId**(*default_id*, ***kwargs*)
A replay marker storage which will return a constant replay id for every subscription

Note: This implementations doesn't actually stores anything for later retrieval.

Parameters **default_id** (int) – A replay id

4.2.4 Exceptions

Exception types

Exception hierarchy:

```
AiosfstreamException
    AuthenticationError
    ClientError
        ClientInvalidOperation
```

(continues on next page)

(continued from previous page)

```
    TransportError
        TransportInvalidOperation
        TransportTimeoutError
        TransportConnectionClosed
    ServerError
    ReplayError
```

exception aiosfstream.exceptions.**AiosfstreamException**
Base exception type.

All exceptions of the package inherit from this class.

exception aiosfstream.exceptions.**AuthenticationError**
Authentication failure

exception aiosfstream.exceptions.**ClientError**
Client side error

exception aiosfstream.exceptions.**ClientInvalidOperation**
The requested operation can't be executed on the current state of the client

exception aiosfstream.exceptions.**TransportError**
Error during the transportation of messages

exception aiosfstream.exceptions.**TransportInvalidOperation**
The requested operation can't be executed on the current state of the transport

exception aiosfstream.exceptions.**TransportTimeoutError**
Transport timeout

exception aiosfstream.exceptions.**TransportConnectionClosed**
The connection unexpectedly closed

exception aiosfstream.exceptions.**ServerError**(*message*, *response*)
Streaming API server side error

If the *response* contains an error field it gets parsed according to the specs

Parameters

- **message** (*str*) – Error description
- **response** (*dict*) – Server response message

If the *response* contains an error field it gets parsed according to the specs

Parameters

- **message** (str) – Error description
- **response** (Optional[Dict[str, Any]]) – Server response message

message
Error description

Return type str

response
Server response message

Return type Optional[Dict[str, Any]]

error
Error field in the *response*

Return type Optional[str]

error_code
Error code part of the error code part of the error, message field

Return type Optional[int]

error_args
Arguments part of the error, message field

Return type Optional[List[str]]

error_message
Description part of the error, message field

Return type Optional[str]

exception aiosfstream.exceptions.**ReplayError**
Message replay related error

4.3 Changelog

4.3.1 0.5.0 (2019-03-08)

- Add support for Change Data Capture events
- Fix some typos in the documentation

4.3.2 0.4.0 (2019-01-06)

- Add type hints
- Configurable replay storage behavior

4.3.3 0.3.0 (2018-11-07)

- Add support for sandbox orgs

4.3.4 0.2.5 (2018-11-06)

- Add missing changelog entries

4.3.5 0.2.4 (2018-11-06)

- Fix platform event message creation date extraction issue

4.3.6 0.2.3 (2018-09-19)

- Fix asynchronous iterator bug in python 3.7

4.3.7 0.2.2 (2018-06-15)

- Update aiocometd dependency to 0.3.1

4.3.8 0.2.1 (2018-05-25)

- Fix replay issues on mass record delete operations
- Improve the documentation of the Client.publish method

4.3.9 0.2.0 (2018-05-05)

- Enable the usage of third party JSON libraries
- Expose authentication results as public attributes in Authenticator classes

4.3.10 0.1.0 (2018-04-26)

- **Supported authentication types:**
  - using a username and password
  - using a refresh token
- **Subscribe to and receive messages on:**
  - PushTopics
  - Generic Streaming Channels
- Support for durable messages and replay of events
- Automatic recovery from replay errors

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

C

D

E

G

H

M

N

O

P

R

S

T

U