hirefire Documentation

Release 0.5

Jannis Leidel and contributors

January 21, 2017

Contents

This is a Python package for HireFire – The Heroku Process Manager:

> HireFire has the ability to automatically scale your web and worker dynos up and down when necessary. When new jobs are queued in to your application's worker queue [..], HireFire will spin up new worker dynos to process these jobs. When the queue is empty, HireFire will shut down the worker dynos again so you're not paying for idle workers.
>
> HireFire also has the ability to scale your web dynos. When your web application experiences heavy traffic during certain times of the day, or if you've been featured somewhere, chances are your application's backlog might grow to a point that your web application will run dramatically slow, or even worse, it might result in a timeout. In order to prevent this, HireFire will automatically scale your web dynos up when traffic increases to ensure that your application runs fast at all times. When traffic decreases, HireFire will spin down your web dynos again.
>
> —from the HireFire frontpage

It supports the following Python queuing systems as backends:

- Celery
- HotQueue
- Huey
- Queues
- RQ

Feel free to contribute other backends if you're using a different queuing system.

CHAPTER 1

Installation

Install the HireFire package with your favorite installer, e.g.:

```
pip install HireFire
```

Sign up for HireFire and set the `HIREFIRE_TOKEN` environment variable with the Heroku CLI as provided on the specific HireFire application page, e.g.:

```
heroku config:set HIREFIRE_TOKEN=f69f0c0ddebe041248daf187caa6abb3e5d943ca
```

Now follow the quickstart guide below and don't forget to tweak the options in the HireFire management system.

For more help see the Hirefire documentation.

CHAPTER 2

Configuration

The `hirefire` Python package currently supports two frameworks: Django and Tornado. Implementations for other frameworks are planned but haven't been worked on: Flask, Pyramid (PasteDeploy), WSGI middleware, ..

Feel free to contribute one if you can't wait.

The following guides imply you have defined at least one `hirefire.procs.Proc` subclass defined matching one of the processes in your Procfile. For each process you want to monitor you have to have one subclass.

For example here is a `Procfile` which uses RQ for the "worker" proccess:

```
web: python manage.py runserver
worker: DJANGO_SETTINGS_MODULE=mysite.settings rqworker high default low
```

Define a `RQProc` subclass somewhere in your project, e.g. `mysite/procs.py`, with the appropriate attributes (`name` and `queues`):

```
from hirefire.procs.rq import RQProc

class WorkerProc(RQProc):
    name = 'worker'
    queues = ['high', 'default', 'low']
```

See the procs API documentation if you're using another backend. Now follow the framework specific guidelines below.

2.1 Django

Setting up HireFire support for Django is easy:

1. Add `'hirefire.contrib.django.middleware.HireFireMiddleware'` to your `MIDDLEWARE` setting:

```
    # Use ``MIDDLEWARE_CLASSES`` prior to Django 1.10
    MIDDLEWARE = [
        'hirefire.contrib.django.middleware.HireFireMiddleware',
        # ...
    ]
```

   Make sure it's the first item in the list/tuple.

2. Set the `HIREFIRE_PROCS` setting to a list of dotted paths to your procs. For the above example proc:

```
HIREFIRE_PROCS = ['mysite.procs.WorkerProc']
```

3. Set the `HIREFIRE_TOKEN` setting to the token that HireFire shows on the specific application page (optional):

```
HIREFIRE_TOKEN = 'f69f0c0ddebe041248daf187caa6abb3e5d943ca'
```

   This is only needed if you haven't set the `HIREFIRE_TOKEN` environment variable already (see the installation section how to do that on Heroku).

4. Check that the middleware has been correctly setup by opening the following URL in a browser:

```
http://localhost:8000/hirefire/test
```

   You should see an empty page with 'HireFire Middleware Found!'.

   You can also have a look at the page that HireFire checks to get the number of current tasks:

```
http://localhost:8000/hirefire/<HIREFIRE_TOKEN>/info
```

   where `<HIREFIRE_TOKEN>` needs to be replaced with your token or – in case you haven't set the token in your settings or environment – just use `development`.

2.2 Tornado

Setting up HireFire support for Tornado is also easy:

1. Use `hirefire.contrib.tornado.handlers.hirefire_handlers` when defining your `tornado.web.Application` instance:

```
import os
from hirefire.contrib.tornado.handlers import hirefire_handlers

application = tornado.web.Application([
    # .. some patterns and handlers
] + hirefire_handlers(os.environ['HIREFIRE_TOKEN'],
                      ['mysite.procs.WorkerProc']))
```

   Make sure to pass a list of dotted paths to the `hirefire_handlers` function.

2. Set the `HIREFIRE_TOKEN` environment variable to the token that HireFire shows on the specific application page (optional):

```
export HIREFIRE_TOKEN='f69f0c0ddebe041248daf187caa6abb3e5d943ca'
```

   See the installation section above for how to do that on Heroku.

3. Check that the handlers have been correctly setup by opening the following URL in a browser:

```
http://localhost:8888/hirefire/test
```

   You should see an empty page with 'HireFire Middleware Found!'.

   You can also have a look at the page that HireFire checks to get the number of current tasks:

```
http://localhost:8888/hirefire/<HIREFIRE_TOKEN>/info
```

   where `<HIREFIRE_TOKEN>` needs to be replaced with your token or – in case you haven't set the token as an environment variable – just use `development`.

2.3 Flask

Setting up HireFire support for Flask is (again!) also easy:

1. The module `hirefire.contrib.flask.blueprint` provides a `build_hirefire_blueprint` factory function that should be called with HireFire token and procs as arguments. The result is a blueprint providing the hirefire routes and which should be registered inside your app:

```
import os
from flask import Flask
from hirefire.contrib.flask.blueprint import build_hirefire_blueprint

app = Flask(__name__)
bp = build_hirefire_blueprint(os.environ['HIREFIRE_TOKEN'],
                              ['mysite.procs.WorkerProc'])
app.register_blueprint(bp)
```

   Make sure to pass a list of dotted paths to the `build_hirefire_blueprint` function.

2. Set the `HIREFIRE_TOKEN` environment variable to the token that HireFire shows on the specific application page (optional):

```
export HIREFIRE_TOKEN='f69f0c0ddebe041248daf187caa6abb3e5d943ca'
```

   See the installation section above for how to do that on Heroku.

3. Check that the handlers have been correctly setup by opening the following URL in a browser:

```
http://localhost:8080/hirefire/test
```

   You should see an empty page with 'HireFire Middleware Found!'.

   You can also have a look at the page that HireFire checks to get the number of current tasks:

```
http://localhost:8080/hirefire/<HIREFIRE_TOKEN>/info
```

   where `<HIREFIRE_TOKEN>` needs to be replaced with your token or – in case you haven't set the token as an environment variable – just use `development`.

CHAPTER 3

Proc backends

Two base classes are includes that you can use to implement custom backends. All the other contributed backends use those base classes, too.

3.1 hirefire.procs.Proc

class hirefire.procs.**Proc**(*name=None*, *queues=None*)

The base proc class. Use this to implement custom queues or other behaviours, e.g.:

```
import mysite.sekrit
from hirefire import procs

class MyCustomProc(procs.Proc):
    name = 'worker'
    queues = ['default']

    def quantity(self):
        return sum([mysite.sekrit.count(queue)
                    for queue in self.queues])
```

Parameters

- **name** (*str*) – the name of the proc (required)
- **queues** (*str or list of str*) – list of queue names to check (required)

name = None

The name of the proc

quantity(***kwargs*)

Returns the aggregated number of tasks of the proc queues.

Needs to be implemented in a subclass.

`kwargs` must be captured even when not used, to allow for future extensions.

The only kwarg currently implemented is `cache`, which is a dictionary made available for cross-proc caching. It is empty when the first proc is processed.

queues = []

The list of queues to check

3.2 hirefire.procs.ClientProc

class hirefire.procs.**ClientProc**(**args*, ***kwargs*)

A special subclass of the *Proc* class that instantiates a list of clients for each given queue, e.g.:

```
import mysite.sekrit
from hirefire import procs

class MyCustomProc(procs.ClientProc):
    name = 'worker'
    queues = ['default']

    def client(self, queue):
        return mysite.sekrit.Client(queue)

    def quantity(self):
        return sum([client.count(queue)
                    for client in self.clients])
```

See the implementation of the *RQProc* class for an example.

client(*queue*, **args*, ***kwargs*)

Returns a client instance for the given queue to be used in the `quantity` method.

Needs to be implemented in a subclass.

quantity(***kwargs*)

Returns the aggregated number of tasks of the proc queues.

Needs to be implemented in a subclass.

`kwargs` must be captured even when not used, to allow for future extensions.

The only kwarg currently implemented is `cache`, which is a dictionary made available for cross-proc caching. It is empty when the first proc is processed.

3.3 Contributed backends

See the following API overview of the other supported queuing backends.

3.3.1 Procs

This is a auto-generated list of supported `Proc` classes.

Celery

class hirefire.procs.celery.**CeleryProc**(*name=None, queues=['celery'], app=None*)

A proc class for the Celery library.

Parameters

- **name** (*str*) – the name of the proc (required)
- **queues** (*str or list*) – list of queue names to check (required)
- **app** (Celery) – the Celery app to check for the queues (optional)

Declarative example:

```
from celery import Celery
from hirefire.procs.celery import CeleryProc

celery = Celery('myproject', broker='amqp://guest@localhost//')

class WorkerProc(CeleryProc):
    name = 'worker'
    queues = ['celery']
    app = celery
```

Or a simpler variant:

```
worker_proc = CeleryProc('worker', queues=['celery'], app=celery)
```

In case you use one of the non-standard Celery clients (e.g. django-celery) you can leave the `app` attribute empty because Celery will automatically find the correct Celery app:

```
from hirefire.procs.celery import CeleryProc

class WorkerProc(CeleryProc):
    name = 'worker'
    queues = ['celery']
```

Querying the tasks that are on the workers is a more expensive process, and if you're sure that you don't need them, then you can improve the response time by not looking for some statuses. The default statuses that are looked for are `active`, `reserved`, and `scheduled`. You can configure to *not* look for those by overriding the `inspect_statuses` property. For example, this proc would not look at any tasks held by the workers.

```
class WorkerProc(CeleryProc):
    name = 'worker'
    queues = ['celery']
    inspect_statuses = []
```

`scheduled` tasks are tasks that have been triggered with an `eta`, the most common example of which is using `retry` on tasks. If you're sure you aren't using these tasks, you can skip querying for these tasks.

`reserved` tasks are tasks that have been taken from the queue by the main process (coordinator) on the worker dyno, but have not yet been given to a worker run. If you've configured Celery to only fetch the tasks that it is currently running, then you may be able to skip querying for these tasks. See http://docs.celeryproject.org/en/latest/userguide/optimizing.html#prefetch-limits form more information.

`active` tasks are currently running tasks. If your tasks are short-lived enough, then you may not need to look for these tasks. If you choose to not look at active tasks, look out for `WorkerLostError` exceptions. See https://github.com/celery/celery/issues/2839 for more information.

If you have a particular simple case, you can use a shortcut to eliminate one inspect call when inspecting statuses. The `active_queues` inspect call is needed to map `exchange` and `routing_key` back to the celery `queue` that it is for. If all of your `queue`, `exchange`, and `routing_key` are the same (which is the default in Celery), then you can use the `simple_queues = True` flag to note that all the queues in the proc use the same name for their `exchange` and `routing_key`. This defaults to `False` for backward compatibility, but if your queues are using this simple setup, you're encouraged to use it like so:

```
class WorkerProc(CeleryProc):
    name = 'worker'
    queues = ['celery']
    simple_queues = True
```

Because of how this is implemented, you will almost certainly wish to use this feature on all of your procs, or on none of them. This is because both variants have separate caches that make separate calls to the inspect methods, so having both kinds present will mean that the inspect calls will be run twice.

app = None
The Celery app to check for the queues (optional).

inspect_count (*cache*)
Use Celery's inspect() methods to see tasks on workers.

inspect_statuses = ['active', 'reserved', 'scheduled']
The Celery task status to check for on workers (optional). Valid options are 'active', 'reserved', and 'scheduled'.

name = None
The name of the proc (required).

quantity (*cache=None*, ***kwargs*)
Returns the aggregated number of tasks of the proc queues.

queues = ['celery']
The list of queues to check (required).

simple_queues = False
Whether or not the exchange and routing_key are the same as the queue name for the queues in this proc. Default: False.

HotQueue

class hirefire.procs.hotqueue.**HotQueueProc**(*name=None*, *queues=[]*, *connection_params={}*)

A proc class for the HotQueue library.

Parameters

- **name** (*str*) – the name of the proc (required)
- **queues** (*str or list*) – list of queue names to check (required)
- **connection_params** (*dict*) – the connection parameter to use by default (optional)

Example:

```
from hirefire.procs.hotqueue import HotQueueProc

class WorkerHotQueueProc(HotQueueProc):
    name = 'worker'
    queues = ['myqueue']
    connection_params = {
        'host': 'localhost',
        'port': 6379,
        'db': 0,
    }
```

client (*queue*)
Given one of the configured queues returns a hotqueue.HotQueue instance with the *connection_params*.

connection_params = {}
The connection parameter to use by default (optional).

name = None
The name of the proc (required).

quantity (***kwargs*)
Returns the aggregated number of tasks of the proc queues.

queues = []
The list of queues to check (required).

Huey

Queues

class hirefire.procs.queues.**QueuesProc**(*name=None*, *queues=[]*)
A proc class for the queues library.

Parameters

- **name** (*str*) – the name of the proc (required)
- **queues** (str or list of str or queues.queues.Queue) – list of queue names to check (required)

Example:

```
from hirefire.procs.queues import QueuesProc

class WorkerQueuesProc(QueuesProc):
    name = 'worker'
    queues = ['default', 'thumbnails']
```

client(*queue*)
Given one of the configured queues returns a queues.queues.Queue instance.

name = None
The name of the proc (required).

quantity (***kwargs*)
Returns the aggregated number of tasks of the proc queues.

queues = []
The list of queues to check (required).

RQ

class hirefire.procs.rq.**RQProc**(*name=None, queues=['default'], connection=None*)
A proc class for the RQ library.

Parameters

- **name** (*str*) – the name of the proc (required)
- **queues** (*str or list*) – list of queue names to check (required)
- **connection** (redis.Redis) – the connection to use for the queues (optional)

Example:

```
from hirefire.procs.rq import RQProc

class WorkerRQProc(RQProc):
    name = 'worker'
    queues = ['high', 'default', 'low']
```

client (*queue*)
Given one of the configured queues returns a `rq.Queue` instance using the *connection*.

connection = None
The connection to use for the queues (optional).

name = None
The name of the proc (required).

quantity (***kwargs*)
Returns the aggregated number of tasks of the proc queues.

queues = ['default']
The list of queues to check (required).

CHAPTER 4

Issues & Feedback

For bug reports, feature requests and general feedback, please use the Github issue tracker.

CHAPTER 5

Thanks

Many thanks to the folks at Hirefire for building a great tool for the Heroku ecosystem.

5.1 Authors

- Emmanuel Leblond
- Jannis Leidel
- Marc Tamlyn
- Ryan Hiebert
- Ryan West
- Shravan Reddy

CHAPTER 6

Changes

CHAPTER 7

0.5 (2017-01-20)

- Add `simple_queues` feature to Celery Proc, to enable optionally skipping one inspect call. (#23)
- Make the default quantity 0 by the recommendation of the HireFire team.

CHAPTER 8

0.4 (2016-06-04)

- Consider all Celery tasks including the ones in the active, reserved and scheduled queues. This fixes a long standing issue where tasks in those queues could have been dropped if HireFire were to scale down the workers. Many thanks to Ryan Hiebert for working on this.
- Removed django-pq backend since the library is unmaintained.

0.3 (2015-05-05)

- Added Flask blueprint.
- Fixed Celery queue length measurement for AMQP backends.

CHAPTER 10

0.2.2 (2014-11-27)

- Fixed a regression in 0.2.1 fix. Thanks to Ryan West.

0.2.1 (2014-05-27)

- Fix the RQ Proc implementation to take the number of task into account that are currently being processed by the workers to prevent accidental shutdown mid-processing. Thanks to Jason Lantz for the report and initial patch.

CHAPTER 12

0.2 (2014-04-20)

- Got rid of d2to1 dependency.
- Added django-pq backend.
- Ported to Python 3.
- Added Tornado contrib handlers.

CHAPTER 13

0.1 (2013-02-17)

- Initial release with backends:
  - Celery
  - HotQueue
  - Huey
  - Queues
  - RQ

A

C

H

I

N

P

Q

R

S