queuey Documentation

Release 0.9

Mozilla Foundation

September 18, 2013

CONTENTS

Read the full documentation at http://queuey.readthedocs.org/

Wat? Another message queue?

Given the proliferation of message queue's, one could be inclined to believe that inventing more is not the answer. Using an existing solution was attempted multiple times with most every existing message queue product.

The others failed (for our use-cases).

Queuey is meant to handle some unique conditions that most other message queue solutions either don't handle, or handle very poorly. Many of them for example are written for queues or pub/sub situations that don't require possibly longer term (multiple days) storage of not just many messages but huge quantities of queues.

Queuey Assumptions and Features:

- Messages may persist for upwards of 3 days
- Range scans with timestamps to rewind and re-read messages in a queue
- Millions of queues may be created
- Message delivery characteristics need to be tweakable based on the specific cost-benefit for a Queuey deployment
- HTTP API for easy access by a variety of clients, including AJAX
- Authentication system to support multiple 'Application' access to Queuey with optional Browser-ID client authentication
- A single deployment may support multiple Applications with varying message delivery characteristics, and authentication restricted queue access

Queuey can be configured with varying message guarantees, such as:

- Deliver once, and exactly once
- Deliver at least once (and under rare conditions, maybe more)
- Deliver no more than once (and under rare conditions, possibly not deliver)

Changing the storage back-end and deployment strategies directly affects the message guarantee's. This enables the Queuey deployment to meet different requirements and performance thresholds.

For more background on Queuey, see the Mozilla wiki section on queuey.

CHAPTER

ONE

REFERENCE MATERIAL

Reference material includes documentation for the *queuey* HTTP API.

1.1 Installation

Queuey is composed of two main parts, the Queuey Python web application and the back-end storage layer used by the web application which defaults to Cassandra.

Cassandra can be setup either as a single node, or preferably as a cluster for higher throughput, availability, and durability. The Queuey web application should ideally be installed on separate web nodes to avoid sharing resources with Cassandra but in smaller setups should be fine on the same node.

1.1.1 Installing the Queuey web application

Using the source

Pre-requisites:

- Python >=2.6
- Make

Checkout the source and build using git from Github:

```
git clone git://github.com/mozilla-services/queuey.git
make
```

This will setup a new virtualenv in this directory with all the other tools installed that Queuey needs to run.

1.1.2 Installing Cassandra

It's recommended that Cassandra be installed using the Datastax Cassandra Community Edition as it goes through more testing then the latest open-source version and provides a smooth upgrade path for the Enterprise Edition should one wish to upgrade later. It also comes with support for the Datastax Opscenter to help manage the Cassandra cluster.

There is complete documentation on the Datastax site that explains the installation in more detail, a quick-start is provided here based on those docs.

Before continuing to install Cassandra, you should make sure the machine you're installing to has the necessary pre-requisites:

```
Sun Java Runtime Environment 1.6.0_19 or later
```

You can check to see what version of Java is available by running:

```
java -version
```

Which should print something like:

```
Java(TM) SE Runtime Environment (build 1.6.0_29-b11-402-11D50b)
```

If you're using an OpenJDK Java version, see the Datastax site for Installing Sun JRE on Redhat Systems or Installing Sun JRE on Ubuntu systems.

These directions include installing the opscenter agent, which reports cluster and node information for the opscenter dashboard.

Using the source

If you installed Queuey using `make` above and Cassandra is being installed on the Queuey node, the Makefile includes Cassandra setup:

```
make cassandra
```

If setting up a cluster, or not installing Cassandra on the same node as the Queuey webapp the following directions should be used. Also note that this step does not install opscenter, which also requires the following.

Download the source tarball to the desired directory (first check for newer versions):

```
cd $HOME
mkdir datastax
cd datastax

wget http://downloads.datastax.com/community/dsc-cassandra-1.1.2-bin.tar.gz

# On your first node *only*, get opscenter
wget http://downloads.datastax.com/community/opscenter-2.1.2-free.tar.gz

# Untar the distributions
tar -xzvf dsc-cassandra-1.1.2-bin.tar.gz
tar -xzvf opscenter-2.1.2-free.tar.gz

# Remove the tarballs
rm *.tar.gz

# Create a data/logging directory
mkdir $HOME/datastax/cassandra-data
```

The opscenter package only needs to be installed on a single node, as the opscenter agent for the other nodes will be configured and tar'd up after the setup is run on the main node. This is because the agent.tar.gz that will be created contains SSL authentication information to protect the agents communication.

For more efficient performance, its recommended that JNA be installed to improve memory performance.

1. Download jna.jar from the JNA project site.
2. Add jna.jar to $CASSANDRA_HOME/lib/ or otherwise place it on the CLASSPATH.
3. Edit the file /etc/security/limits.conf, adding the following entries for the user or group that runs Cassandra:

```
    $USER soft memlock unlimited
    $USER hard memlock unlimited
```

Via RPM's

See the Datastax RPM installation instructions.

1.2 Configuration

1.2.1 Cassandra

Cassandra is configured via *cassandra.yaml* and *log4j-server.properties* files. Queuey doesn't have any specific configuration requirements for Cassandra, though availability and durability guarantees depend on appropriate Cassandra settings.

Please refer to the Datastax community edition documentation for further details.

1.2.2 Pyramid

Queuey is implemented on top of the Pyramid web framework. Documentation for configuring WSGI servers and general deployment techniques therefor also apply to Queuey. The Pyramid cookbook contains some advice on a variety of web servers.

The simplest example of a Pyramid pipeline contains of the following:

```
[app:pyramidapp]
use = egg:queuey

[filter:catcherror]
paste.filter_app_factory = mozsvc.middlewares:make_err_mdw

[pipeline:main]
pipeline = catcherror
           pyramidapp
```

1.2.3 Queuey

Queuey is configured via an ini-style file, which is also used to configure general Pyramid settings. This ini file contains a number of sections. The following sections contain Queuey specific settings.

[application_keys]

Contains a mapping of application name to application key. The application key acts as a shared secret between server and client. For example:

```
[application_keys]
app_1 = f25bfb8fe200475c8a0532a9cbe7651e
```

[storage]

Configures the storage for message data.

backend The type of storage, for Cassandra use: *queuey.storage.cassandra.CassandraQueueBackend*

Further settings are dependent on the storage.

[metadata]

Configures the storage for message metadata.

backend The type of storage, for Cassandra use: *queuey.storage.cassandra.CassandraMetadata*

Further settings are dependent on the storage.

Cassandra storage options

The Cassandra storages support the following additional settings:

host A comma separated list of either *host* or *host:port* values specifying the Cassandra servers. Defaults to *localhost:9160*.

username A username used for connecting to Cassandra's Thrift interface.

password A password used for connecting to Cassandra's Thrift interface.

multi_dc A boolean indicating whether or not Cassandra runs in a multi-datacenter environment, defaults to *False*. If enabled, read and write operations default to *LOCAL_QUORUM* instead of *QUORUM*.

create_schema A boolean value indicating if the required Cassandra schema should be automatically created during startup. Defaults to *True*. If enabled the first server in the host list is used to create the schema.

database The name of the keyspace, defaults to *MessageStore* for the storage and *MetadataStore* for the metadata section.

[metlog]

Queuey uses metlog for logging metrics. For detailed information see the metlog docs.

1.3 Queuey API

The Queuey API is documented by URL and valid HTTP methods for each Queuey resource. Arrows around the name indicate variables in the URL.

All calls return JSON, and unless otherwise indicated methods that take input in the body expect form-encoded variables.

1.3.1 Queue Management

Access queue information, create, update, and delete queues. All calls to these methods must include an Authorization header with the application key:

```
Authorization: Application <key here>
```

Calls missing a valid Authorization header or valid Application key will be rejected.

GET **/v1/*{application}***

Path arguments **application** – Application name

Opt. params

- **limit** (*integer*) – Amount of queues to list.
- **offset** (*string*) – A queue name to start with for paginating through the result
- **details** (*boolean*) – Whether additional queue details such as the consistency, type, partitions, and principles for the queue should also be returned. Defaults to false.
- **include_count** (*boolean*) – When including details, should the total message count be included? Defaults to false.

Returns a list of queues for the application. No sorting or ordering for this operation is available, queues are not in any specific ordering but their order is consistent for proper pagination.

Example response:

```
{
    'status': 'ok',
    'queues': [
        {'queue_name': 'a queue'},
        {'queue_name': 'another queue'}
    ]
}
```

Example response with details:

```
{
    'status': 'ok',
    'queues': [
        {
            'queue_name': 'ea2f39c0de9a4b9db6463123641631de',
            'partitions': 1,
            'created': 1322521547,
            'type': 'user',
            'count': 932
        },
        {
            'queue_name': 'another queue',
            'partitions': 4,
            'created': 1325243233,
            'type': 'user',
            'count': 232
        },
    ]
}
```

POST **/v1/*{application}***

Path arguments **application** – Application name

Opt. params

- **queue_name** – Name of the queue to create
- **partitions** (*integer*) – How many partitions the queue should have. Defaults to 1.

- **type** – Type of queue to create, defaults to `user` which requires authentication to access messages.
- **consistency** – Level of consistency for the queue, defaults to `strong`.
- **principles** – List of App or Browser ID's separated with a comma if there's more than one

Create a new queue for the application. Returns a JSON response indicating the status, the UUID4 hex string of the queue name (if a queue_name was not supplied), and the partitions created.

Calling this method with no parameters at all will yield a response like the one below.

Example response:

```
{
    'status': 'ok',
    'queue_name': 'ea2f39c0de9a4b9db6463123641631de',
    'partitions': 1,
    'type': 'user',
    'consistency': 'strong'
}
```

PUT ***/v1/{application}/{queue_name}***

Path arguments

- **application** – Application name
- **queue_name** – Queue name to access

Opt. params

- **partitions** (*integer*) – How many partitions the queue should have.
- **type** – Type of queue to create, 'user' or 'public'.
- **consistency** – Level of consistency for the queue.
- **principles** – List of App or Browser ID's separated with a comma if there's more than one

Update queue parameters. Partitions may only be increased, not decreased. Other settings overwrite existing parameters for the queue, to modify the principles one should first fetch the existing ones, change them as appropriate and PUT the new ones.

Example response:

```
{
    'status': 'ok',
    'queue_name': 'ea2f39c0de9a4b9db6463123641631de',
    'partitions': 1,
    'type': 'user',
    'consistency': 'strong'
}
```

DELETE ***/v1/{application}/{queue_name}***

Path arguments

- **application** – Application name
- **queue_name** – Queue name to access

Delete a queue and all its partitions and messages.

Example success response:

```
{'status': 'ok'}
```

1.3.2 Message Management

Create messages on a queue, get messages, and delete messages. Access varies depending on the queue, queues with a type of `public` may have messages viewed without any authentication. All other queue's require an Application key to create messages, and viewing messages varies depending on queue principles. By default an Application may create/view messages it creates unless a set of principles was registered for the queue.

Responses containing messages include a message *timestamp*. The value is in seconds since the epoch in GMT. It uses a precision down to multiples of 100 nanoseconds. The precision matches that of UUID1s used for the *message_id*. Note that double precision floating point numbers don't guarantee enough significant decimal digits to represent those numbers accurately.

`GET` ***/v1/{application}/{queue_name}***

Path arguments

- **application** – Application name
- **queue_name** – Queue name to access

Opt. params

- **since** – All messages newer than this timestamp *or* message id. Should be formatted as seconds since epoch in GMT, or the hexadecimal message id. For exact results with single message accuracy use the hexadecimal message id.
- **limit** – Only return N amount of messages.
- **order** – Order of messages, can be set to either *ascending* or *descending*. Defaults to *ascending*.
- **partitions** – A specific partition number to retrieve messages from or a comma separated list of partitions. Defaults to retrieving messages from partition 1.

Get messages from a queue. Messages are returned in order of newest to oldest.

Example response:

```
{
    'status': 'ok',
    'messages': [
        {
            'message_id': '3a6592301e0911e190b1002500f0fa7c',
            'timestamp': 1323973966282.637,
            'body': 'jlaijwiel2432532jilj',
            'partition': 1
        },
        {
            'message_id': '3a8553d71e0911e19262002500f0fa7c',
            'timestamp': 1323973966918.241,
            'body': 'ion12oibasdfjioawneilnf',
            'partition': 2
        }
    ]
}
```

`POST` ***/v1/{application}/{queue_name}***

Path arguments

- **application** – Application name
- **queue_name** – Queue name to access

A body containing a single message and optional partition value, or a set of message and partition pairs by number.

When the partition is not specified, the message will be randomly assigned to one of the partitions for a queue (or just the first one if the queue has only one partition).

A TTL can be specified per message, in seconds till it should expire and be unavailable.

Posting a batch of messages (Using JSON)

Include a `Content-Type` HTTP header set to `application/json` with a body like the following:

```
{'messages': [
    {
        'body': 'this is message 1',
        'ttl': 3600,
    },
    {
        'body': 'this is message 2',
        'partition': 3
    }
]}
```

Post an individual message

Any `Content-Type` header will be recorded with the message. The body is assumed to be the entirety of the POST body. The TTL or Partition can be set by including the appropriate value with either `X-TTL` or `X-Partition` HTTP headers in the request.

Example POST as seen by server including both *optional* HTTP headers:

```
POST /v1/notifications/ea2f39c0de9a4b9db6463123641631de HTTP/1.1
Host: site.running.queuey
User-Agent: AwesomeClient
Content-Length: 36
Content-Type: application/text
X-TTL: 3600
X-Partition: 2

A really cool message body to store.
```

Example success response:

```
{
    'status': 'ok',
    'messages' [
        {
            'key': '3a6592301e0911e190b1002500f0fa7c',
            'timestamp': 1323976306.988889,
            'partition': 1
        },
    ]
}
```

GET ***/v1/{application}/{queue_name}/{messages}***

Path arguments

- **application** – Application name

- **queue_name** – Queue name to access
- **messages** – A single hex message id, or comma separated list of hex message id's. To indicate partitions for the messages, prefix the hex message with the partition number and a colon.

Get messages by their message ids from a queue. This API acts as a search, so any message that cannot be found is omitted from the result.

Example response:

```
{
    'status': 'ok',
    'messages': [
        {
            'message_id': '3a6592301e0911e190b1002500f0fa7c',
            'timestamp': 1323973966282.637,
            'body': 'jlaijwiel2432532jilj',
            'partition': 1
        },
        {
            'message_id': '3a8553d71e0911e19262002500f0fa7c',
            'timestamp': 1323973966918.241,
            'body': 'ion12oibasdfjioawneilnf',
            'partition': 2
        }
    ]
}
```

PUT **/v1/*{application}*/*{queue_name}*/*{messages}***

Path arguments

- **application** – Application name
- **queue_name** – Queue name to access
- **messages** – A single hex message id, or comma separated list of hex message id's. To indicate partitions for the messages, prefix the hex message with the partition number and a colon.

Opt. params **X-TTL** – The message's TTL, defaults to three days.

Overwrite existing messages with new data or create new messages. The body is assumed to be the entirety of the PUT body for all messages. There's no support for doing a batch update with different body or TTL values.

Example PUT as seen by server:

```
PUT /v1/my_application/somequeuename/2%38cc967e0cf1e45e3b0d4926c90057caf HTTP/1.1
Host: site.running.queuey
User-Agent: AwesomeClient
Content-Length: 9
Content-Type: application/text
X-TTL: 3600

New text.
```

Example success response:

```
{'status': 'ok'}
```

DELETE **/v1/*{application}*/*{queue_name}*/*{messages}***

Path arguments

- **application** – Application name
- **queue_name** – Queue name to access
- **messages** – A single hex message id, or comma separated list of hex message id's. To indicate partitions for the messages, prefix the hex message with the partition number and a colon.

Delete a message, or multiple messages from a queue. The message ID must be prefixed with the partition number and a colon if the queue has multiple partitions to indicate which one contains the message.

Example of deleting a message from partition 2:

```
# The %3 is a URL encoded colon
DELETE /v1/my_application/somequeuename/2%38cc967e0cf1e45e3b0d4926c90057caf
```

Example success response:

```
{'status': 'ok'}
```

1.4 Changelog

1.4.1 0.9 (unreleased)

1.4.2 0.8 (2012-08-28)

Features

- Compatibility with Cassandra 1.1
- Add new API's to get, post and update messages by their message id
- Add new memory storage backend for testing purposes.
- Add metlog based metrics logging.
- Use pycassa's system manager support to programmatically create the Cassandra schema during startup.

Bug fixes

- Fix precision errors in server side message id to timestamp conversion.
- Enforce message keys to be valid UUID1 instead of just any UUID.

CHAPTER

TWO

SOURCE CODE

All source code is available on github under queuey.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *Glossary*

CHAPTER

FOUR

LICENSE

`queuey` is offered under the MPL 2.0 license.

CHAPTER

FIVE

AUTHORS

`queuey` is made available by the *Mozilla Foundation*.