Fork me on GitHub

A *job* is a Python object, representing a function that is invoked asynchronously in a worker (background) process. Any Python function can be invoked asynchronously, by simply pushing a reference to the function and its arguments onto a queue. This is called *enqueueing*.

Enqueueing Jobs

To put jobs on queues, first declare a function:

```
import requests

def count_words_at_url(url):
    resp = requests.get(url)
    return len(resp.text.split())
```

Noticed anything? There's nothing special about this function! Any Python function call can be put on an RQ queue.

To put this potentially expensive word count for a given URL in the background, simply do this:

```
from rq import Queue
from redis import Redis
from somewhere import count_words_at_url
import time

# Tell RQ what Redis connection to use
redis_conn = Redis()
q = Queue(connection=redis_conn)  # no args implies the default queue

# Delay execution of count_words_at_url('http://nvie.com')
job = q.enqueue(count_words_at_url, 'http://nvie.com')
print(job.result)   # => None  # Changed to job.return_value() in RQ >= 1.12.0

# Now, wait a while, until the worker is finished
time.sleep(2)
print(job.result)   # => 889  # Changed to job.return_value() in RQ >= 1.12.0
```

If you want to put the work on a specific queue, simply specify its name:

```
q = Queue('low', connection=redis_conn)
q.enqueue(count_words_at_url, 'http://nvie.com')
```

Notice the `Queue('low')` in the example above? You can use any queue name, so you can quite flexibly distribute work to your own desire. A common naming pattern is to name your queues after priorities (e.g. `high`, `medium`, `low`).

In addition, you can add a few options to modify the behaviour of the queued job. By default, these are popped out of the kwargs that will be passed to the job function.

- `job_timeout` specifies the maximum runtime of the job before it's interrupted and marked as `failed`. Its default unit is second and it can be an integer or a string representing an integer(e.g. `2`, `'2'`). Furthermore, it can be a string with specify unit including hour, minute, second(e.g. `'1h'`, `'3m'`, `'5s'`).
- `result_ttl` specifies how long (in seconds) successful jobs and their results are kept. Expired jobs will be automatically deleted. Defaults to 500 seconds.
- `ttl` specifies the maximum queued time (in seconds) of the job before it's discarded. This argument defaults to `None` (infinite TTL).
- `failure_ttl` specifies how long failed jobs are kept (defaults to 1 year)
- `depends_on` specifies another job (or list of jobs) that must complete before this job will be queued.
- `job_id` allows you to manually specify this job's `job_id`
- `at_front` will place the job at the *front* of the queue, instead of the back
- `description` to add additional description to enqueued jobs.
- `on_success` allows you to run a function after a job completes successfully
- `on_failure` allows you to run a function after a job fails
- `on_stopped` allows you to run a function after a job is stopped
- `args` and `kwargs`: use these to explicitly pass arguments and keyword to the underlying job function. This is useful if your function happens to have conflicting argument names with RQ, for example `description` or `ttl`.

In the last case, if you want to pass `description` and `ttl` keyword arguments to your job and not to RQ's enqueue function, this is what you do:

```
q = Queue('low', connection=redis_conn)
q.enqueue(count_words_at_url,
          ttl=30,  # This ttl will be used by RQ
          args=('http://nvie.com',),
          kwargs={
              'description': 'Function description', # This is passed on to count_words_at_url
              'ttl': 15  # This is passed on to count_words_at_url function
          })
```

For cases where the web process doesn't have access to the source code running in the worker (i.e. code base X invokes a delayed function from code base Y), you can pass the function as a string reference, too.

```
q = Queue('low', connection=redis_conn)
q.enqueue('my_package.my_module.my_func', 3, 4)
```

Bulk Job Enqueueing

New in version 1.9.0.
You can also enqueue multiple jobs in bulk with `queue.enqueue_many()` and `Queue.prepare_data()`:

```
jobs = q.enqueue_many(
  [
    Queue.prepare_data(count_words_at_url, ('http://nvie.com',), job_id='my_job_id'),
    Queue.prepare_data(count_words_at_url, ('http://nvie.com',), job_id='my_other_job_id'),
  ]
)
```

which will enqueue all the jobs in a single redis `pipeline` which you can optionally pass in yourself:

```
with q.connection.pipeline() as pipe:
  jobs = q.enqueue_many(
    [
      Queue.prepare_data(count_words_at_url, ('http://nvie.com',), job_id='my_job_id'),
      Queue.prepare_data(count_words_at_url, ('http://nvie.com',), job_id='my_other_job_id'),
    ],
    pipeline=pipe
  )
  pipe.execute()
```

`Queue.prepare_data` accepts all arguments that `Queue.parse_args` does.

Grouping jobs

New in version 2.0.
Multiple jobs can be added to a Group to allow them to be tracked by a single ID:

```
from rq import Queue
from rq.group import Group

group = Group.create(connection=redis_conn)
jobs = group.enqueue_many(
  queue="my_queue",
  [
    Queue.prepare_data(count_words_at_url, ('http://nvie.com',), job_id='my_job_id'),
    Queue.prepare_data(count_words_at_url, ('http://nvie.com',), job_id='my_other_job_id'),
  ]
)
```

You can then access jobs by calling the group's `get_jobs()` method:

```
print(group.get_jobs())  # [Job('my_job_id'), Job('my_other_job_id')]
```

Existing groups can be fetched from Redis:

```
from rq.group import Group
group = Group.fetch(id='my_group', connection=redis_conn)
```

If all of a group's jobs expire or are deleted, the group is removed from Redis.

Job dependencies

RQ allows you to chain the execution of multiple jobs. To execute a job that depends on another job, use the `depends_on` argument:

```
q = Queue('low', connection=my_redis_conn)
report_job = q.enqueue(generate_report)
q.enqueue(send_report, depends_on=report_job)
```

Specifying multiple dependencies are also supported:

```
queue = Queue('low', connection=redis)
foo_job = queue.enqueue(foo)
bar_job = queue.enqueue(bar)
baz_job = queue.enqueue(baz, depends_on=[foo_job, bar_job])
```

The ability to handle job dependencies allows you to split a big job into several smaller ones. By default, a job that is dependent on another is enqueued only when its dependency finishes *successfully*.

New in 1.11.0.

If you want a job's dependencies to execute regardless if the job completes or fails, RQ provides the `Dependency` class that will allow you to dictate how to handle job failures.

The `Dependency(jobs=...)` parameter accepts:

- a string representing a single job id
- a Job object
- an iteratable of job id strings and/or Job objects
- `enqueue_at_front` boolean parameter to put dependents at the front when they are enqueued

Example:

```
from redis import Redis
from rq.job import Dependency
from rq import Queue

queue = Queue(connection=Redis())
job_1 = queue.enqueue(div_by_zero)
dependency = Dependency(
```

```
    jobs=[job_1],
    allow_failure=True,    # allow_failure defaults to False
    enqueue_at_front=True  # enqueue_at_front defaults to False
)
job_2 = queue.enqueue(say_hello, depends_on=dependency)

"""
  job_2 will execute even though its dependency (job_1) fails,
  and it will be enqueued at the front of the queue.
"""
```

Job Callbacks

New in version 1.9.0.

If you want to execute a function whenever a job completes, fails, or is stopped, RQ provides `on_success`, `on_failure`, and `on_stopped` callbacks.

```
queue.enqueue(say_hello, on_success=report_success, on_failure=report_failure, on_stopped=report_stopped)
```

Callback Class and Callback Timeouts

New in version 1.14.0

RQ lets you configure the method and timeout for each callback - success, failure, and stopped.
To configure callback timeouts, use RQ's `Callback` object that accepts `func` and `timeout` arguments. For example:

```
from rq import Callback
queue.enqueue(say_hello,
              on_success=Callback(report_success),  # default callback timeout (60 seconds)
              on_failure=Callback(report_failure, timeout=10), # 10 seconds timeout
              on_stopped=Callback(report_stopped, timeout="2m")) # 2 minute timeout
```

You can also pass the function as a string reference: `Callback('my_package.my_module.my_func')`

Success Callback

Success callbacks must be a function that accepts `job`, `connection` and `result` arguments. Your function should also accept `*args` and `**kwargs` so your application doesn't break when additional parameters are added.

```
def report_success(job, connection, result, *args, **kwargs):
    pass
```

Success callbacks are executed after job execution is complete, before dependents are enqueued. If an exception happens when your callback is executed, job status will be set to `FAILED` and dependents won't be enqueued.

Callbacks are limited to 60 seconds of execution time. If you want to execute a long running job, consider using RQ's job dependency feature instead.

Failure Callbacks

Failure callbacks are functions that accept `job`, `connection`, `type`, `value` and `traceback` arguments. `type`, `value` and `traceback` values returned by sys.exc_info(), which is the exception raised when executing your job.

```
def report_failure(job, connection, type, value, traceback):
    pass
```

Failure callbacks are limited to 60 seconds of execution time.

Stopped Callbacks

Stopped callbacks are functions that accept `job` and `connection` arguments.

```
def report_stopped(job, connection):
  pass
```

Stopped callbacks are functions that are executed when a worker receives a command to stop a job that is currently executing. See Stopping a Job.

CLI Enqueueing

New in version 1.10.0.

If you prefer enqueueing jobs via the command line interface or do not use python you can use this.

Usage:

```
rq enqueue [OPTIONS] FUNCTION [ARGUMENTS]
```

Options:

- `-q, --queue [value]` The name of the queue.
- `--timeout [value]` Specifies the maximum runtime of the job before it is interrupted and marked as failed.
- `--result-ttl [value]` Specifies how long successful jobs and their results are kept.
- `--ttl [value]` Specifies the maximum queued time of the job before it is discarded.
- `--failure-ttl [value]` Specifies how long failed jobs are kept.
- `--description [value]` Additional description of the job
- `--depends-on [value]` Specifies another job id that must complete before this job will be queued.
- `--job-id [value]` The id of this job
- `--at-front` Will place the job at the front of the queue, instead of the end
- `--retry-max [value]` Maximum number of retries
- `--retry-interval [value]` Interval between retries in seconds
- `--schedule-in [value]` Delay until the function is enqueued (e.g. 10s, 5m, 2d).
- `--schedule-at [value]` Schedule job to be enqueued at a certain time formatted in ISO 8601 without timezone (e.g. 2021-05-27T21:45:00).
- `--quiet` Only logs errors.

Function:

There are two options:

- Execute a function: dot-separated string of package, module and function (Just like passing a string to `queue.enqueue()`).
- Execute a python file: dot-separated pathname of the file. Because it is technically an import `__name__ == '__main__'` will not work.

Arguments:

	plain text	json	literal-eval
keyword	`[key]=[value]`	`[key]:=[value]`	`[key]%=[value]`
no keyword	`[value]`	`:[value]`	`%[value]`

Where `[key]` is the keyword and `[value]` is the value which is parsed with the corresponding parsing method.

If the first character of `[value]` is `@` the subsequent path will be read.

Examples:

- `rq enqueue path.to.func abc` -> `queue.enqueue(path.to.func, 'abc')`
- `rq enqueue path.to.func abc=def` -> `queue.enqueue(path.to.func, abc='def')`
- `rq enqueue path.to.func ':{"json": "abc"}'` -> `queue.enqueue(path.to.func, {'json': 'abc'})`
- `rq enqueue path.to.func 'key:={"json": "abc"}'` -> `queue.enqueue(path.to.func, key={'json': 'abc'})`

- `rq enqueue path.to.func '%1, 2'` -> `queue.enqueue(path.to.func, (1, 2))`
- `rq enqueue path.to.func '%None'` -> `queue.enqueue(path.to.func, None)`
- `rq enqueue path.to.func '%True'` -> `queue.enqueue(path.to.func, True)`
- `rq enqueue path.to.func 'key%=(1, 2)'` -> `queue.enqueue(path.to.func, key=(1, 2))`
- `rq enqueue path.to.func 'key%={"foo": True}'` -> `queue.enqueue(path.to.func, key={"foo": True})`
- `rq enqueue path.to.func @path/to/file` -> `queue.enqueue(path.to.func, open('path/to/file', 'r').read())`
- `rq enqueue path.to.func key=@path/to/file` -> `queue.enqueue(path.to.func, key=open('path/to/file', 'r').read())`
- `rq enqueue path.to.func :@path/to/file.json` -> `queue.enqueue(path.to.func, json.loads(open('path/to/file.json', 'r').read()))`
- `rq enqueue path.to.func key:=@path/to/file.json` -> `queue.enqueue(path.to.func, key=json.loads(open('path/to/file.json', 'r').read()))`

Warning: Do not use plain text without keyword if you do not know what the value is. If the value starts with `@`, `:` or `%` or includes `=` it would be recognised as something else.

Working with Queues

Besides enqueuing jobs, Queues have a few useful methods:

```
from rq import Queue
from redis import Redis

redis_conn = Redis()
q = Queue(connection=redis_conn)

# Getting the number of jobs in the queue
# Note: Only queued jobs are counted, not including deferred ones
print(len(q))

# Retrieving jobs
queued_job_ids = q.job_ids # Gets a list of job IDs from the queue
queued_jobs = q.jobs # Gets a list of enqueued job instances
job = q.fetch_job('my_id') # Returns job having ID "my_id"

# Emptying a queue, this will delete all jobs in this queue
q.empty()

# Deleting a queue
q.delete(delete_jobs=True) # Passing in `True` will remove all jobs in the queue
# queue is now unusable. It can be recreated by enqueueing jobs to it.
```

On the Design

With RQ, you don't have to set up any queues upfront, and you don't have to specify any channels, exchanges, routing rules, or whatnot. You can just put jobs onto any queue you want. As soon as you enqueue a job to a queue that does not exist yet, it is created on the fly.

RQ does *not* use an advanced broker to do the message routing for you. You may consider this an awesome advantage or a handicap, depending on the problem you're solving.

Lastly, it does not speak a portable protocol, since it depends on pickle to serialize the jobs, so it's a Python-only system.

The delayed result

When jobs get enqueued, the `queue.enqueue()` method returns a `Job` instance. This is nothing more than a proxy object that can be used to check the outcome of the actual job.

For this purpose, it has a convenience `result` accessor property, that will return `None` when the job is not yet finished, or a non-`None` value when the job has finished (assuming the job *has* a return value in the first place, of course).

The `@job` decorator

If you're familiar with Celery, you might be used to its `@task` decorator. Starting from RQ >= 0.3, there exists a similar decorator:

```
from rq.decorators import job

@job('low', connection=my_redis_conn, timeout=5)
def add(x, y):
    return x + y

job = add.delay(3, 4)
time.sleep(1)
print(job.return_value())
```

Bypassing workers

For testing purposes, you can enqueue jobs without delegating the actual execution to a worker (available since version 0.3.1). To do this, pass the `is_async=False` argument into the Queue constructor:

```
>>> q = Queue('low', is_async=False, connection=my_redis_conn)
>>> job = q.enqueue(fib, 8)
>>> job.result
21
```

The above code runs without an active worker and executes `fib(8)` synchronously within the same process. You may know this behaviour from Celery as `ALWAYS_EAGER`. Note, however, that you still need a working connection to a redis instance for storing states related to job execution and completion.

The worker

To learn about workers, see the workers documentation.

Suspending and Resuming

Sometimes you may want to suspend RQ to prevent it from processing new jobs. A classic example is during the initial phase of a deployment script or in advance of putting your site into maintenance mode. This is particularly helpful when you have jobs that are relatively long-running and might otherwise be forcibly killed during the deploy.

The `suspend` command stops workers on *all* queues (in a single Redis database) from picking up new jobs. However currently running jobs will continue until completion.

```
# Suspend indefinitely
rq suspend

# Suspend for a specific duration (in seconds) then automatically
# resume work again.
rq suspend --duration 300

# Resume work again.
rq resume
```

Considerations for jobs

Technically, you can put any Python function call on a queue, but that does not mean it's always wise to do so. Some things to consider before putting a job on a queue:

- Make sure that the function's `__module__` is importable by the worker. In particular, this means that you cannot enqueue functions that are declared in the `__main__` module.
- Make sure that the worker and the work generator share *exactly* the same source code.

- Make sure that the function call does not depend on its context. In particular, global variables are evil (as always), but also *any* state that the function depends on (for example a “current” user or “current” web request) is not there when the worker will process it. If you want work done for the “current” user, you should resolve that user to a concrete instance and pass a reference to that user object to the job as an argument.

Limitations

RQ workers will only run on systems that implement `fork()`. Most notably, this means it is not possible to run the workers on Windows without using the Windows Subsystem for Linux and running in a bash shell.

RQ is written by Vincent Driessen.

It is open sourced under the terms of the BSD license.