
jobtastic- Celery tasks plus more awesome

build unknown

Jobtastic makes your user-responsive long-running Celery jobs totally awesomer. Celery is the ubiquitous python job queueing tool and jobtastic is a python library that adds useful features to your Celery tasks. Specifically, these are features you probably want if the results of your jobs are expensive or if your users need to wait while they compute their results.

Jobtastic gives you goodies like: * Easy progress estimation/reporting * Job status feedback * Helper methods for gracefully handling a dead task broker (`delay_or_eager` and `delay_or_fail`) * Super-easy result caching * Thundering herd avoidance * Integration with a celery jQuery plugin for easy client-side progress display * Memory leak detection in a task run

Make your Celery jobs more awesome with Jobtastic.

Why Jobtastic?

If you have user-facing tasks for which a user must wait, you should try Jobtastic. It's great for: * Complex reports * Graph generation * CSV exports * Any long-running, user-facing job

You could write all of the stuff yourself, but why?

Installation

1. Install gcc and the python C headers so that you can build psutil.

On Ubuntu, that means running:

```
$ sudo apt-get install build-essential python-dev python2.7-dev  
python3.5-dev rabbitmq-server
```

On OS X, you'll need to run the "XcodeTools" installer.

2. Get the project source and install it

```
$ pip install jobtastic
```

Creating Your First Task

Let's take a look at an example task using Jobtastic:

```

1 from time import sleep
2
3 from jobtastic import JobtasticTask
4
5 class LotsOfDivisionTask(JobtasticTask):
6     """
7     Division is hard. Make Celery do it a bunch.
8     """
9     # These are the Task kwargs that matter for caching purposes
10    significant_kwargs = [
11        ('numerators', str),
12        ('denominators', str),
13    ]
14    # How long should we give a task before assuming it has failed?
15    herd_avoidance_timeout = 60 # Shouldn't take more than 60 seconds
16    # How long we want to cache results with identical ``
17    # significant_kwargs``
18    cache_duration = 0 # Cache these results forever. Math is pretty
19    # stable.
20    # Note: 0 means different things in different cache backends. RTFM
21    # for yours.
22
23    def calculate_result(self, numerators, denominators, **kwargs):
24        """
25        MATH!!!
26        """
27        results = []
28        divisions_to_do = len(numerators)
29        # Only actually update the progress in the backend every 10
30        # operations
31        update_frequency = 10
32        for count, divisors in enumerate(zip(numerators, denominators)):
33            :
34            numerator, denominator = divisors
35            results.append(numerator / denominator)
36            # Let's let everyone know how we're doing
37            self.update_progress(
38                count,
39                divisions_to_do,
40                update_frequency=update_frequency,
41            )
42            # Let's pretend that we're using the computers that landed
43            # us on the moon
44            sleep(0.1)
45
46        return results

```

This task is very trivial, but imagine doing something time-consuming instead of division (or just a ton of division) while a user waited. We wouldn't want a double-clicker to cause this to happen twice

concurrently, we wouldn't want to ever redo this work on the same numbers and we would want the user to have at least some idea of how long they'll need to wait. Just by setting those 3 member variables, we've done all of these things.

Basically, creating a Celery task using Jobtastic is a matter of:

1. Subclassing `jobtastic.JobtasticTask`
2. Defining some required member variables
3. Writing your `calculate_result` method (instead of the normal Celery `run()` method)
4. Sprinkling `update_progress()` calls in your `calculate_result()` method to communicate progress

Now, to use this task in your Django view, you'll do something like:

```
1 from django.shortcuts import render_to_response
2
3 from my_app.tasks import LotsOfDivisionTask
4
5 def lets_divide(request):
6     """
7     Do a set number of divisions and keep the user up to date on
8     progress.
9     """
10    iterations = request.GET.get('iterations', 1000) # That's a lot.
11    Right?
12    step = 10
13
14    # If we can't connect to the backend, let's not just 500. k?
15    result = LotsOfDivisionTask.delay_or_fail(
16        numerators=range(0, step * iterations * 2, step * 2),
17        denominators=range(1, step * iterations, step),
18    )
19
20    return render_to_response(
21        'my_app/lets_divide.html',
22        {'task_id': result.task_id},
23    )
```

The `my_app/lets_divide.html` template will then use the `task_id` to query the task result all asynchronous-like and keep the user up to date with what is happening.

For Flask, you might do something like:

```
1 from flask import Flask, render_template
2
3 from my_app.tasks import LotsOfDivisionTask
4
5 app = Flask(__name__)
6
```

```
7 @app.route("/", methods=['GET'])
8 def lets_divide():
9     iterations = request.args.get('iterations', 1000)
10    step = 10
11
12    result = LotsOfDivisionTask.delay_or_fail(
13        numerators=range(0, step * iterations * 2, step * 2),
14        denominators=range(1, step * iterations, step),
15    )
16
17    return render_template('my_app/lets_divide.html', task_id=result.
        task_id)
```

Required Member Variables

“But wait, Wes. What the heck do those member variables actually do?” You ask.

Firstly. How the heck did you know my name?

And B, why don't I tell you!?

significant_kwargs This is key to your caching magic. It's a list of 2-tuples containing the name of a kwarg plus a function to turn that kwarg in to a string. Jobtastic uses these to determine if your task should have an identical result to another task run. In our division example, any task with the same numerators and denominators can be considered identical, so Jobtastic can do smart things.

```
1 significant_kwargs = [
2     ('numerators', str),
3     ('denominators', str),
4 ]
```

If we were living in bizzaro world, and only the numerators mattered for division results, we could do something like:

```
1 significant_kwargs = [
2     ('numerators', str),
3 ]
```

Now tasks called with an identical list of numerators will share a result.

herd_avoidance_timeout This is the max number of seconds for which Jobtastic will wait for identical task results to be determined. You want this number to be on the very high end of the amount of time you expect to wait (after a task starts) for the result. If this number is hit, it's assumed that

something bad happened to the other task run (a worker failed) and we'll start calculating from the start.

Optional Member Variables

These let you tweak the default behavior. Most often, you'll just be setting the `cache_duration` to enable result caching.

cache_duration If you want your results cached, set this to a non-negative number of seconds. This is the number of seconds for which identical jobs should try to just re-use the cached result. The default is -1, meaning don't do any caching. Remember, `JobtasticTask` uses your `significant_kwargs` to determine what is identical.

cache_prefix This is an optional string used to represent tasks that should share cache results and thundering herd avoidance. You should almost never set this yourself, and instead should let Jobtastic use the `module.class` name. If you have two different tasks that should share caching, or you have some very-odd cache key conflict, then you can change this yourself. You probably don't need to.

memleak_threshold Set this value to monitor your tasks for any runs that increase the memory usage by more than this number of Megabytes (the SI definition). Individual task runs that increase resident memory by more than this threshold get some extra logging in order to help you debug the problem. By default, it logs the following via standard Celery logging: * The memory increase * The memory starting value * The memory ending value * The task's kwargs

You then grep for `Jobtastic:memleak memleak_detected` in your logs to identify offending tasks.

If you'd like to customize this behavior, you can override the `warn_of_memory_leak` method in your own `Task`.

Method to Override

Other than tweaking the member variables, you'll probably want to actually, you know, *do something* in your task.

calculate_result This is where your magic happens. Do work here and return the result.

You'll almost definitely want to call `update_progress` periodically in this method so that your users get an idea of for how long they'll be waiting.

Progress feedback helper

This is the guy you'll want to call to provide nice progress feedback and estimation.

update_progress In your `calculate_result`, you'll want to periodically make calls like:

```
1 self.update_progress(work_done, total_work_to_do)
```

Jobtastic takes care of handling timers to give estimates, and assumes that progress will be roughly uniform across each work item.

Most of the time, you really don't need ultra-granular progress updates and can afford to only give an update every `N` items completed. Since every update would potentially hit your CELERY_RESULT_BACKEND, and that might cause a network trip, it's probably a good idea to use the optional `update_frequency` argument so that Jobtastic doesn't swamp your backend with updated estimates no user will ever see.

In our division example, we're only actually updating the progress every 10 division operations:

```
1 # Only actually update the progress in the backend every 10 operations
2 update_frequency = 10
3 for count, divisors in enumerate(zip(numerators, denominators)):
4     numerator, denominator = divisors
5     results.append(numerator / denominator)
6     # Let's let everyone know how we're doing
7     self.update_progress(count, divisions_to_do, update_frequency=10)
```

Using your JobtasticTask

Sometimes, your Task Broker just up and dies (I'm looking at you, old versions of RabbitMQ). In production, calling straight up `delay()` with a dead backend will throw an error that varies based on what backend you're actually using. You probably don't want to just give your user a generic 500 page if your broker is down, and it's not fun to handle that exception every single place you might use Celery. Jobtastic has your back.

Included are `delay_or_eager` and `delay_or_fail` methods that handle a dead backend and do something a little more production-friendly.

Note: One very important caveat with `JobtasticTask` is that all of your arguments must be keyword arguments.

Note: This is a limitation of the current `significant_kwargs` implementation, and totally fixable if someone wants to submit a pull request.

delay_or_eager

If your broker is behaving itself, this guy acts just like `delay()`. In the case that your broker is down, though, it just goes ahead and runs the task in the current process and skips sending the task to a worker. You get back a nice shiny `EagerResult` object, which behaves just like the `AsyncResult` you were expecting. If you have a task that realistically only takes a few seconds to run, this might be better than giving your users an error message.

This method uses `async_or_eager()` under the hood.

delay_or_fail

Like `delay_or_eager`, this helps you handle a dead broker. Instead of running your task in the current process, this actually generates a task result representing the failure. This means that your client-side code can handle it like any other failed task and do something nice for the user. Maybe send them a fruit basket?

For tasks that might take a while or consume a lot of RAM, you're probably better off using this than `delay_or_eager` because you don't want to make a resource problem worse.

This method uses `async_or_fail()` under the hood.

async_or_eager

This is a version of `delay_or_eager()` that exposes the calling signature of `apply_async()`.

async_or_fail

This is a version of `delay_or_fail()` that exposes the calling signature of `apply_async()`.

Client Side Handling

That's all well and good on the server side, but the biggest benefit of Jobtastic is useful user-facing feedback. That means handling status checks using AJAX in the browser.

The easiest way to get rolling is to use our sister project, jquery-celery. It contains jQuery plugins that help you: * Poll for task status and handle the result * Display a progress bar using the info from the **PROGRESS** state. * Display tabular data using DataTables.

If you want to roll your own, the general pattern is to poll a URL (such as the django-celery task_status view) with your taskid to get JSON status information and then handle the possible states to keep the user informed.

The jquery-celery jQuery plugin might still be useful as reference, even if you're rolling your own. In general, you'll want to handle the following cases:

PENDING

Your task is still waiting for a worker process. It's generally useful to display something like "Waiting for your task to begin".

PROGRESS

Your task has started and you've got a JSON object like:

```
1 {  
2     "progress_percent": 0,  
3     "time_remaining": 300  
4 }
```

progress_percent is a number between 0 and 100. It's a good idea to give a different message if the percent is 0, because the time remaining estimate might not yet be well-calibrated.

time_remaining is the number of seconds estimated to be left. If there's no good estimate available, this value will be -1.

SUCCESS

You've got your data. It's time to display the result.

FAILURE

Something went wrong and the worker reported a failure. This is a good time to either display a useful error message (if the user can be expected to correct the problem), or to ask the user to retry their task.

Non-200 Request

There are occasions where requesting the task status itself might error out. This isn't a reflection on the worker itself, as it could be caused by any number of application errors. In general, you probably want to try again if this happens, but if it persists, you'll want to give your user feedback.

Running The Test Suite

We use tox to run our tests against various combinations of python/Django/Celery. We only officially support the combinations listed in our `.travis.yml` file, but we're working on (Issue 33) supporting everything defined in `tox.ini`. Until then, you can run tests against supported combos with:

```
1 $ pip install tox
2 $ tox -e py27-django1.8.X-djangocelery3.1.X-celery3.1.X
```

Our test suite currently only tests usage with Django, which is definitely a bug. Especially if you use Jobtastic with Flask, we would love a pull request.

Dynamic Time Estimates via JobtasticMixins

Have tasks whose duration is difficult to estimate or that doesn't have smooth progress? JobtasticMixins to the rescue!

JobtasticMixins provides an `AVGTimeRedis` mixin that stores duration data in a Redis backend. It then automatically uses this stored historical data to calculate an estimate. For more details, check out JobtasticMixins on github.

Is it Awesome?

Yes. Increasingly so.

Project Status

Jobtastic is currently known to work with Django 1.6+ and Celery 3.1.X The goal is to support those versions and newer. Please file issues if there are problems with newer versions of Django/Celery.

Gotchas

At this time of this writing, the latest supported version of kombu with celery 4.x is 4.0.2. This is due to an issue with invalid or temporarily broken brokers with the newer versions of kombu.

Also, [RabbitMQ](#) should be running in the background while running tests.

A note on usage with Flask

Previously, if you were using Flask instead of Django, then the only currently-supported way to work with Jobtastic was with Memcached as your [CELERY_RESULT_BACKEND](#).

Thanks to @rhunwicks this is no longer the case!

A cache is now selected with the following priority:

- If the Celery appconfig has a [JOBTASTIC_CACHE](#) setting and it is a valid cache, use it
- If Django is installed, then:
 - If the setting is a valid Django cache entry, then use that.
 - If the setting is empty use the default cache
- If Werkzeug is installed, then:
 - If the setting is a valid Celery Memcache or Redis Backend, then use that.
 - If the setting is empty and the default Celery Result Backend is Memcache or Redis, then use that

Non-affiliation

This project isn't affiliated with the awesome folks at the Celery Project (unless having a huge crush counts as affiliation). It's a library that the folks at PolicyStat have been using internally and decided to open source in the hopes it is useful to others.