KQ Documentation

Release 2.0.0

Joohwan Oh

Jun 05, 2018

Contents

Welcome to the documentation for **KQ (Kafka Queue)**, a lightweight Python library which lets you queue and execute jobs asynchronously using Apache Kafka. It uses kafka-python under the hood.

CHAPTER 1

Requirements

- Apache Kafka 0.9+
- Python 3.5+

CHAPTER 2

Installation

To install a stable version from PyPI (recommended):

```
~$ pip install kq
```

To install the latest version directly from GitHub:

```
~$ pip install -e git+git@github.com:joowani/kq.git@master#egg=kq
```

You may need to use `sudo` depending on your environment.

CHAPTER 3

Contents

3.1 Getting Started

First, ensure that your Kafka instance is up and running:

```
~$ ./kafka-server-start.sh -daemon server.properties
```

Define your KQ worker module:

```
# my_worker.py

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://www.google.com')
```

Sit back and watch 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]>
```

You can also specify the job timeout, message key and partition:

```
job = queue.using(timeout=5, key=b'foo', partition=0).enqueue(requests.get, 'https://
↪www.google.com')
```

3.2 Queue

class kq.queue.**Queue**(*topic*, *producer*, *serializer=None*, *timeout=0*, *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 int | float

enqueue (*func*, **args*, ***kwargs*)
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=None*, *key=None*, *partition=None*)

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()*.

Example:

```
import requests

from kafka import KafkaProducer
from kq import Job, Queue
```

(continues on next page)

(continued from previous page)

```
# 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*, *consumer*, *callback=None*, *deserializer=None*, *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'
)

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

(continues on next page)

(continued from previous page)

```
# 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=inf*, *commit_offsets=True*)
Start processing Kafka messages and executing jobs.

Parameters

- **max_messages** (*int*) – 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` namedtuples, which have the following fields:

- **id** (str): Job ID.
- **timestamp** (int): Unix timestamp indicating the time of enqueue.
- **topic** (str): Name of the Kafka topic.
- **func** (callable): Function to execute.
- **args** (list | tuple): Positional arguments for the function.
- **kwargs** (dict): Keyword arguments for the function.
- **timeout** (int | float): Job timeout threshold in seconds.
- **key** (bytes | None): Kafka message key. Jobs with the same keys are sent to the same topic partition and executed sequentially. Applies only if the **partition** field is not set, and the producer's partitioner configuration is left as default.
- **partition** (int | None): Kafka topic partition. If set, the **key** field is ignored.

```
from collections import namedtuple

Job = namedtuple(
    typename='Job',
    field_names=(
        'id',
        'timestamp',
        'topic',
        'func',
        'args',
        'kwargs',
        'timeout',
        'key',
        'partition'
    )
)
```

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

3.5 Message

KQ encapsulates Kafka messages using `kq.Message` namedtuples, which have the following fields:

- **topic** (str): Name of the Kafka topic.
- **partition** (int): Kafka topic partition.
- **offset** (int): Partition offset.
- **key** (bytes | None): Kafka message key.
- **value** (bytes): Kafka message payload.

```
from collections import namedtuple

Message = namedtuple(
    typename='Message',
    field_names=(
        'topic',
```

(continues on next page)

(continued from previous page)

```
        'partition',
        'offset',
        'key',
        'value'
    )
)
```

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

3.6 Callback

KQ allows you to assign a callback function to workers. The callback function is invoked every time a message is processed. It must take 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 inject 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
        assert stacktrace is None

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

(continues on next page)

(continued from previous page)

```
    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 inject your own functions for serializing (pickling) jobs. 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
from kq import Worker

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

(continues on next page)

(continued from previous page)

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

3.8 Logging

By default, *queues* log messages via `kq.queue` logger, and *workers* log messages via `kq.worker` logger. You can either use these loggers or inject 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)
```

3.9 Contributing

3.9.1 Requirements

Before submitting a pull request on GitHub, please make sure you meet the following requirements:

- The pull request points to dev branch.
- Changes are squashed into a single commit. I like to use git rebase for this.
- Commit message is in present tense. For example, "Fix bug" is good while "Fixed bug" is not.
- Sphinx-compatible docstrings.

- PEP8 compliance.
- No missing docstrings or commented-out lines.
- Test coverage remains at %100. If a piece of code is trivial and does not need unit tests, use this to exclude it from coverage.
- No build failures on Travis CI. Builds automatically trigger on pull request submissions.
- Documentation is kept up-to-date with the new changes (see below).

Warning: The dev branch is occasionally rebased, and its commit history may be overwritten in the process. Before you begin your feature work, git fetch or pull to ensure that your local branch has not diverged. If you see git conflicts and want to start with a clean slate, run the following commands:

```
~$ git checkout dev
~$ git fetch origin
~$ git reset --hard origin/dev  # THIS WILL WIPE ALL LOCAL CHANGES
```

3.9.2 Style

To ensure PEP8 compliance, run flake8:

```
~$ pip install flake8
~$ git clone https://github.com/joowani/kq.git
~$ cd kq
~$ flake8
```

If there is a good reason to ignore a warning, see here on how to exclude it.

3.9.3 Testing

To test your changes, you can run the integration test suite that comes with **kq**. It uses pytest and requires an actual Kafka instance.

To run the test suite (use your own Kafka broker host and port):

```
~$ pip install pytest
~$ git clone https://github.com/joowani/kq.git
~$ cd kq
~$ py.test -v -s --host=127.0.0.1 --port=9092
```

To run the test suite with coverage report:

```
~$ pip install coverage pytest pytest-cov
~$ git clone https://github.com/joowani/kq.git
~$ cd kq
~$ py.test -v -s --host=127.0.0.1 --port=9092 --cov=kq
```

As the test suite creates real topics and messages, it should only be run in development environments.

3.9.4 Documentation

The documentation including the README is written in reStructuredText and uses Sphinx. To build an HTML version on your local machine:

```
~$ pip install sphinx sphinx_rtd_theme
~$ git clone https://github.com/joowani/kq.git
~$ cd kq/docs
~$ sphinx-build . build  # Open build/index.html in a browser
```

As always, thank you for your contribution!

Index