Queue Constructor

Default Options

```
const Queue = require('rethinkdb-job-queue')
const q = new Queue()
```

Queue Connection

Option	Type	Default
host	String	localhost
port	Integer	28015
db	String	rjqJobQueue

Connection Driver

```
const rethinkdbdash = require('rethinkdbdash')
const Queue = require('rethinkdb-job-queue')

const db01driver = rethinkdbdash({ host: 'db01.domain.com', db: 'JobQueue' })

const q = new Queue(db01driver)
```

Queue Options

Option	Type	Default	Version
name	String	rjqJobList	
databaseInitDelay	Integer	1000 ms (1 sec)	
queryRunOptions	Boolean	{ readMode: ‘majority’ }	
changeFeed	Boolean	true	
concurrency	Integer	1	

Option	Type	Default	Version
masterInterval	Integer	310000 ms (5 min 10 sec)	
limitJobLogs	Integer	1000 log entries	v3.1.0
removeFinishedJobs	Bool / Int	15552000000 ms (180 days)	

Queue databaseInitDelay Option

If ***multiple*** Queue objects are instantiated at the ***same time*** it might cause two databases or tables with the ***same name*** to be created within the same RethinkDB instance.

- To mitigate this issue there is a delay before the databases resources are created.
- This delay comprises of a fixed portion and a random portion.
- The ***databaseInitDelay*** option is exposing the ***fixed portion*** of the initialization delay.
- The ***random portion is hard coded*** and will be between zero and one second.

Queue queryRunOptions Option

If you do change the run option ***readMode*** to ***single*** and you experience the same job being processed by multiple Queue workers, change it back to the default majority value.

Queue changeFeed Option

If enabled the changeFeed option will cause the Queue object to raise events for the entire queue rather than just the local Queue object.
Once the changes are received by the Queue object, they are checked to ensure they are not local changes.
If the changes are detected as local changes (a change made by self) they are ignored because the events will be raised by the Queue object.
If the changes are not local then the relevant event is raised.

Local Event Example

```
const Queue = require('rethinkdb-job-queue')

const qOptions = {
  changeFeed: false
}
```

```
const q = new Queue(null, qOptions)

q.on('completed', (queueId, jobId, isRepeating) => {
  console.log('Job completed: ' + jobId)
})
```

Global Event Example

```
const Queue = require('rethinkdb-job-queue')

const qOptions = {
  changeFeed: true // This is the default and is not required
}

const q = new Queue(null, qOptions)

q.on('completed', (queueId, jobId, isRepeating) => {
  console.log('Job completed: ' + jobId)
})
```

Queue masterInterval Option

The ***masterInterval*** option determines how often the ***Queue Master*** review process is carried out against the queue.

`Warning:` Setting the masterInterval value too low will cause excessive database load. Every time this period lapses queries are run against the database. Keep this value as high as possible.

Queue Master

The Queue Master role in rethinkdb-job-queue is an integral role to **ensure delayed and failed jobs get processed and the database is cleaned**.

When the ***time period elapses***, the Queue Master will ***review*** the database table backing the queue. This is called the **Queue Review process**.

A Queue Master will perform four tasks within the job queue during the Queue Review process:

- ***Failed Node.js Process***

  Discover and enable jobs that have failed due to the Node.js process crashing or hanging.

- ***Remove Finished Jobs***

  Remove completed, cancelled, or terminated jobs from the queue.
- ***Delayed Job Processing***

  Enable processing of delayed jobs or failed jobs waiting for retry.
- ***Update Queue State***

  At the completion of the review process the queue State Document will be updated.

Failed Node.js Process

During normal queue operation, Queue objects processing jobs will ***detect when a job has taken too long and is operating past its timeout value.*** If this situation occurs the job status in the database is set to failed and the job will be delayed based on the ***retryDelay***, ***retryCount***, and ***retryMax*** values.

However, if a Node.js process ***fails for any reason*** whilst working on a job, ***the job will not be completed and will remain in the database with an active status causing an orphaned job***.

To ensure the job is not forgotten, ***a Queue Master will repeatedly review the queue database backing table based on the masterInterval.*** When the Queue Master reviews the queue backing table, it looks for jobs that have a status of active and are past their ***dateEnable*** value. **The *dateEnable* value is set when the job is created or when it is retrieved from the database for processing.**

The queue review process will update the job status based on the *retryCount* and *retryMax* values

`NOTE:` If the jobs ***retryCount*** value is less than the ***retryMax*** then the job status will be set to ***'failed'*** and the ***retryCount*** value will be incremented. This job will now be ***ready for processing.***

`NOTE:` If the jobs ***retryCount*** value is equal to the ***retryMax*** value then the job status will be set to terminated and the job is considered finished.

> *It is possible for normal job being processed to extend past its initial timeout value and be marked as failed by the Queue Master review process. To prevent this, call the Job.progress method on the Job object. When progress for a job is updated, the dateEnable value and the timeout process also get updated. Therefore calling Job.progress periodically within the job timeout period will prevent the job from erroneously being marked as failed on review.*

Remove Finished Jobs

In this context a ***finished job*** is defined as a job in the queue that has a status of either ***completed, cancelled, or terminated.***

NOTE: Once a job has finished processing it will no longer be an active part of the queue. The job details in the database including its log entries and other properties are just taking up space.

Now if you are processing thousands of jobs a day this might not be a big deal and you may very well be happy to just leave the job details in the database for future reference. However if you are processing millions of jobs a day, the space taken up by the completed jobs could add up over a year or more. If that is the case then you will want to remove finished jobs from the database to free up space.

Fortunately Queue objects have three options for cleaning up jobs once they are finished based on the ***removeFinishedJobs*** Queue Option.

Two of the values you can set the ***removeFinishedJobs*** Queue option to will be ignored by the Queue Master review process: ***true or false.***

If you set the ***removeFinishedJobs*** option to ***true***, finished jobs will be ***removed from the database immediately.***

If you set the ***removeFinishedJobs*** option to ***false***, jobs will ***never be removed from the database*** no matter what their status is.

The third value you can assign to the ***removeFinishedJobs*** Queue option is a ***positive Integer***. This number ***represents a time period in milliseconds.***

Jobs will be considered ***eligible to be removed*** when their ***dateFinished property is older than the dateFinished plus removeFinishedJobs resultant date.***

The Queue Master review process will ***permanently remove these jobs from the queue.***

Setting the ***removeFinishedJobs*** value to a ***low number*** such as 7 days (in milliseconds) would give you enough time to use the job logs to help you ***debug issues*** while still keeping your queue database clean.

Alternatively, setting ***removeFinishedJobs*** value to a ***high number*** such as 365 days (in milliseconds) would give you plenty of data ***for analysis***.

NOTE: Please consider disabling the ***removeFinishedJobs*** process if you can. It can always be enabled at a later date.

Delayed Job Processing

`Important:` The following is only valid if the Queue Master Queue object has a process handler assigned. If it does not, the Update Queue State task below will enable delayed job processing.

In a busy queue the database will be queried upon completion of jobs in order to find more jobs that need processing. This includes ***finding jobs with a status of waiting or failed*** with the current date after the job ***dateEnable*** value.

`NOTE:` If the last job in the queue ***fails*** and the ***retryDelay*** value is ***not 0***, the job will be delayed for retry and the queue will enter an idle state. There may be other jobs delayed in the queue also.

Without something initiating the queue to process jobs, the last job will remain in the database until more jobs are added to the queue.

To prevent this situation from delaying the last job well beyond its dateEnable value, the Queue Master database review process calls the queue process task. The queue process task will query the database discovering the delayed jobs and retrieve them for processing. Again, this is only if the process handler is populated on the Queue Master.

Update Queue State

`NOTE:` Finally, at the completion of the review process the Queue Master will update the State Document to a state of reviewed. This is an important change in a distributed processing queue environment.

If the queue is currently quiet with no jobs being processed, there is nothing to prompt the Queue objects to go to work. Whilst time is passing some jobs in the queue may become available for processing due to their ***dateEnable*** value. As soon as the current date has past the jobs ***dateEnable*** date, the job is ready for processing.

To remedy this situation and to initiate processing of delayed jobs, the Queue Master review process completes by changing the State Document. This change is detected by all Queue objects connected to the same queue. If a Queue object detects a state update defined as reviewed, it will initiate a process restart function to query the database for more work.

There are three state changes that occur within the queue:

- **paused**
- **active**
- **reviewed**

If a Queue object receives a ***paused*** global state update, it will change its own status to paused. This enables any Queue object to globally pause the queue.

If a Queue object receives an ***active*** global state update, it will resume processing if paused. This enables any Queue object to globally resume the queue processing.

When a Queue Master completes the queue ***review*** process it updates the State Document to a state of ***reviewed***. All Queue objects connected to the queue will restart processing if their running job count is less than their concurrency value.

This example shows how to cancel more than one job.

```
const Queue = require('rethinkdb-job-queue')
const q = new Queue()
const jobs = [q.createJob(), q.createJob()]
// The jobs array will contain 2 jobs
// Don't forget to decorate your jobs with job data

q.addJob(jobs).then((savedJobs) => {
  // savedJobs is an array of a 2 job objects
  return q.cancelJob(savedJobs)
}).then((cancelledJobIds) => {
  // If you need the cancelled job ids for any reason, here they are
  console.dir(cancelledJobIds)
}).catch(err => console.error(err))
```

This example shows how to cancel one job.

```
const Queue = require('rethinkdb-job-queue')
const q = new Queue()
const job = q.createJob()
// Decorate your job with job data for processing here

q.addJob(job).then((savedJobs) => {
  // savedJobs is an array of a single job object
  return q.cancelJob(savedJobs)
}).then((cancelledJobIds) => {
  // If you need the cancelled job id for any reason, here it is
```

```
  console.dir(cancelledJobIds)
}).catch(err => console.error(err))
```

IN PROCESS FILE

```
q.process((job, next, onCancel) => {
    onCancel(job, () => {
        //code
    })
})
```

Job Options

Option	Type	Default
name	String	uuid same as the Job id
priority	String	normal
timeout	Integer	300000 ms (5 min)
retryMax	Integer	3
retryDelay	Integer	600000 ms (10 min)
repeat	Bool/Int	false
repeatDelay	Integer	300000 ms (5 min)
dateEnable	Date	new Date()

Methods

- Job.setName
- Job.setPriority
- Job.setTimeout
- Job.setRetryMax
- Job.setRetryDelay
- Job.setDateEnable
- Set the property directly on the job

Job Schema

`dateCreated:` The date is added when the job is created and should never change.

`dateEnable:` Jobs will not be processed until the current date is past this date.

`dateFinished:` Records the date when a job completes, fails, gets cancelled, or is terminated.

`dateStarted:` The date the job processing was last started. It may be a retry start date.

`id:` The document id within the RethinkDB database.

`log:` An array of log entries created from within the Queue object or by using Job.addLog.

`name:` A string value used to identify the job. See the Job Name document for more detail.

`priority:` A number representing the processing order of the jobs.

`processCount:` The number of times the job has been processed or attempted to be processed.

`progress:` The user reported progress for the job. Updated with Job.updateProgress.

`queueId:` A string representing the Queue object that last updated the job in the database.

`repeat:` Whether the job will repeat.

`repeatDelay:` The delay between repeats.

`retryCount:` A record of the number of retry attempts the job has had.

`retryDelay:` The time delay in milliseconds before failed jobs will be retried.

`retryMax:` The maximum number of retries before the job is terminated.

`status:` The current status of the job.

`timeout:` How long the job can be processed for before it is considered failed.

Job name Option

`Warning:` Setting the job name is not guaranteed to be unique. Use the Job.id for unique identification.

The benefit of the name property is that it is backed by an index within the database.

This makes finding named jobs and testing job existence faster with less load on the database.

```
let job = q.createJob().setName('Batman')
```

Job priority Option

Name	Number	Description
lowest	60	The last jobs to get processed
low	50	Not as important as a typical job

Name	Number	Description
normal	40	The default job priority
medium	30	For more important jobs
high	20	Need to be processed first
highest	10	These jobs will be processed before all others

```
let job = q.createJob().setPriority('high')
```

Job timeout Option

The job ***timeout*** option is for defining the maximum time a job should be processed for before getting classified as ***failed***.

`Note:` There are two ways a job will be considered a failed job using this timeout option; if the Queue object **job processing takes too long and runs past the timeout value**, or if the **nodejs process hangs or crashes.**

```
let job = q.createJob().setTimeout(600000)
```

Job retryMax Option

If the ***retryMax*** value of a job is greater than zero, then it will be retried if the job processing fails.

Setting the ***retryMax*** value to ***0*** will disable retries.

Setting the ***retryMax*** option to ***n*** will mean the job will get processed ***n+1*** times. As above, once for the first try, then ***n*** retries.

```
let job = q.createJob().setRetryMax(1)
```

Job retryDelay Option

The ***retryDelay*** option allows you to progressively delay the job processing on successive retries.

```
let job = q.createJob().setRetryDelay(1000000)
```

Job repeat Option

The ***repeat*** option allows you to repeat processing of a job.

```
let job = q.createJob().setRepeat(3)
```

Job repeatDelay Option

The ***repeatDelay*** option allows you to delay repeat processing of a job.

e.g.

The below example sets the repeat value to 3 and updates the repeatDelay to 24 hours. This will cause the job to be processed a total of 4 times; once for the original processing with three repeats with a 24 hour delay between:

```
let job = q.createJob().setRepeat(3).setRepeatDelay(86400000)
```

Job dateEnable Option

The job will be processed only after the current date is past the ***dateEnable*** value.

We don't need to change this option if we are happy for the job to be processed as soon as possible. If we wish to delay the job processing until a future date, change this option.

```
let job = q.createJob().setDateEnable(new Date() + 1000000)
```

Job Retry

When creating job objects we can configure the ***timeout***, ***retryMax***, and ***retryDelay*** options. These options will determine what will happen if a job fails either by a Node.js process not responding or the job process failing.

Every job has a property called ***dateEnable*** which is used to determine if the job is ready for processing. The ***dateEnable*** value is set when the job is created and when it is retrieved from the database for processing. The retrieval query will not return jobs where the dateEnable value is greater than the current date time value.

Currently the formula used to set the ***dateEnable*** value during the job retrieval process is:

```
now() + job.timeout + ( job.retryDelay * job.retryCount )
```

The plan in the future is to move this to an exponential formula once RethinkDB has a power function.

`Note:` In the below step through I am stating that the queue review process is re-activating the jobs after failure. This may not be the case with an active queue. If the Queue object is constantly working on jobs then the jobs will be activated as soon as the time is right.

If we take the job properties and the Queue Master ***'masterInterval'*** value, then the following sequence of events will occur:

1. The job has never been processed and has default properties. It has been added to the queue.

   ```
   status = 'waiting'
   timeout = 300000
   retryCount = 0
   retryMax = 3
   retryDelay = 600000
   dateEnable = dateCreated
   ```

2. The job is retrieved from the database setting the dateEnable value.

   ```
   status = 'active'
   timeout = 300000
   retryCount = 0
   retryMax = 3
   retryDelay = 600000
   dateEnable = now() + timeout
   ```

3. The job fails for some reason.

   ```
   status = 'failed'
   timeout = 300000
   retryCount = 1
   retryMax = 3
   retryDelay = 600000
   dateEnable = now() + (retryDelay * retryCount)
   ```

4. The job remains inactive within the database until after dateEnable or approximately 600000 milliseconds (i.e. retryDelay * retryCount).
5. The Queue Master database review is initiated and the job is retrieved from the database for the first retry.

   ```
   status = 'active'
   timeout = 300000
   retryCount = 1
   retryMax = 3
   ```

```
retryDelay = 600000
dateEnable = now() + timeout + (retryDelay * retryCount)
```

6. The job fails again for some reason.

```
status = 'failed'
timeout = 300000
retryCount = 2
retryMax = 3
retryDelay = 600000
dateEnable = now() + (retryDelay * retryCount)
```

7. The job remains inactive within the database until after dateEnable or approximately 1200000 milliseconds (i.e. retryDelay * retryCount).
8. The Queue Master database review is initiated and the job is retrieved from the database for the second retry.

```
status = 'active'
timeout = 300000
retryCount = 2
retryMax = 3
retryDelay = 600000
dateEnable = now() + timeout + (retryDelay * retryCount)
```

9. The job fails again. What is wrong with this job?

```
status = 'failed'
timeout = 300000
retryCount = 3
retryMax = 3
retryDelay = 600000
dateEnable = now() + (retryDelay * retryCount)
```

10. The job remains inactive within the database until after dateEnable or approximately 1800000 milliseconds.
11. The Queue Master database review is initiated and the job is retrieved from the database for the third and final retry.

```
status = 'active'
timeout = 300000
retryCount = 3
retryMax = 3
retryDelay = 600000
dateEnable = now + timeout + (retryDelay * retryCount)
```
this is redundant however still set

12. The job fails for the last time.

```
status = 'terminated'
```

13. Because the job status is set to terminated it will no longer be retrieved from the database.

Job Repeat

When we create a Job object we can set the ***repeat*** and the ***repeatDelay*** options. These allow us to repeat the processing of jobs either for ***a fixed number of times or continuously.***

By setting the ***repeat*** option to ***true*** the job will continually repeat after waiting for the ***repeatDelay*** time period. Alternatively, you can set the ***repeat*** option to a ***number*** and the job will be processed and then repeated the correct number of times.

If a job fails processing and has the ***retry*** options enabled, the job will follow the normal retry procedure. If the retries fail instead of transitioning to a terminated status, the job goes into a waiting status and the ***retryDelay*** will apply.

`Note:` If the queue is not working hard the repeated processing of jobs will require the Queue Master review process to begin processing the job after the Job.repeatDelay time period.

`Note:` Consider setting the Queue option limitJobLogs to a smaller number than the default 1000 log entries.

`Warning:` Do not set the retryDelay value too low unless the repeated job takes a long time to run.

Job Status

Status	Description
created	After creating the job
waiting	After being added to the queue
active	Whilst the job is being processed
completed	When the job has been processed successfully
cancelled	If you cancel a job
failed	When the job has failed being processed
terminated	When a job has failed and will not be retried

Job Editing

One of the most powerful and flexible methods within rethinkdb-job-queue is the **Job.update** method.

It allows us, without restriction, to change any value of a job and save those changes back to the queue database.

Job Data Changes

If for some reason we need to change the data associated with a job we can do so with the following code.

For this example we need to change the job.data property from foo to bar.

Assumptions:

- The job is in the queue.
- The job has not been processed.
- The job.data property is set to foo.
- We have the job id.

```
q.getJob('5127d082-fe7e-4e88-b1de-7093029695c3').then((savedJobs) => {
  savedJobs[0].data = 'bar'
  return savedJobs[0].update()
}).catch(err => console.error(err))
```

Removing Job Data

If for some reason we had extra data saved with a job and we need it removed for some security reason. We could do so easily enough as this example shows.

Assumptions:

- The job is in the queue.
- The job.secret property is set.
- We have the job id.

```
q.getJob('5127d082-fe7e-4e88-b1de-7093029695c3').then((savedJobs) => {
  delete savedJobs[0].secret
  return savedJobs[0].update()
}).catch(err => console.error(err))
```

Disable Job

If we had a job waiting for processing and we need to disable the job until further notice, here is one way we could achieve this.

Assumptions:

- The job is in the queue.
- The job has not been processed.
- We have the job id.

```
q.getJob('5127d082-fe7e-4e88-b1de-7093029695c3').then((savedJobs) => {
  delete savedJobs[0].status = 'disabled'
  return savedJobs[0].update()
}).catch(err => console.error(err))
```

> *The example above is setting the Job.status property to disabled which is an invalid status. By setting the status to an invalid value, the internal getNextJob() method will not retrieve the job; hence it is disabled.*

Queue.summary

```
q.summary().then((summary) => {
  console.dir(summary)
}).catch(err => console.error(err))
```

This is a simple queue summary of job status numbers stored in the queue backing table. The resulting object will look like this:

```
{ waiting: 11,
  active: 30,
  completed: 9721,
  cancelled: 4,
  failed: 1,
  terminated: 0,
  total: 9767 }
```