KQ

Feb 04, 2022

Contents

KQ (Kafka Queue) is a Python library which lets you enqueue and execute jobs asynchronously using Apache Kafka. It uses kafka-python under the hood.

CHAPTER 1

Requirements

- Apache Kafka 0.9+
- Python 3.6+

CHAPTER 2

Installation

Install via pip:

```
pip install kq
```

CHAPTER 3

Contents

3.1 Getting Started

Start your Kafka instance. Example using Docker:

```
docker run -p 9092:9092 -e ADV_HOST=127.0.0.1 lensesio/fast-data-dev
```

Define your KQ `worker.py module:

```
import logging

from kafka import KafkaConsumer
from kq import Worker

# Set up logging.
formatter = logging.Formatter('[%(levelname)s] %(message)s')
stream_handler = logging.StreamHandler()
stream_handler.setFormatter(formatter)
logger = logging.getLogger('kq.worker')
logger.setLevel(logging.DEBUG)
logger.addHandler(stream_handler)

# Set up a Kafka consumer.
consumer = KafkaConsumer(
    bootstrap_servers='127.0.0.1:9092',
    group_id='group',
    auto_offset_reset='latest'
)

# Set up a worker.
worker = Worker(topic='topic', consumer=consumer)
worker.start()
```

Start the worker:

```
python my_worker.py
[INFO] Starting Worker(hosts=127.0.0.1:9092 topic=topic, group=group) ...
```

Enqueue a function call:

```
import requests

from kafka import KafkaProducer
from kq import Queue

# Set up a Kafka producer.
producer = KafkaProducer(bootstrap_servers='127.0.0.1:9092')

# Set up a queue.
queue = Queue(topic='topic', producer=producer)

# Enqueue a function call.
job = queue.enqueue(requests.get, 'https://google.com')

# You can also specify the job timeout, Kafka message key and partition.
job = queue.using(timeout=5, key=b'foo', partition=0).enqueue(requests.get, 'https://
↪google.com')
```

Let the worker process it in the background:

```
python my_worker.py
[INFO] Starting Worker(hosts=127.0.0.1:9092, topic=topic, group=group) ...
[INFO] Processing Message(topic=topic, partition=0, offset=0) ...
[INFO] Executing job c7bf2359: requests.api.get('https://www.google.com')
[INFO] Job c7bf2359 returned: <Response [200]>
```

3.2 Queue

class kq.queue.**Queue**(*topic: str, producer: kafka.producer.kafka.KafkaProducer, serializer: Optional[Callable[[...], bytes]] = None, timeout: int = 0, logger: Optional[logging.Logger] = None*)

Enqueues function calls in Kafka topics as *jobs*.

Parameters

- **topic** (*str*) – Name of the Kafka topic.
- **producer** (kafka.KafkaProducer) – Kafka producer instance. For more details on producers, refer to kafka-python's documentation.
- **serializer** (*callable*) – Callable which takes a *job* namedtuple and returns a serialized byte string. If not set, `dill.dumps` is used by default. See *here* for more details.
- **timeout** (*int | float*) – Default job timeout threshold in seconds. If left at 0 (default), jobs run until completion. This value can be overridden when enqueueing jobs.
- **logger** (*logging.Logger*) – Logger for recording queue activities. If not set, logger named `kq.queue` is used with default settings (you need to define your own formatters and handlers). See *here* for more details.

Example:

```
import requests

from kafka import KafkaProducer
from kq import Queue

# Set up a Kafka producer.
producer = KafkaProducer(bootstrap_servers='127.0.0.1:9092')

# Set up a queue.
queue = Queue(topic='topic', producer=producer, timeout=3600)

# Enqueue a function call.
job = queue.enqueue(requests.get, 'https://www.google.com/')
```

hosts
Return comma-separated Kafka hosts and ports string.

Returns Comma-separated Kafka hosts and ports.

Return type str

topic
Return the name of the Kafka topic.

Returns Name of the Kafka topic.

Return type str

producer
Return the Kafka producer instance.

Returns Kafka producer instance.

Return type kafka.KafkaProducer

serializer
Return the serializer function.

Returns Serializer function.

Return type callable

timeout
Return the default job timeout threshold in seconds.

Returns Default job timeout threshold in seconds.

Return type float | int

enqueue (*func: Callable[[...], bytes], *args, **kwargs*) → kq.job.Job
Enqueue a function call or a *job*.

Parameters

- **func** (callable | *kq.Job*) – Function or a *job* object. Must be serializable and available to *workers*.
- **args** – Positional arguments for the function. Ignored if **func** is a *job* object.
- **kwargs** – Keyword arguments for the function. Ignored if **func** is a *job* object.

Returns Enqueued job.

Return type *kq.Job*

Example:

```
import requests

from kafka import KafkaProducer
from kq import  Job, Queue

# Set up a Kafka producer.
producer = KafkaProducer(bootstrap_servers='127.0.0.1:9092')

# Set up a queue.
queue = Queue(topic='topic', producer=producer)

# Enqueue a function call.
queue.enqueue(requests.get, 'https://www.google.com/')

# Enqueue a job object.
job = Job(func=requests.get, args=['https://www.google.com/'])
queue.enqueue(job)
```

Note: The following rules apply when enqueueing a *job*:

- If `Job.id` is not set, a random one is generated.
- If `Job.timestamp` is set, it is replaced with current time.
- If `Job.topic` is set, it is replaced with current topic.
- If `Job.timeout` is set, its value overrides others.
- If `Job.key` is set, its value overrides others.
- If `Job.partition` is set, its value overrides others.

---

using (*timeout: Union[float, int, None] = None*, *key: Optional[bytes] = None*, *partition: Optional[int] = None*) → kq.queue.EnqueueSpec

Set enqueue specifications such as timeout, key and partition.

Parameters

- **timeout** (*int | float*) – Job timeout threshold in seconds. If not set, default timeout (specified during queue initialization) is used instead.
- **key** (*bytes*) – Kafka message key. Jobs with the same keys are sent to the same topic partition and executed sequentially. Applies only if the **partition** parameter is not set, and the producer's partitioner configuration is left as default. For more details on producers, refer to kafka-python's documentation.
- **partition** (*int*) – Topic partition the message is sent to. If not set, the producer's partitioner selects the partition. For more details on producers, refer to kafka-python's documentation.

Returns Enqueue specification object which has an `enqueue` method with the same signature as *kq.queue.Queue.enqueue()*.

Return type EnqueueSpec

Example:

```
import requests

from kafka import KafkaProducer
from kq import Job, Queue

# Set up a Kafka producer.
producer = KafkaProducer(bootstrap_servers='127.0.0.1:9092')

# Set up a queue.
queue = Queue(topic='topic', producer=producer)

url = 'https://www.google.com/'

# Enqueue a function call in partition 0 with message key 'foo'.
queue.using(partition=0, key=b'foo').enqueue(requests.get, url)

# Enqueue a function call with a timeout of 10 seconds.
queue.using(timeout=10).enqueue(requests.get, url)

# Job values are preferred over values set with "using" method.
job = Job(func=requests.get, args=[url], timeout=5)
queue.using(timeout=10).enqueue(job)  # timeout is still 5
```

3.3 Worker

class kq.worker.**Worker**(*topic: str, consumer: kafka.consumer.group.KafkaConsumer, callback: Optional[Callable[[...], Any]] = None, deserializer: Optional[Callable[[bytes], Any]] = None, logger: Optional[logging.Logger] = None*)

Fetches *jobs* from Kafka topics and processes them.

Parameters

- **topic** (*str*) – Name of the Kafka topic.
- **consumer** (kafka.KafkaConsumer) – Kafka consumer instance with a group ID (required). For more details on consumers, refer to kafka-python's documentation.
- **callback** (*callable*) – Callback function which is executed every time a job is processed. See *here* for more details.
- **deserializer** (*callable*) – Callable which takes a byte string and returns a deserialized *job* namedtuple. If not set, `dill.loads` is used by default. See *here* for more details.
- **logger** (*logging.Logger*) – Logger for recording worker activities. If not set, logger named `kq.worker` is used with default settings (you need to define your own formatters and handlers). See *here* for more details.

Example:

```
from kafka import KafkaConsumer
from kq import Worker

# Set up a Kafka consumer. Group ID is required.
consumer = KafkaConsumer(
    bootstrap_servers='127.0.0.1:9092',
    group_id='group'
```

(continues on next page)

(continued from previous page)

```
)

# Set up a worker.
worker = Worker(topic='topic', consumer=consumer)

# Start the worker to process jobs.
worker.start()
```

hosts
Return comma-separated Kafka hosts and ports string.

Returns Comma-separated Kafka hosts and ports.

Return type str

topic
Return the name of the Kafka topic.

Returns Name of the Kafka topic.

Return type str

group
Return the Kafka consumer group ID.

Returns Kafka consumer group ID.

Return type str

consumer
Return the Kafka consumer instance.

Returns Kafka consumer instance.

Return type kafka.KafkaConsumer

deserializer
Return the deserializer function.

Returns Deserializer function.

Return type callable

callback
Return the callback function.

Returns Callback function, or None if not set.

Return type callable | None

start(*max_messages: Optional[int] = None*, *commit_offsets: bool = True*) → int
Start processing Kafka messages and executing jobs.

Parameters

- **max_messages** (*int | None*) – Maximum number of Kafka messages to process before stopping. If not set, worker runs until interrupted.
- **commit_offsets** (*bool*) – If set to True, consumer offsets are committed every time a message is processed (default: True).

Returns Total number of messages processed.

Return type int

3.4 Job

KQ encapsulates jobs using `kq.Job` dataclass:

```
from dataclasses import dataclass
from typing import Callable, Dict, List, Optional, Union


@dataclass(frozen=True)
class Job:

    # KQ job UUID.
    id: Optional[str] = None

    # Unix timestamp indicating when the job was queued.
    timestamp: Optional[int] = None

    # Name of the Kafka topic.
    topic: Optional[str] = None

    # Function to execute.
    func: Optional[Callable] = None

    # Positional arguments for the function.
    args: Optional[List] = None

    # Keyword arguments for the function.
    kwargs: Optional[Dict] = None

    # Job timeout threshold in seconds.
    timeout: Optional[Union[float, int]] = None

    # Kafka message key. Jobs with the same keys are sent
    # to the same topic partition and executed sequentially.
    # Applies only when the "partition" field is not set.
    key: Optional[str] = None

    # Kafka topic partition. If set, the "key" field is ignored.
    partition: Optional[str] = None
```

When a function call is enqueued, an instance of this dataclass is created to store the message and the metadata. It is then serialized into a byte string and sent to Kafka.

3.5 Message

KQ encapsulates Kafka messages using `kq.Message` dataclass:

```
from dataclasses import dataclass
from typing import Optional


@dataclass(frozen=True)
class Message:
    # Name of the Kafka topic.
    topic: str
```

(continues on next page)

(continued from previous page)

```
    # Kafka topic partition.
    partition: int

    # Partition offset.
    offset: int

    # Kafka message key.
    key: Optional[bytes]

    # Kafka message payload.
    value: bytes
```

Raw Kafka messages are converted into above dataclasses, which are then sent to *workers* (and also to *callback functions* if defined).

3.6 Callback

KQ lets you assign a callback function to workers. The callback function is invoked each time a message is processed. It must accept the following positional arguments:

- **status** (str): Job status. Possible values are:
  - `invalid` : Job could not be deserialized, or was malformed.
  - `failure` : Job raised an exception.
  - `timeout` : Job took too long and timed out.
  - `success` : Job successfully finished and returned a result.
- **message** (*kq.Message*): Kafka message.
- **job** (*kq.Job* | None): Job object, or None if Kafka message was invalid or malformed.
- **result** (object | None): Job result, or None if an exception was raised.
- **exception** (Exception | None): Exception raised, or None if job finished successfully.
- **stacktrace** (str | None): Exception stacktrace, or None if job finished successfully.

You can assign your callback function during *worker* initialization.

Example:

```
from kafka import KafkaConsumer
from kq import Worker


def callback(status, message, job, result, exception, stacktrace):
    """This is an example callback showing what arguments to expect."""

    assert status in ['invalid', 'success', 'timeout', 'failure']
    assert isinstance(message, kq.Message)

    if status == 'invalid':
        assert job is None
        assert result is None
        assert exception is None
```

(continues on next page)

(continued from previous page)

```
        assert stacktrace is None

    if status == 'success':
        assert isinstance(job, kq.Job)
        assert exception is None
        assert stacktrace is None

    elif status == 'timeout':
        assert isinstance(job, kq.Job)
        assert result is None
        assert exception is None
        assert stacktrace is None

    elif status == 'failure':
        assert isinstance(job, kq.Job)
        assert result is None
        assert exception is not None
        assert stacktrace is not None

consumer = KafkaConsumer(
    bootstrap_servers='127.0.0.1:9092',
    group_id='group'
)

# Inject your callback function during worker initialization.
worker = Worker('topic', consumer, callback=callback)
```

3.7 Serializer

You can use custom functions for serialization. By default, KQ uses the dill library.

The serializer function must take a *job* namedtuple and return a byte string. You can inject it during queue initialization.

Example:

```
# Let's use pickle instead of dill
import pickle

from kafka import KafkaProducer
from kq import Queue

producer = KafkaProducer(bootstrap_servers='127.0.0.1:9092')

# Inject your serializer function during queue initialization.
queue = Queue('topic', producer, serializer=pickle.dumps)
```

The deserializer function must take a byte string and returns a *job* namedtuple. You can inject it during worker initialization.

Example:

```
# Let's use pickle instead of dill
import pickle

from kafka import KafkaConsumer
```

(continues on next page)

(continued from previous page)

```
from kq import Worker

consumer = KafkaConsumer(
    bootstrap_servers='127.0.0.1:9092',
    group_id='group'
)

# Inject your deserializer function during worker initialization.
worker = Worker('topic', consumer, deserializer=pickle.loads)
```

3.8 Logging

By default, *queues* log messages to `kq.queue` logger, and *workers* to `kq.worker` logger. You can use these default loggers or set your own during queue/worker initialization.

Example:

```
import logging

from kafka import KafkaConsumer, KafkaProducer
from kq import Queue, Worker

formatter = logging.Formatter('[%(levelname)s] %(message)s')
stream_handler = logging.StreamHandler()
stream_handler.setFormatter(formatter)

# Set up "kq.queue" logger.
queue_logger = logging.getLogger('kq.queue')
queue_logger.setLevel(logging.INFO)
queue_logger.addHandler(stream_handler)

# Set up "kq.worker" logger.
worker_logger = logging.getLogger('kq.worker')
worker_logger.setLevel(logging.DEBUG)
worker_logger.addHandler(stream_handler)

# Alternatively, you can inject your own loggers.
queue_logger = logging.getLogger('your_worker_logger')
worker_logger = logging.getLogger('your_worker_logger')

producer = KafkaProducer(bootstrap_servers='127.0.0.1:9092')
consumer = KafkaConsumer(bootstrap_servers='127.0.0.1:9092', group_id='group')

queue = Queue('topic', producer, logger=queue_logger)
worker = Worker('topic', consumer, logger=worker_logger)
```

Index