## CONTENTS

1 Quick intro

2 Contents
   2.1 Installation ............................................. 5
   2.2 Configuration ........................................... 5
   2.3 Jobs ..................................................... 7
   2.4 Logging and metrics ..................................... 8
   2.5 API Documentation ..................................... 9
   2.6 Design notes .............................................. 11
   2.7 Development ............................................ 11
   2.8 Changelog ............................................... 12

3 Index and Glossary ........................................... 13
   3.1 Glossary .................................................. 13

Python Module Index .......................................... 15
**qdo** pronounced *qu-doe*

This is a Python implementation of a worker library used for processing queued messages from a *Mozilla Services* message queue. The message queue is called *Queuey* and is implemented as a *Pyramid* based web service on top of *Cassandra*.
Assuming you have a working installation and setup of *Queuey*, you need at least one Python module, for example `hello.py`:

```python
from contextlib import contextmanager

@contextmanager
def job_context():
    try:
        # do some one-time setup per worker, like open DB connections
        context = {'counter': 0}
        yield context
    finally:
        # tear things down
        print('Messages processed: %s' % context['counter'])

def job(message, context):
    context['counter'] += 1
    print ('%s: %s' % (message['message_id'], message['body']))
```

And one config file named `hello.conf`:

```ini
[qdo-worker]
job = hello:job
job_context = hello:job_context

[queuey]
connection = http://127.0.0.1:5000/v1/queuey/,http://127.0.0.1:5001/v1/queuey/
app_key = f25bfb8fe200475c8a0532a9cbe7651e
```

Then run:

```
$ bin/qdo-worker -c hello.conf
```
2.1 Installation

2.1.1 Requirements

Make sure you have the following software already installed before proceeding:

- Make
- Python 2.7 (or 2.6)
- Java 1.6 (or 1.7)
- Zero MQ (zmq-dev)

2.1.2 Installation

After downloading the repository for the first time, cd into the directory and run:

```
make
```

This will do the following:

- Create a virtual Python environment
- Install required Python packages into this environment

If you want to use the Zookeeper based partitioner, you can also run:

```
make zookeeper
```

2.2 Configuration

`qdo` uses an ini-style configuration file for its configuration. The configuration file is specified via the `-c` option to the `qdo-worker` script. It defaults to `etc/qdo-worker.conf`.

For example:

```
bin/qdo-worker -c etc/my-qdo.conf
```
2.2.1 Settings

Settings are organized into multiple sections.

[qdo-worker]

job The resource specification for the Python job function. For example: qdo.testing:example_job

job_context The resource specification for a Python job context (manager). Defaults to: qdo.worker:dict_context

job_failure The resource specification for a Python exception handler. The default is qdo.worker:log_failure, which logs full tracebacks of job failures using metlog-raven. Another built-in alternative is qdo.worker:save_failed_message, which logs in the same way, but also copies the failed message to an error queue for later inspection.

name An optional identifier used in addition to the current host name and process id to identify the worker process.

cia_bundle Path to a private certificate used for SSL connections, in addition to all officially signed ones, as trusted by the certifi distribution.

wait_interval Interval in seconds for which the worker pauses if it has no messages to work on. Defaults to 30 seconds. The actual wait time adds some jitter of 20%, to avoid multiple workers hitting the Queuey back-end at exactly the same times. It also uses exponential back-off up to a factor of 1024. The back-off factor is reset whenever any message is actually processed.

[partitions]

policy Specifies how to distribute work across workers. Defaults to manual, in which case each worker will work on all partitions it has been assigned to. The other option is automatic, which uses Zookeeper to distribute partitions across workers.

ids A new-line separated list of partitions, for example:

ids =
a4bb2fb6dcda4b68aad743a4746d7f58-1
a4bb2fb6dcda4b68aad743a4746d7f58-2
9588c8c64384f13b7fb32f27a42a9f-1

If no explicit list of ids is given, Queuey is queried for all partitions.

[queuey]

collection Which Queuey instance(s) to connect to. Defaults to http://127.0.0.1:5000/v1/queuey/. Multiple instances can be specified as a comma separated list: https://127.0.0.1:5001/v1/queuey/,https://localhost:5002/v1/queuey/.

If multiple servers are specified, one will be selected as default and the others serve as transparent fallback options. The selection prefers a local server (127.0.0.*, localhost or ::1) and chooses at random amongst multiple other candidates.

app_key The application key used for authorization.

[zookeeper]

collection Which Zookeeper servers to connect to when the automatic policy is used. Defaults to 127.0.0.1:2181/mozilla-qdo, using mozilla-qdo as the top-level namespace. If multiple servers are specified, one will be selected at random and the others will serve as transparent fallback.
**party_wait**  How long does the set partitioner wait for the list of workers to become stable before actually starting any work. Defaults to 10 seconds. This prevents the system from having to reconfigure itself a number of times while worker processes are restarted or new processes come online.

**[metlog]**

For detailed information see the metlog docs.

qdo uses the standard metlog configuration for timing and counter data. In addition you can configure the `metlog_plugin_raven` section, if you want to get full tracebacks logged for job failures.

### 2.3 Jobs

qdo has a very simple job execution model. You can only configure one function for each of the job hooks. The job function is used to process all messages in all queues. There’s currently no categorization or prioritization of messages.

#### 2.3.1 Hooks

There are three different hooks as configured in the configuration file.

**job_context**

The `job_context` points to a Python context manager and is used for one-time setup and tear down of shared resources for all jobs. It’s called at worker startup before any job is executed and its tear down is called after the worker has been told to shut down.

An easy way to write a context manager is using the contextlib module:

```python
from contextlib import contextmanager

@contextmanager
def job_context():
    try:
        # do some one-time setup per worker, like open DB connections
        context = {'counter': 0}
        yield context
    finally:
        # tear things down
        print('Messages processed: %s' % context['counter'])
```

The `job_context` function takes no arguments and yields some context object. Since there’s only one process and no threads involved in the worker itself, you can use simple local or global data structures for the context. The context object can be of any type, as long as the `job` hook can handle it.

**job**

The `job` hook points to a Python callable and is used to process any message:

```python
def job(message, context):
    context['counter'] += 1
    print('%s: %s' % (message['message_id'], message['body']))
```
The callable takes two arguments. A message as returned by Queuey and the context as setup by the `job_context` hook. A Queuey message has some common metadata and the main `body` workload, for example:

```python
{
    'body': 'some data',
    'message_id': '79924e0ff10411e1850bb88d120c81de',
    'partition': 1,
    'timestamp': '1346153731.9620111',
}
```

The timestamp denotes seconds since the Unix epoch in the GMT timezone. Message ids are UUID1’s.

### job_failure

The `job_failure` hook is called whenever an exception is raised inside the job hook. Only Python exceptions inheriting from `Exception` are handled. System exceptions like `SystemExit` or `MemoryError` will cause the job and worker to abort without calling this hook.

Messages will still be marked as processed if a failure occurred. If messages should be retained for reprocessing, you can use the built-in `save_failed_message` function. Either configure it directly or in addition to some custom error handling code:

```python
from qdo.worker import save_failed_message

def log_failure(message, context, queue, exc, queuey_conn):
    # retain messages for re-processing and log tracebacks
    save_failed_message(message, context, queue, exc, queuey_conn)
    # do some custom error handling
    pass
```

The callable takes the original message in the same format as received by the job hook and the same job context. In addition the queue name including the partition is provided, for example `fecafc1678cb4810b4720c41d1c29787-2`. The `exc` argument provides the concrete Python exception that was raised by the job hook. The `queuey_conn` argument provides access to the `queuey_py.Client` used for retrieving messages and can be used to store messages back into different queues like the error queues.

### 2.4 Logging and metrics

qdo uses `metlog` libraries for logging of events and metrics.

#### 2.4.1 Metrics

**Counter**

The following metrics are sent as incrementing counter events.

**worker.wait_for_jobs** Sent when a worker has no more messages to process and sits idle. Sent once per configured wait period.
Exceptions

By default failed jobs will be logged and skipped. The job failure will be logged using `metlog-raven` including a full Python traceback.

Timer

The following metrics are sent as timing data.

- `worker.job_time` Time for a job to process a single message.
- `worker.job_failure_time` Time to process each job failure.

2.5 API Documentation

Comprehensive reference material for every public API exposed by `qdo` is available within this chapter. The API documentation is organized alphabetically by module name.

2.5.1 `qdo.log`

Exposes exception and metrics logging functions based on `metlog`.

Functions

- `qdo.log.get_logger()` Get a global `metlog` client.
  
  Return type `metlog.client.MetlogClient`

- `qdo.log.configure(settings, debug=False)` Configure a `metlog` client and sender, either based on the passed in `settings` or as a debug sender.

2.5.2 `qdo.partition`

Contains partition helpers.

Classes

- `class qdo.partition.Partition(queuey_conn, name, msgid=None, worker_id='')` Represents a specific partition in a message queue.

  Parameters

  - `queuey_conn (object)` A `Queuey` client instance.
  - `name (unicode)` The queue name (a uuid4 hash) or the combined queue name and partition id, separated by a dash.
  - `msgid (unicode)` The key of the message in the status queue, holding information about the processing state of this partition.
  - `worker_id` An id for the current worker process, used for logging.
last_message
Property for the message id of the last processed message.

messages (limit=100, order='ascending')
Returns messages for the partition, by default from oldest to newest.

Parameters
• limit (int) – Only return N number of messages, defaults to 100
• order (str) – ‘descending’ or ‘ascending’, defaults to ascending

Raises queuey_py.HTTPError
Return type list

2.5.3 qdo.worker
Contains the worker class including the main loop.

Functions
qdo.worker.dict_context()
The default job context manager. It sets context to be a dict.

qdo.worker.log_failure (message, context, queue, exc, queuey_conn)
A simple job failure handler. It logs a full traceback for any failed job using metlog-raven.

qdo.worker.save_failed_message (message, context, queue, exc, queuey_conn)
A job failure handler. It does the same as the log_failure handler and in addition saves a copy of each failed message in a special error queue named qdo_error in Queuey.

Failed messages get a TTL of 30 days to provide some more time for debugging purposes. The failed messages are left in their original queues untouched, but will be purged after the shorter but configurable Queuey default TTL (3 days).

Exceptions

exception qdo.worker.StopWorker
An exception which causes the worker loop to shut down cleanly. Especially useful in writing tests.

Classes

class qdo.worker.Worker (settings)
A Worker works on jobs.

Parameters settings (dict) – Configuration settings

stop ()
Stop the worker loop. Used in an atexit hook.

work ()
Work on jobs.
2.6 Design notes

qdo tries to keep things simple and robust. Some features are intentionally left out, like process management, memory or CPU limitations as there’s dedicated external tools like circus or supervisord for these tasks.

There’s also no process or thread pools implemented inside qdo. Scaling qdo is done via starting multiple qdo worker scripts. Qdo can automatically discover all queues in Queuey and coordinate queue to worker assignment using Zookeeper, so starting new worker instances is automatic and painless.

All actual persistent data is stored inside Queuey (Cassandra) including information on task completion. If Zookeeper is used, it only stores volatile data about the current configuration of the cluster. But if the entire cluster is shut down, all data inside Zookeeper can be removed and will be recreated when the cluster is started up again.

All parts of Queuey and qdo are optimized for a cluster setup of at least three nodes for each component. It’s possible to use a single instance setup for development purposes though. This is reflected in all parts supporting configuration of multiple back-end nodes and random selection of nodes and automatic fail-over amongst nodes. It’s assumed that the exact same on-disk configuration files will be used for configuring multiple nodes (as they are likely put in place by tools like Puppet or Chef). The entire system can transparently handle reconfiguration and re-balancing in case a minority of nodes of any part goes down.

One qdo worker process handles zero up to many partitions of queues at the same time. One specific partition is only ever handled by one worker. This exact assignment is either handled manually via configuration files or is ensured by using distributed locks backed by Zookeeper.

2.6.1 Monitoring (metlog)

qdo uses Mozilla Services metlog libraries to provide logging and metrics gathering. No logging or metric data is kept on local machines, but all data is sent to dedicated logging and metric services.

2.7 Development

Development of qdo takes place on github at: https://github.com/mozilla-services/qdo

2.7.1 Tests

Start a Queuey instance via supervisor:

bin/supervisord

To shut it down:

bin/supervisorctl shutdown

To run the tests call:

make test

If you don’t start supervisor yourself, make test will do so for you.

There’s some services being started during the tests. Currently those are hard-coded to the following ports (on 127.0.0.1):

• 4999 Supervisor
• 5000 Queuey
2.7.2 Helpers

To check for new releases of all Python dependencies, run:

```
bin/checkversions -i http://c.pypi.python.org/simple
```

or if you only want bug-fix (level 2 in 0.1.2) releases:

```
bin/checkversions -l 2 -i http://c.pypi.python.org/simple
```

Choose a PyPi mirror that’s close to you.

2.8 Changelog

2.8.1 0.2 (unreleased)

2.8.2 0.1 (2012-09-17)

• Initial release and feature scope.
INDEX AND GLOSSARY

3.1 Glossary

Cassandra  Apache Cassandra is a distributed NoSQL database, read more at http://cassandra.apache.org/ or http://www.datastax.com/docs/1.1/index.

metlog  Metlog is a system of application logging and metrics gathering developed by the Mozilla Services team. For the Python library see http://metlog-py.readthedocs.org/ or https://github.com/mozilla-services/metlog-py/.

Mozilla Services  The Mozilla Services team builds and operates scalable infrastructure for internal Mozilla projects. See https://wiki.mozilla.org/Services.

Pyramid  Pyramid is a Python based web application framework, read more at http://www.pylonsproject.org/projects/pyramid/about.

Queuey  Yet another message queue. See http://queuey.readthedocs.org/.

resource specification  A resource specification is a string, consisting of a dotted name pointing to a Python module and separated by a colon the name of a function in that module. For example: os.path:join

Zookeeper  Apache Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.
qdo.log, 9
qdo.partition, 9
qdo.worker, 10