aiobotocore Documentation

Release 0.10.0-

Nikolay Novik

Jan 26, 2019

Contents

Async client for amazon services using botocore and aiohttp/asyncio.

Main purpose of this library to support amazon S3 API, but other services should work (may be with minor fixes). For now we have tested only upload/download API for S3, other users report that SQS and Dynamo services work also. More tests coming soon.

CHAPTER 1

Features

- Full async support for AWS services with botocore.
- Library used in production with S3, SQS and Dynamo services

CHAPTER 2

Basic Example

```
import asyncio
import aiobotocore

AWS_ACCESS_KEY_ID = "xxx"
AWS_SECRET_ACCESS_KEY = "xxx"


async def go(loop):
    bucket = 'dataintake'
    filename = 'dummy.bin'
    folder = 'aiobotocore'
    key = '{}/{}'.format(folder, filename)

    session = aiobotocore.get_session(loop=loop)
    async with session.create_client('s3', region_name='us-west-2',
                                     aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
                                     aws_access_key_id=AWS_ACCESS_KEY_ID) as client:
        # upload object to amazon s3
        data = b'\x01'*1024
        resp = await client.put_object(Bucket=bucket,
                                       Key=key,
                                       Body=data)
        print(resp)

        # getting s3 object properties of file we just uploaded
        resp = await client.get_object_acl(Bucket=bucket, Key=key)
        print(resp)

        # get object from s3
        response = await client.get_object(Bucket=bucket, Key=key)
        # this will ensure the connection is correctly re-used/closed
        async with response['Body'] as stream:
            assert await stream.read() == data
```

(continues on next page)

(continued from previous page)

```
        # list s3 objects using paginator
        paginator = client.get_paginator('list_objects')
        async for result in paginator.paginate(Bucket=bucket, Prefix=folder):
            for c in result.get('Contents', []):
                print(c)

        # delete object from s3
        resp = await client.delete_object(Bucket=bucket, Key=key)
        print(resp)

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

CHAPTER 3

awscli

awscli depends on a single version of botocore, however aiobotocore only supports a specific range of botocore versions. To ensure you install the latest version of awscli that your specific combination or aiobotocore and botocore can support use:

```
pip install -U aiobotocore[awscli]
```

CHAPTER 4

Contents

4.1 Getting Started With aiobotocore

Following tutorial based on botocore tutorial.

The `aiobotocore` package provides a low-level interface to Amazon services. It is responsible for:

- Providing access to all available services
- Providing access to all operations within a service
- Marshaling all parameters for a particular operation in the correct format
- Signing the request with the correct authentication signature
- Receiving the response and returning the data in native Python data structures

`aiobotocore` does not provide higher-level abstractions on top of these services, operations and responses. That is left to the application layer. The goal of `aiobotocore` is to handle all of the low-level details of making requests and getting results from a service.

The `aiobotocore` package is mainly data-driven. Each service has a JSON description which specifies all of the operations the service supports, all of the parameters the operation accepts, all of the documentation related to the service, information about supported regions and endpoints, etc. Because this data can be updated quickly based on the canonical description of these services, it's much easier to keep `aiobotocore` current.

4.2 Using Botocore

The first step in using aiobotocore is to create a `Session` object. `Session` objects then allow you to create individual clients:

```
session = aiobotocore.get_session(loop=loop)
client = session.create_client('s3', region_name='us-west-2',
                               aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
                               aws_access_key_id=AWS_ACCESS_KEY_ID)
```

Once you have that client created, each operation provided by the service is mapped to a method. Each method takes `**kwargs` that maps to the parameter names exposed by the service. For example, using the `client` object created above:

```
# upload object to amazon s3
data = b'\x01'*1024
resp = await client.put_object(Bucket=bucket,
                               Key=key, Body=data)
print(resp)

# getting s3 object properties of file we just uploaded
resp = await client.get_object_acl(Bucket=bucket, Key=key)
print(resp)

# delete object from s3
resp = await client.delete_object(Bucket=bucket, Key=key)
print(resp)
```

4.3 Examples of aiobotocore usage

Below is a list of examples from aiobotocore/examples

Every example is a correct tiny python program.

4.3.1 Basic Usage

Simple put, get, delete example for S3 service:

```
import asyncio
import aiobotocore

AWS_ACCESS_KEY_ID = "xxx"
AWS_SECRET_ACCESS_KEY = "xxx"


async def go(loop):

    bucket = 'dataintake'
    filename = 'dummy.bin'
    folder = 'aiobotocore'
    key = '{}/{}'.format(folder, filename)

    session = aiobotocore.get_session(loop=loop)
    async with session.create_client(
            's3', region_name='us-west-2',
            aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
            aws_access_key_id=AWS_ACCESS_KEY_ID) as client:
        # upload object to amazon s3
        data = b'\x01' * 1024
        resp = await client.put_object(Bucket=bucket,
                                       Key=key,
                                       Body=data)
        print(resp)
```

(continues on next page)

(continued from previous page)

```
        # getting s3 object properties of file we just uploaded
        resp = await client.get_object_acl(Bucket=bucket, Key=key)
        print(resp)

        # delete object from s3
        resp = await client.delete_object(Bucket=bucket, Key=key)
        print(resp)


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

4.3.2 SQS

Queue Create

This snippet creates a queue, lists the queues, then deletes the queue.

```
# Boto should get credentials from ~/.aws/credentials or the environment
import asyncio

import aiobotocore


async def go(loop):
    session = aiobotocore.get_session(loop=loop)
    client = session.create_client('sqs', region_name='us-west-2')

    print('Creating test_queue1')
    response = await client.create_queue(QueueName='test_queue1')
    queue_url = response['QueueUrl']

    response = await client.list_queues()

    print('Queue URLs:')
    for queue_name in response.get('QueueUrls', []):
        print(' ' + queue_name)

    print('Deleting queue {0}'.format(queue_url))
    await client.delete_queue(QueueUrl=queue_url)

    print('Done')
    await client.close()


def main():
    try:
        loop = asyncio.get_event_loop()
        loop.run_until_complete(go(loop))
    except KeyboardInterrupt:
        pass


if __name__ == '__main__':
    main()
```

Producer Consumer

Here is a quick and simple producer/consumer example. The producer will put messages on the queue with a delay of up to 4 seconds between each put. The consumer will read off any messages on the queue, waiting up to 2 seconds for messages to appear before returning.

```
#!/usr/bin/env python3
"""
aiobotocore SQS Producer Example
"""
import asyncio
import random
import sys

import aiobotocore
import botocore.exceptions

QUEUE_NAME = 'test_queue12'


async def go(loop):
    # Boto should get credentials from ~/.aws/credentials or the environment
    session = aiobotocore.get_session(loop=loop)
    client = session.create_client('sqs', region_name='us-west-2')
    try:
        response = await client.get_queue_url(QueueName=QUEUE_NAME)
    except botocore.exceptions.ClientError as err:
        if err.response['Error']['Code'] == \
                'AWS.SimpleQueueService.NonExistentQueue':
            print("Queue {0} does not exist".format(QUEUE_NAME))
            await client.close()
            sys.exit(1)
        else:
            raise

    queue_url = response['QueueUrl']

    print('Putting messages on the queue')

    msg_no = 1
    while True:
        try:
            msg_body = 'Message #{0}'.format(msg_no)
            await client.send_message(
                QueueUrl=queue_url,
                MessageBody=msg_body
            )
            msg_no += 1

            print('Pushed "{0}" to queue'.format(msg_body))

            await asyncio.sleep(random.randint(1, 4))
        except KeyboardInterrupt:
            break

    print('Finished')
    await client.close()
```

(continues on next page)

(continued from previous page)

```
def main():
    try:
        loop = asyncio.get_event_loop()
        loop.run_until_complete(go(loop))
    except KeyboardInterrupt:
        pass


if __name__ == '__main__':
    main()
```

```
#!/usr/bin/env python3
"""
aiobotocore SQS Consumer Example
"""
import asyncio
import sys

import aiobotocore
import botocore.exceptions

QUEUE_NAME = 'test_queue12'


async def go(loop):
    # Boto should get credentials from ~/.aws/credentials or the environment
    session = aiobotocore.get_session(loop=loop)
    client = session.create_client('sqs', region_name='us-west-2')
    try:
        response = await client.get_queue_url(QueueName=QUEUE_NAME)
    except botocore.exceptions.ClientError as err:
        if err.response['Error']['Code'] == \
                'AWS.SimpleQueueService.NonExistentQueue':
            print("Queue {0} does not exist".format(QUEUE_NAME))
            await client.close()
            sys.exit(1)
        else:
            raise

    queue_url = response['QueueUrl']

    print('Pulling messages off the queue')

    while True:
        try:
            # This loop wont spin really fast as there is
            # essentially a sleep in the receieve_message call
            response = await client.receive_message(
                QueueUrl=queue_url,
                WaitTimeSeconds=2,
            )

            if 'Messages' in response:
                for msg in response['Messages']:
                    print('Got msg "{0}"'.format(msg['Body']))
```

(continues on next page)

(continued from previous page)

```
                    # Need to remove msg from queue or else it'll reappear
                    await client.delete_message(
                        QueueUrl=queue_url,
                        ReceiptHandle=msg['ReceiptHandle']
                    )
            else:
                print('No messages in queue')
        except KeyboardInterrupt:
            break

    print('Finished')
    await client.close()


def main():
    try:
        loop = asyncio.get_event_loop()
        loop.run_until_complete(go(loop))
    except KeyboardInterrupt:
        pass


if __name__ == '__main__':
    main()
```

4.3.3 DynamoDB

Table Creation

When you create a DynamoDB table, it can take quite a while (especially if you add a few secondary index's). Instead of polling *describe_table* yourself, boto3 came up with "waiters" that will do all the polling for you. The following snippet shows how to wait for a DynamoDB table to be created in an async way.

```
# Boto should get credentials from ~/.aws/credentials or the environment
import uuid
import asyncio

import aiobotocore


async def go(loop):
    session = aiobotocore.get_session(loop=loop)
    client = session.create_client('dynamodb', region_name='us-west-2')
    # Create random table name
    table_name = 'aiobotocore-' + str(uuid.uuid4())

    print('Requesting table creation...')
    await client.create_table(
        TableName=table_name,
        AttributeDefinitions=[
            {
                'AttributeName': 'testKey',
                'AttributeType': 'S'
            },
```

(continues on next page)

(continued from previous page)

```
        ],
        KeySchema=[
            {
                'AttributeName': 'testKey',
                'KeyType': 'HASH'
            },
        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 10,
            'WriteCapacityUnits': 10
        }
    )

    print("Waiting for table to be created...")
    waiter = client.get_waiter('table_exists')
    await waiter.wait(TableName=table_name)
    print("Table {0} created".format(table_name))

    await client.close()


def main():
    try:
        loop = asyncio.get_event_loop()
        loop.run_until_complete(go(loop))
    except KeyboardInterrupt:
        pass


if __name__ == '__main__':
    main()
```

Batch Insertion

Now if you have a massive amount of data to insert into Dynamo, I would suggest using an EMR data pipeline (theres even an example for exactly this). But if you stubborn, here is an example of inserting lots of items into Dynamo (it's not really that complicated once you've read it).

What the code does is generates items (e.g. item0, item1, item2...) and writes them to a table "test" against a primary partition key called "pk" (with 5 read and 5 write units, no auto-scaling).

The *batch_write_item* method only takes a max of 25 items at a time, so the script computes 25 items, writes them, then does it all over again.

After Dynamo has had enough, it will start throttling you and return any items that have not been written in the response. Once the script is being throttled, it will start sleeping for 5 seconds until the failed items have been successfully written, after that it will exit.

```
# Boto should get credentials from ~/.aws/credentials or the environment
import asyncio

import aiobotocore


def get_items(start_num, num_items):
    """
```

(continues on next page)

(continued from previous page)

```
    Generate a sequence of dynamo items

    :param start_num: Start index
    :type start_num: int
    :param num_items: Number of items
    :type num_items: int
    :return: List of dictionaries
    :rtype: list of dict
    """
    result = []
    for i in range(start_num, start_num+num_items):
        result.append({'pk': {'S': 'item{0}'.format(i)}})
    return result


def create_batch_write_structure(table_name, start_num, num_items):
    """
    Create item structure for passing to batch_write_item

    :param table_name: DynamoDB table name
    :type table_name: str
    :param start_num: Start index
    :type start_num: int
    :param num_items: Number of items
    :type num_items: int
    :return: dictionary of tables to write to
    :rtype: dict
    """
    return {
        table_name: [
            {'PutRequest': {'Item': item}}
            for item in get_items(start_num, num_items)
        ]
    }


async def go(loop):
    session = aiobotocore.get_session(loop=loop)
    client = session.create_client('dynamodb', region_name='us-west-2')
    table_name = 'test'

    print('Writing to dynamo')
    start = 0
    while True:
        # Loop adding 25 items to dynamo at a time
        request_items = create_batch_write_structure(table_name, start, 25)
        response = await client.batch_write_item(
            RequestItems=request_items
        )
        if len(response['UnprocessedItems']) == 0:
            print('Writted 25 items to dynamo')
        else:
            # Hit the provisioned write limit
            print('Hit write limit, backing off then retrying')
            await asyncio.sleep(5)

            # Items left over that haven't been inserted
```

(continues on next page)

(continued from previous page)

```
            unprocessed_items = response['UnprocessedItems']
            print('Resubmitting items')
            # Loop until unprocessed items are written
            while len(unprocessed_items) > 0:
                response = await client.batch_write_item(
                    RequestItems=unprocessed_items
                )
                # If any items are still left over, add them to the
                # list to be written
                unprocessed_items = response['UnprocessedItems']

                # If there are items left over, we could do with
                # sleeping some more
                if len(unprocessed_items) > 0:
                    print('Backing off for 5 seconds')
                    await asyncio.sleep(5)

            # Inserted all the unprocessed items, exit loop
            print('Unprocessed items successfully inserted')
            break

        start += 25

    # See if DynamoDB has the last item we inserted
    final_item = 'item' + str(start + 24)
    print('Item "{0}" should exist'.format(final_item))

    response = await client.get_item(
        TableName=table_name,
        Key={'pk': {'S': final_item}}
    )
    print('Response: ' + str(response['Item']))

    await client.close()


def main():
    try:
        loop = asyncio.get_event_loop()
        loop.run_until_complete(go(loop))
    except KeyboardInterrupt:
        pass


if __name__ == '__main__':
    main()
```

4.4 API

4.5 Contributing

4.5.1 Running Tests

Thanks for your interest in contributing to `aiobotocore`, there are multiple ways and places you can contribute.

Fist of all just clone repository:

```
$ git clone git@github.com:aio-libs/aiobotocore.git
```

Create virtualenv with at least python3.5 (older version are not supported). For example using *virtualenvwrapper* commands could look like:

```
$ cd aiobotocore
$ mkvirtualenv --python=`which python3.5` aiobotocore
```

After that please install libraries required for development:

```
$ pip install -r requirements-dev.txt
$ pip install -e .
```

Congratulations, you are ready to run the test suite:

```
$ make cov
```

To run individual use following command:

```
$ py.test -sv tests/test_monitor.py -k test_name
```

4.5.2 Reporting an Issue

If you have found issue with *aiobotocore* please do not hesitate to file an issue on the GitHub project. When filing your issue please make sure you can express the issue with a reproducible test case.

When reporting an issue we also need as much information about your environment that you can include. We never know what information will be pertinent when trying narrow down the issue. Please include at least the following information:

- Version of *aiobotocore* and *python*.
- Version fo *botocore*.
- Platform you're running on (OS X, Linux).

4.5.3 Background and Implementation

aiobotocore adds async functionality to botocore by replacing certain critical methods in botocore classes with async versions. The best way to see how this works is by working backwards from *AioEndpoint._request*. Because of this tight integration aiobotocore is typically version locked to a particular release of botocore.

4.5.4 How to Upgrade Botocore

aiobotocore's file names try to match the botocore files they functionally match. For the most part botocore classes are sub-classed with the majority of the botocore calls eventually called... however certain methods like *PageIterator.next_page* had to be re-implemented so watch for changes in those types of methods.

The best way I've seen to upgrade botocore support is by downloading the sources of the release of botocore you're trying to upgrade to, and the version of botocore that aiobotocore is currently locked to and do a folder based file comparison (tools like DiffMerge are nice). You can then manually apply the relevant changes to their aiobotocore equivalent(s). In order to support a range of versions one would need validate the version each change was introduced and select the newest of these to the current version. This is further complicated by the aiobotocore "extras" requirements which need to be updated to the versions that are compatible with the above changes.

See next section describing types of changes we must validate and support.

4.5.5 Hashes of Botocore Code (important)

Because of the way aiobotocore is implemented (see Background section), it is very tightly coupled with botocore. The validity of these couplings are enforced in test_patches.py. We also depend on some private properties in aiohttp, and because of this have entries in test_patches.py for this too.

These patches are important to catch cases where botocore functionality was added/removed and needs to be reflected in our overridden methods. Changes include:

- parameters to methods added/removed
- classes/methods being moved to new files
- bodies of overridden methods updated

To ensure we catch and reflect this changes in aiobotocore, the test_patches.py file has the hashes of the parts of botocore we need to manually validate changes in.

test_patches.py file needs to be updated in two scenarios:

1. You're bumping the supported botocore/aiohttp version. In this case a failure in test_patches.py means you need to validate the section of code in aiohttp/botocore that no longer matches the hash in test_patches.py to see if any changes need to be reflected in aiobotocore which overloads, on depends on the code which triggered the hash mismatch. This could there are new parameters we weren't expecting, parameters that are no longer passed to said overriden function(s), or an overridden function which calls a modified botocore method. If this is a whole class collision the checks will be more extensive.
2. You're implementing missing aiobotocore functionality, in which case you need to add entries for all the methods in botocore/aiohttp which you are overriding or depending on private functionality. For special cases, like when private attributes are used, you may have to hash the whole class so you can catch any case where the private property is used/updated to ensure it matches our expectations.

After you've validated the changes, you can update the hash in test_patches.py.

One would think we could just write enough unittests to catch all cases, however, this is impossible for two reasons:

1. We do not support all botocore unittests, for future work see discussion: https://github.com/aio-libs/aiobotocore/issues/213
2. Even if we did all the unittests from 1, we would not support NEW functionality added, unless we automatically pulled all new unittests as well from botocore.

Until we can perform ALL unittests from new releases of botocore, we are stuck with the patches.

4.5.6 The Future

The long term goal is that botocore will implement async functionality directly. See botocore issue: https://github.com/boto/botocore/issues/458 for details, tracked in aiobotocore here: https://github.com/aio-libs/aiobotocore/issues/36

CHAPTER 5

Indices and tables

- genindex
- modindex
- search