Alligator Documentation

Release 0.10.0

Daniel Lindsley

Sep 27, 2017

Contents

Simple offline task queues. For Python.

"See you later, alligator."

CHAPTER 1

Guide

Installing Alligator

Installation of Alligator itself is a relatively simple affair. For the most recent stable release, simply use pip to run:

```
$ pip install alligator
```

Alternately, you can download the latest development source from Github:

```
$ git clone https://github.com/toastdriven/alligator.git
$ cd alligator
$ python setup.py install
```

Queue Backends

Alligator includes a `Local Memory Client`, which is useful for development or testing (no setup required). However, this is not very scalable.

For production use, you should install one of the following servers used for queuing:

Redis

A in-memory data structure server, it offers excellent speed as well as being a frequently-already-installed server. Official releases can be found at http://redis.io/download.

You'll also need to install the `redis` package:

```
$ pip install redis
```

You can also install via other package managers:

```
# On Mac with Homebrew
$ brew install redis

# On Ubuntu
$ sudo aptitude install redis
```

Beanstalk

A simple & fast queue. Official releases can be found at http://kr.github.io/beanstalkd/.

> **Warning:** This backend does not support Python 3.3+, as the underlying dependencies have not been ported.

You'll also need to install the both the `beanstalkc` & `PyYAML` packages:

```
$ pip install beanstalkc
$ pip install PyYAML
```

You can also install via other package managers:

```
# On Mac with Homebrew
$ brew install beanstalk

# On Ubuntu
$ sudo aptitude install beanstalkd
```

> **Warning:** Beanstalk works differently than the other queues in several notable ways:
>
> 1. It does **NOT** support custom `task_id``s. You can still set them and it will be preseved in the task, but the backend will overwrite your ``task_id` choice once it's in the queue.
> 2. Dropping a whole queue is **VERY** inefficient.
> 3. It depends on **PyYAML** & uses a known-unsafe parsing method, which could make you susceptible to MiTM attacks.
>
> It's still an excellent choice at large volumes, but you should be aware of the shortcomings.

SQS

'Amazon SQS'_ is a queue service created by Amazon Web Services. It works well in large-scale environments or if you're already using other AWS services.

It has the benefit of not requiring an installed setup, only an AWS account & a credit card, making it the easiest of the production queues to setup.

You'll need to install the `boto` packages:

```
$ pip install 'boto>=2.35.0'
```

Warning: SQS works differently than the other queues in a couple ways:

1. It does **NOT** support custom `task_id``s. You can still set them and it will be preseved in the task, but the backend will overwrite your ``task_id` choice once it's in the queue.
2. You must **manually** create queues! Alligator will not auto-create them (to save on requests performed & therefore how much you are billed). Create the `Gator` queues at the AWS console before trying to use them.
3. It does **NOT** support `gator.get(...)`, as this functionality is not supported by the SQS service itself.

It's also an excellent choice at large volumes, but you should be aware of the shortcomings.

Alligator Tutorial

Alligator is a simple offline task queuing system. It enables you to take expensive operations & move them offline, either to a different process or even a whole different server.

This is extremely useful in the world of web development, where request-response cycles should be kept as quick as possible. Scheduling tasks helps remove expensive operations & keeps end-users happy.

Some example good use-cases for offline tasks include:

- Sending emails
- Resizing images/creating thumbnails
- Notifying social networks
- Fetching data from other data sources

You should check out the instructions on *Installing Alligator* to install Alligator.

Alligator is written in pure Python & can work with all frameworks. For this tutorial, we'll assume integration with a Django-based web application, but it could just as easily be used with Flask, Pyramid, pure WSGI applications, etc.

Philosophy

Alligator is a bit different in approach from other offline task systems. Let's highlight some ways & the why's.

Tasks Are Any Plain Old Function No decorators, no special logic/behavior needed inside, no inheritance. **ANY** importable Python function can become a task with no modifications required.

Importantly, it must be importable. So instance methods on a class aren't processable.

Plain Old Python Nothing specific to any framework or architecture here. Plug it in to whatever code you want.

Simplicity The code for Alligator should be small & fast. No complex gymnastics, no premature optimizations or specialized code to suit a specific backend.

You're In Control Your code calls the tasks & can setup all the execution options needed. There are hook functions for special processing, or you can use your own `Task` or `Client` classes.

Additionally, you control the consuming of the queue, so it can be processed your way (or fanned out, or prioritized, or whatever).

Figure Out What To Offline

The very first thing to do is figure out where the pain points in your application are. Doing this analysis differs wildly (though things like django-debug-toolbar, profile or snakeviz can be helpful). Broadly speaking, you should look for things that:

- access the network
- do an expensive operation
- may fail & require retrying
- things that aren't immediately required for success

If you have a web application, just navigating around & timing pageloads can be a cheap/easy way of finding pain points.

For the purposes of this tutorial, we'll assume a user of our hot new Web 3.0 social network made a new post & all their followers need to see it.

So our existing view code might look like:

```
from django.conf import settings
from django.http import Http404
from django.shortcuts import redirect, send_email

from sosocial.models import Post


def new_post(request):
    if not request.method == 'POST':
        raise Http404('Gotta use POST.')

    # Don't write code like this. Sanitize your data, kids.
    post = Post.objects.create(
        message=request.POST['message']
    )

    # Ugh. We're sending an email to everyone who follows the user, which
    # could mean hundreds or thousands of emails. This could timeout!
    subject = "A new post by {}".format(request.user.username)
    to_emails = [follow.email for follow in request.user.followers.all()]
    send_email(
        subject,
        post.message,
        settings.SERVER_EMAIL,
        recipient_list=to_emails
    )

    # Redirect like a good webapp should.
    return redirect('activity_feed')
```

Creating a Task

The next step won't involve Alligator at all. We'll extract that slow code into an importable function, then call it from where the code used to be. So we can convert our existing code into:

```
from django.contrib.auth.models import User
from django.conf import settings
```

```
from django.http import Http404
from django.shortcuts import redirect, send_email

from sosocial.models import Post


def send_post_email(user_id, post_id):
    post = Post.objects.get(pk=post_id)
    user = User.objects.get(pk=user_id)

    subject = "A new post by {}".format(user.username)
    to_emails = [follow.email for follow in user.followers.all()]
    send_email(
        subject,
        post.message,
        settings.SERVER_EMAIL,
        recipient_list=to_emails
    )


def new_post(request):
    if not request.method == 'POST':
        raise Http404('Gotta use POST.')

    # Don't write code like this. Sanitize your data, kids.
    post = Post.objects.create(
        message=request.POST['message']
    )

    # The code was here. Now we'll call the function, just to make sure
    # things still work.
    send_post_email(request.user.pk, post.pk)

    # Redirect like a good webapp should.
    return redirect('activity_feed')
```

Now go run your tests or hand-test things to ensure they still work. This is important because it helps guard against regressions in your code.

You'll note we're not directly passing the `User` or `Post` instances, instead passing the primary identifiers, even as it stands it's causing two extra queries. While this is sub-optimal as things stands, it neatly prepares us for offlining the task.

Note: Why not pass the instances themselves?

While it's possible to create instances that nicely serialize, the problem with this approach is stale data & unnecessarily large payloads.

While the ideal situation is tasks that are processed within seconds of being added to the queue, in the real world, queues can get backed up & users may further change data. By fetching the data fresh when processing the task, you ensure you're not working with old data.

Further, most queues are optimized for small payloads. The more data to send over the wire, the slower things go. Given that's the opposite reason for adding a task queue, it doesn't make sense.

Create a Gator Instance

While it's great we got better encapsulation by pulling out the logic into its own function, we're still doing the sending of email in-process, which means our view is still slow.

This is where Alligator comes in. We'll start off by importing the `Gator` class at the top of the file & making an instance.

Note: Unless you're only using Alligator in **one** file, a best practice would be to put that import & initialization into it's own file, then import that configured `gator` object into your other files. Configuring it in one place is better than many instantiations (but also allows for setting up a different instance elsewhere).

When creating a `Gator` instance, you'll need to choose a queue backend. Alligator ships with support for local-memory, Redis & Beanstalk. See the *Installing Alligator* docs for setup info.

Local Memory

Primarily only for development or in testing, this has no dependencies, but keeps everything in-process.

```
from alligator import Gator

# Creates an in-memory/in-process queue.
# The same process must consume from the queue, or things will be thrown
# away when the process exits.
gator = Gator('locmem://')
```

Redis

Redis is a good option for production and small-large installations.

```
from alligator import Gator

# Connect to a locally-running Redis server & use DB 0.
gator = Gator('redis://localhost:6379/0')
```

Beanstalk

Beanstalk specializes in queuing & can be used in production at large-very large installations.

```
from alligator import Gator

# Connect to a locally-running Beanstalk server.
gator = Gator('beanstalk://localhost:11300/')
```

SQS

Amazon SQS is specifically a queue service & works well in large-scale environments.

```
from alligator import Gator
```

```
# Connect to a locally-running Beanstalk server.
gator = Gator('sqs://us-west-2/')
```

For the duration of the tutorial, we'll assume you chose Redis.

Put the Task on the Queue

After we make a `Gator` instance, the only other change is to how we call `send_post_email`. Instead of calling it directly, we'll need to enqueue a task.

There are two common ways of creating a task in Alligator:

`gator.task()` A typical function call. You pass in the callable & the `*args`/`**kwargs` to provide to the callable. It gets put on the queue with the default task execution options.

`gator.options()` Creates a context manager that has a `.task()` method that works like the above. This is useful for controlling the task execution options, such as retries or if the task should be asynchronous. See the "Working Around Failsome Tasks" section below.

Since we're just starting out with Alligator & looking to replicate the existing behavior, we'll use `gator.task(...)` to create & enqueue the task.

```
# Old code
send_post_email(request.user.pk, post.pk)

# New code
gator.task(send_post_email, request.user.pk, post.pk)
```

Hardly changed in code, but a world of difference in execution speed. Rather than blasting out hundreds of emails & possibly timing out, a task is placed on the queue & execution continues quickly. The complete code looks like:

```
from alligator import Gator

from django.contrib.auth.models import User
from django.conf import settings
from django.http import Http404
from django.shortcuts import redirect, send_email

from sosocial.models import Post


# Please configure this once & import it elsewhere.
# Bonus points if you use a settings (e.g. ``settings.ALLIGATOR_DSN``)
# instead of a hard-coded string.
gator = Gator('redis://localhost:6379/0')

def send_post_email(user_id, post_id):
    post = Post.objects.get(pk=post_id)
    user = User.objects.get(pk=user_id)

    subject = "A new post by {}".format(user.username)
    to_emails = [follow.email for follow in user.followers.all()]
    send_email(
        subject,
        post.message,
        settings.SERVER_EMAIL,
        recipient_list=to_emails
```

```
    )


def new_post(request):
    if not request.method == 'POST':
        raise Http404('Gotta use POST.')

    # Don't write code like this. Sanitize your data, kids.
    post = Post.objects.create(
        message=request.POST['message']
    )

    # The function call was here. Now we'll create a task then carry on.
    gator.task(send_post_email, request.user.pk, post.pk)

    # Redirect like a good webapp should.
    return redirect('activity_feed')
```

Running a Worker

Time to kick back, relax & enjoy your speedy new site, right?

Unfortunately, not quite. Now we're successfully queuing up tasks for later processing & things are completing quickly, but *nothing is processing those tasks*. So we need to run a `Worker` to consume the queued tasks.

We have two options here. We can either use the included `latergator.py` script or we can create our own. The following are identical in function:

```
$ latergator.py redis://localhost:6379/0
```

Or...

```
# Within something like ``run_tasks.py``...
from alligator import Gator, Worker

# Again, bonus points for an import and/or settings usage.
gator = Gator('redis://localhost:6379/0')

worker = Worker(gator)
worker.run_forever()
```

Both of these will create a long-running process, which will consume tasks off the queue as fast as they can.

While this is fine to start off, if you have a heavily trafficked site, you'll likely need many workers. Simply start more processes (using a tool like Supervisor works best).

You can also make things like management commands, build other custom tooling around processing or even launch workers on their own dedicated servers.

Working Around Failsome Tasks

Sometimes tasks don't always succeed on the first try. Maybe the database is down, the mail server isn't working or a remote resource can't be loaded. As it stands, our task will try once then fail loudly.

Alligator also supports retrying tasks, as well as having an `on_error` hook. To specify we want retries, we'll have to use the other important bit of Alligator, `Gator.options`.

`Gator.options` gives you a context manager & allows you to configure task execution options that then apply to all tasks within the manager. Using that looks like:

```
# Old code
# gator.task(send_post_email, request.user.pk, post.pk)

# New code
with gator.options(retries=3) as opts:
    # Be careful to use ``opts.task``, not ``gator.task`` here!
    opts.task(send_post_email, request.user.pk, post.pk)
```

Now that task will get three retries when it's processed, making network failures much more tolerable.

Testing Tasks

All of this is great, but if you can't test the task, you might as well not have code.

Alligator supports an `async=False` option, which means that rather than being put on the queue, your task runs right away (acting like you just called the function, but with all the retries & hooks included).

```
# Bonus points for using ``settings.DEBUG`` (or similar) instead of a
# hard-coded ``False``.
with gator.options(async=False) as opts:
    opts.task(send_post_email, request.user.pk, post.pk)
```

Now your existing integration tests (from before converting to offline tasks) should work as expected.

Warning: Make sure you don't accidently commit this & deploy to production. If so, why have an offline task system at all?

Additionally, you get naturally improved ability to test, because now your tasks are just plain old functions. This means you can typically just import the function & write tests against it (rather than the whole view), which makes for better unit tests & fewer integration tests to ensure things work right.

Going Beyond

This is 90%+ of the day-to-day usage of Alligator, but there's plenty more you can do with it.

You may wish to peruse the *Best Practices* docs for ideas on how to keep your Alligator clean & flexible.

If you need more custom functionality, the *Extending Alligator* docs have examples on:

- Customizing task behavior using the `on_start/on_success/on_error` hook functions.
- Custom `Task` classes.
- Multiple queues & `Workers` for scalability.
- Custom backends.
- `Worker` subclasses.

Happy queuing!

Best Practices

Moving to offlining tasks requires some shifts in the way you develop your code. There are also some good tricks/ideas for integrating Alligator.

If you have suggestions for other best practices, please submit a pull request at https://github.com/toastdriven/alligator/pulls!

Configure One `Gator`

This is alluded to in the *Alligator Tutorial*, but unless you have advanced needs, you're probably best off configuring a **single** `Gator` instance in your code. Then you can import that instance wherever you need it.

Generally speaking, you'll want to create a new file for just this, though if you have a `utils.py` or other common file, you can add it there. For example:

```
# Create a new file, like ``myapp/gator.py``
from alligator import Gator

gator = Gator('redis://localhost:6379/0')
```

Then your code elsewhere imports it:

```
# ``myapp/views.py``
from myapp.gator import gator

# ...Later...
def previously_slow_view(request):
    gator.task(expensive_cache_rebuild, user_id=request.user.pk)
```

This helps DRY (Don't Repeat Yourself) up your code. It also helps you avoid having to change many files if you change backends or configuration.

Use Environment Variables or Settings for the `Gator` DSN

Instead of hard-coding the DSN (Data Source Name) for each `Gator` instance, you should rely on a configuration setting instead.

If you're using plain old Python or subscribe to the Twelve-Factor App, you might lean on environment variables set in the shell. For instance, the Alligator test suite does:

```
import os

from alligator import Gator


# Lean on the ENV variable.
gator = Gator(os.environ['ALLIGATOR_CONN'])
```

Then when running your app, you could do the following in development, for ease of setup:

```
$ export ALLIGATOR_CONN=locmem://
$ python myapp.py
```

But the following on production, for handling large loads:

```
$ export ALLIGATOR_CONN=redis://some.dns.name.com:6379/0
$ python myapp.py
```

If you're using something like Django, you could lean on `settings` instead, like:

```
from alligator import Gator

from django.conf import settings


# Lean on the settings variable.
gator = Gator(settings.ALLIGATOR_CONN)
```

And have differing settings files for development vs. production.

Use an Alternate Queue for Testing

This is an **important** one. By default, Alligator doesn't make any assumptions about what environment (development, testing, production) it is in. So the same queue name will be used.

Especially if you have a shared queue setup for running tests, you can **accidentally** add testing data to your queue! There are two possible resolutions to this:

1. Don't Share

   Set your testing environment up such that it has it's own queue stack. This will nicely isolate things & not require any code changes.

2. Prefix your `queue_name`

   If you must share setup (for instance, developing & testing on the same machine), use a similar approach to the "Env/Settings for Gator DSN" tip, providing a prefix for your queue name. For example:

```
import os

from alligator import Gator


# Lean on the ENV variable for a queue prefix.
gator = Gator(
    'redis://localhost:6379/0',
    # If you ``export ALLIGATOR_PREFIX=test```, your queue name
    # becomes 'test_all'. If not set, it's just 'all'.
    queue_name='_'.join([os.environ.get('ALLIGATOR_PREFIX', ''), 'all'])
)
```

Use Environment Variables or Settings for `Task.async`

If you're just using `gator.task` & trying to write tests, you may have a hard time verifying behavior in an integration test (though you should be able to just unit test the task function).

On the other hand, if you use the `gator.options` context manager & supply an `async=False` execution option, integration tests become easy, as the expense of possibly accidentally committing that & causing issues in production.

The best approach is to use the `gator.options` context manager, but use an environment variable/setting to control if things run asynchronously.

```
import os

# Using the above tip of a single import...
from myapp.gator import gator


def some_view(request):
    with gator.options(async=os.environ['ALLIGATOR_ASYNC']) as opts:
        opts.task(expensive_thing)
```

This allows you to set `export ALLIGATOR_ASYNC=False` in development/testing (so the task runs right away in-process) but queues appropriately in production.

Simple Task Parameters

When creating task functions, you want to simplify the arguments passed to it, as well as removing as many assumptions as possible.

You may be tempted to try to save queries by passing full objects or large lists of things as a parameter.

However, you must remember that the task may run at a very different time (perhaps hours in the future if you're overloaded) or on a completely different machine than the one scheduling the task. Data goes stale easily & few things are as frustrating to debug as stale data being re-written over the top of new data.

Where possible, do the following things:

- Pass primary keys or identifiers instead of rich objects
- Persist large collections in the database or elsewhere, then pass a lookup identifier to the task
- Use simple data types, as they serialize well & result in smaller queue payloads, meaning faster scheduling & consuming of tasks

Re-use the `Gator.options` Context Manager

All the examples in the Alligator docs show creating a single task within a `gator.options(...)` context manager. So you might be tempted to write code like:

```
with gator.options(retries=3) as opts:
    opts.task(send_mass_mail, list_id)

with gator.options(retries=3) as opts:
    opts.task(update_follow_counts, request.user.pk)
```

However, you can reuse that context manager to provide the same execution options to **all** tasks within the block. So we can clean up & shorten our code to:

```
with gator.options(retries=3) as opts:
    opts.task(send_mass_mail, list_id)
    opts.task(update_follow_counts, request.user.pk)
```

Two unique tasks will still be created, but both will have the `retries=3` provided to better ensure they succeeed.

Extending Alligator

If you've read the *Alligator Tutorial*, you've seen some basic usage of Alligator, which largely comes down to:

- create a `Gator` instance, ...
- use either `gator.task(...)` or `gator.options(...)` to enqueue the task, ...
- and then using either `latergator.py` or a custom script with a `Worker` to process the queue.

While this is great in the common/simple case, there may be times where you need more complex things. Alligator was built for extension, so this document will outline ways you can get more out of it.

Hook Methods

In addition to the task function itself, every task supports three optional hook functions:

`on_start(task)` A function called when the task is first pulled off the queue but processing hasn't started.

`on_success(task, result)` A function called when the task completes successfully (no exceptions thrown). If the task function returns a value, it is passed to this function as well.

`on_error(task, err)` A function called when the task fails during processing (an exception was raised). The exception is passed as well as the failed task.

All together, this lets you do more complex things without muddying the task function itself. For instance, the following code would log the start/completion of a task, increment a success count & potentially extend the number of retries on error.

```
import logging

from alligator import Gator
import requests

from myapp.exceptions import FeedUnavailable, FeedNotFound
from myapp.utils import cache


log = logging.getLogger(__file__)


# This is the main task function.
def fetch_feeds(feeds):
    for feed_url in feeds:
        resp = requests.get(feed_url)

        if resp.status_code == 503:
            raise FeedUnavailable(feed_url)
        elif resp.status_code != 200:
            raise FeedNotFound(feed_url)

        # Some other processing of the feed data

    return len(feeds)


# Hook functions
def log_start(task):
    log.info('Starting to import feeds via task {}...'.format(task.task_id))
```

```
def log_success(task, result):
    log.info('Finished importing {} feeds via task {}.'.format(
        result,
        task.task_id
    ))
    cache.incr('feeds_imported', incr_by=result)

def maybe_retry_error(task, err):
    if isinstance(err, FeedUnavailable):
        # Try again soon.
        task.retries += 1
    else:
        log.error('Importing feed url {} failed.'.format(str(err)))


# Now we can use those hooks.
with gator.options(on_start=log_start, on_success=log_success, on_error=maybe_retry_
↪error) as opts:
    opts.task(fetch_feeds, [
        'http://daringfireball.net/feeds/main',
        'http://xkcd.com/rss.xml',
        'http://www.reddit.com/r/DotA2/.rss',
    ])
```

Custom Task Classes

Sometimes, just the built-in arguments for `Task` (like `retries`, `async`, `on_start`/`on_success`/`on_error`) may not be enough. Or perhaps your hook methods will *always* be the same & you don't want to have to pass them all the time. Or perhaps you never need the hook methods, but are running into payload size restrictions by your preferred backend & need some extra space.

For this, you can create custom `Task` classes for use in place of the built-in one. Since that last restriction can be especially pertinent, let's show how we'd handle getting more space in our payload.

First, we need a `Task` subclass. You can create your own (as long as they follow the protocol), but subclassing is easier here.

```
# myapp/skinnytask.py
import bz2

from alligator import Task


class SkinnyTask(Task):
    # We're both going to ignore some keys (async, options) we don't care
    # about, as well as compress/decompress the payload.
    def serialize(self):
        data = {
            'task_id': self.task_id,
            'retries': self.retries,
            'module': determine_module(self.func),
            'callable': determine_name(self.func),
            'args': self.func_args,
            'kwargs': self.func_kwargs,
        }
        raw_json = json.dumps(data)
```

```
        return bz2.compress(raw_json)

    @classmethod
    def deserialize(cls, data):
        raw_json = bz2.decompress(data)
        data = json.loads(data)

        task = cls(
            task_id=data['task_id'],
            retries=data['retries'],
            async=data['async']
        )

        func = import_attr(data['module'], data['callable'])
        task.to_call(func, *data.get('args', []), **data.get('kwargs', {}))
        return task
```

Now that we have our `SkinnyTask`, all we need is to use it. Each `Gator` instance supports a `task_class=...` keyword argument to replace the class used. So we'd do:

```
from alligator import Gator

from myapp.skinnytask import SkinnyTask


gator = Gator('redis://localhost:6379/0', task_class=SkinnyTask)
```

Every call to `gator.task(...)` or `gator.options(...)` will now use our `SkinnyTask`.

The last bit is that you can no longer use the included `latergator.py` script to process your queue. Instead, you'll have to manually run a `Worker`.

```
# myapp/skinnylatergator.py
from alligator import Gator, Worker

from myapp.skinnytask import SkinnyTask


gator = Gator('redis://localhost:6379/0', task_class=SkinnyTask)
# Now the worker will pick up the class as well.
worker = Worker(gator)
worker.run_forever()
```

Multiple Queues

If you have a high-volume site or the priority of tasks is important, the one main default queue (`alligator.constants.ALL`) may not work well. Fortunately, each `Gator` instance supports customizing the queue name it places tasks in.

Let's say that sending a notification email is way more important to use than creating thumbnails of photo uploads. We'll create two `Gator` instances, one for each type of processing.

```
from alligator import Gator

redis_dsn = 'redis://localhost:'
email_gator = Gator(redis_dsn, queue_name='notifications')
```

```
image_gator = Gator(redis_dsn, queue_name='images')


# Later...
email_gator.task(send_welcome_email, request.user.pk)
# And elsewhere...
image_gator.task(create_thumbnail, photo_path)
```

Now several large uploads won't block the sending of emails later in the queue. You will however now need to run more `Workers`. Just like the "Custom Task Classes" section, your `Worker` instances will need either `email_gator` or `image_gator` passed to them.

You could also fire up many `email_gator` workers (say 4) and just 1-2 `image_gator` workers if the number of tasks justifies it.

Custom Backend Clients

As of the time of writing, Alligator supports the following clients:

- Locmem
- Redis
- Beanstalk

However, if you have a different datastore or queue you'd like to use, you can write a custom backend `Client` to talk to that store. For example, let's write a naive version based on SQLite using the `sqlite3` module included with Python.

Warning: This code is simplistic for purposes of illustration. It's not thread-safe nor particularly suited to large loads. It's a demonstration of how you might approach things. Your Mileage May Vary.™

First, we need to create our custom `Client`. Where you put it doesn't matter much, as long as it is importable.

Each `Client` must have the following methods:

- `len`
- `drop_all`
- `push`
- `pop`
- `get`

```
# myapp/sqlite_backend.py
import sqlite3


class Client(object):
    def __init__(self, conn_string):
        # This is actually the filepath to the DB file.
        self.conn_string = conn_string
        # Kill the 'sqlite://' portion.
        path = self.conn_string.split('://', 1)[1]
        self.conn = sqlite3.connect(path)
```

```
    def _run_query(self, query, args):
        cur = self.conn.cursor()

        if not args:
            cur.execute(query)
        else:
            cur.execute(query, args)

        return cur

    def len(self, queue_name):
        query = 'SELECT COUNT(task_id) FROM `{}`'.format(queue_name)
        cur = self._run_query(query, [])
        res = cur.fetchone()
        return res[0]

    def drop_all(self, queue_name):
        query = 'DELETE FROM `{}`'.format(queue_name)
        self._run_query(query, [])

    def push(self, queue_name, task_id, data):
        query = 'INSERT INTO `{}` (task_id, data) VALUES (?, ?)'.format(
            queue_name
        )
        self._run_query(query, [task_id, data])
        return task_id

    def pop(self, queue_name):
        query = 'SELECT task_id, data FROM `{}` LIMIT 1'.format(queue_name)
        cur = self._run_query(query, [])
        res = cur.fetchone()

        query = 'DELETE FROM `{}` WHERE task_id = ?'.format(queue_name)
        self._run_query(query, [res[0]])

        return res[1]

    def get(self, queue_name, task_id):
        query = 'SELECT task_id, data FROM `{}` WHERE task_id = ?'.format(
            queue_name
        )
        cur = self._run_query(query, [task_id])
        res = cur.fetchone()

        query = 'DELETE FROM `{}` WHERE task_id = ?'.format(queue_name)
        self._run_query(query, [task_id])

        return res[1]
```

Now using it is simple. We'll make a `Gator` instance, passing our new class via the `backend_class=...` keyword argument.

```
from alligator import Gator

from myapp.sqlite_backend import Client as SQLiteClient


gator = Gator('sqlite:///tmp/myapp_queue.db', backend_class=SQLiteClient)
```

And use that `Gator` instance as normal!

Different Workers

The `Worker` class that ships with Alligator is somewhat opinionated & simple-minded. It assumes it will be used from a command-line & can print informational messages to `STDOUT`.

However, this may not work for your purposes. To work around this, you can subclass `Worker` (or make your own entirely new one).

For instance, let's make `Worker` use `logging` instead of `STDOUT`. We'll swap out all the methods that `print(...)` for methods that log instead.

```
# myapp/logworkers.py
import logging

from alligator import Worker


log = logging.getLogger('alligator.worker')


class LoggingWorker(Worker):
    def starting(self):
        ident = self.ident()
        log.info('{} starting & consuming "{}".'.format(ident, self.to_consume))

        if self.max_tasks:
            log.info('{} will die after {} tasks.'.format(ident, self.max_tasks))
        else:
            log.info('{} will never die.'.format(ident))

    def stopping(self):
        ident = self.ident()
        log.info('{} for "{}" shutting down. Consumed {} tasks.'.format(
            ident,
            self.to_consume,
            self.tasks_complete
        ))

    def result(self, result):
        # Because we don't usually care about the return values.
        log.debug(result)
```

As with previous `Worker` customizations, you won't be able to use `latergator.py` anymore. Instead, we'll make a script.

```
# myapp/logginggator.py
from alligator import Gator

from myapp.logworkers import LoggingWorker


gator = Gator('redis://localhost:6379/0')
worker = LoggingWorker(gator)
worker.run_forever()
```

And now there's no more nasty `STDOUT` messages!

Contributing

Alligator is open-source and, as such, grows (or shrinks) & improves in part due to the community. Below are some guidelines on how to help with the project.

Philosophy

- Alligator is BSD-licensed. All contributed code must be either
  - the original work of the author, contributed under the BSD, or...
  - work taken from another project released under a BSD-compatible license.
- GPL'd (or similar) works are not eligible for inclusion.
- Alligator's git master branch should always be stable, production-ready & passing all tests.
- Major releases (1.x.x) are commitments to backward-compatibility of the public APIs. Any documented API should ideally not change between major releases. The exclusion to this rule is in the event of a security issue.
- Minor releases (x.3.x) are for the addition of substantial features or major bugfixes.
- Patch releases (x.x.4) are for minor features or bugfixes.

Guidelines For Reporting An Issue/Feature

So you've found a bug or have a great idea for a feature. Here's the steps you should take to help get it added/fixed in Alligator:

- First, check to see if there's an existing issue/pull request for the bug/feature. All issues are at https://github.com/toastdriven/alligator/issues and pull reqs are at https://github.com/toastdriven/alligator/pulls.
- If there isn't one there, please file an issue. The ideal report includes:
  - A description of the problem/suggestion.
  - How to recreate the bug.
  - If relevant, including the versions of your:
    - Python interpreter
    - Web framework (if applicable)
    - Alligator
    - Optionally of the other dependencies involved
  - Ideally, creating a pull request with a (failing) test case demonstrating what's wrong. This makes it easy for us to reproduce & fix the problem. Instructions for running the tests are at *Alligator*

Guidelines For Contributing Code

If you're ready to take the plunge & contribute back some code/docs, the process should look like:

- Fork the project on GitHub into your own account.
- Clone your copy of Alligator.
- Make a new branch in git & commit your changes there.

- Push your new branch up to GitHub.
- Again, ensure there isn't already an issue or pull request out there on it. If there is & you feel you have a better fix, please take note of the issue number & mention it in your pull request.
- Create a new pull request (based on your branch), including what the problem/feature is, versions of your software & referencing any related issues/pull requests.

In order to be merged into Alligator, contributions must have the following:

- A solid patch that:
  - is clear.
  - works across all supported versions of Python.
  - follows the existing style of the code base (mostly PEP-8).
  - comments included as needed.
- A test case that demonstrates the previous flaw that now passes with the included patch.
- If it adds/changes a public API, it must also include documentation for those changes.
- Must be appropriately licensed (see "Philosophy").
- Adds yourself to the AUTHORS file.

If your contribution lacks any of these things, they will have to be added by a core contributor before being merged into Alligator proper, which may take additional time.

Security

Alligator takes security seriously. By default, it:

- does not access your filesystem in any way.
- only handles JSON-serializable data.
- only imports code available to your `PYTHONPATH`.

While no known vulnerabilities exist, all software has bugs & Alligator is no exception.

If you believe you have found a security-related issue, please **DO NOT SUBMIT AN ISSUE/PULL REQUEST**. This would be a public disclosure & would allow for 0-day exploits.

Instead, please send an email to "daniel@toastdriven.com" & include the following information:

- A description of the problem/suggestion.
- How to recreate the bug.
- If relevant, including the versions of your:
  - Python interpreter
  - Web framework (if applicable)
  - Alligator
  - Optionally of the other dependencies involved

Please bear in mind that I'm not a security expert/researcher, so a layman's description of the issue is very important.

Upon reproduction of the exploit, steps will be taken to fix the issue, release a new version & make users aware of the need to upgrade. Proper credit for the discovery of the issue will be granted via the AUTHORS file & other mentions.

CHAPTER 2

API Reference

alligator.constants

A set of constants included with `alligator`.

Task Constants

WAITING = `0`

SUCCESS = `1`

FAILED = `2`

DELAYED = `3`

RETRYING = `4`

Queue Constants

ALL = `all`

alligator.exceptions

exception `alligator.exceptions.`**`AlligatorException`**
A base exception for all Alligator errors.

exception `alligator.exceptions.`**`TaskFailed`**
Raised when a task fails.

exception `alligator.exceptions.`**`UnknownCallableError`**
Thrown when trying to import an unknown attribute from a module for a task.

exception `alligator.exceptions.`**`UnknownModuleError`**
Thrown when trying to import an unknown module for a task.

alligator.gator

class `alligator.gator.`**`Gator`**(*conn_string*, *queue_name='all'*, *task_class=<class 'alligator.tasks.Task'>*, *backend_class=None*)

`build_backend`(*conn_string*)
Given a DSN, returns an instantiated backend class.

Ex:

```
backend = gator.build_backend('locmem://')
# ...or...
backend = gator.build_backend('redis://127.0.0.1:6379/0')
```

Parameters **`conn_string`** (*string*) – A DSN for connecting to the queue. Passed along to the backend.

Returns A backend `Client` instance

`cancel`(*task_id*)
Takes an existing task & cancels it before it is processed.

Returns the canceled task, as that could be useful in creating a new task.

Ex:

```
task = gator.task(add, 18, 9)

# Whoops, didn't mean to do that.
gator.cancel(task.task_id)
```

Parameters **`task_id`** (*string*) – The identifier of the task to process

Returns The canceled `Task` instance

`execute`(*task*)
Given a task instance, this runs it.

This includes handling retries & re-raising exceptions.

Ex:

```
task = Task(async=False, retries=5)
task.to_call(add, 101, 35)
finished_task = gator.execute(task)
```

Parameters **`task_id`** (*string*) – The identifier of the task to process

Returns The completed `Task` instance

`get`(*task_id*)
Gets a specific task, by `task_id` off the queue & runs it.

Using this is not as performant (because it has to search the queue), but can be useful if you need to specifically handle a task *right now*.

Ex:

```
# Tasks were previously added, maybe by a different process or
# machine...
finished_task = gator.get('a-specific-uuid-here')
```

Parameters **task_id** (*string*) – The identifier of the task to process

Returns The completed `Task` instance

len()

Returns the number of remaining queued tasks.

Returns An integer count

options(***kwargs*)

Allows specifying advanced `Task` options to control how the task runs.

This returns a context manager which will create `Task` instances with the supplied options. See `Task.__init__` for the available arguments.

Ex:

```
def party_time(task, result):
    # Throw a party in honor of this task completing.
    # ...

with gator.options(retries=2, on_success=party_time) as opts:
    opts.task(increment, incr_by=2678)
```

Parameters **kwargs** (*dict*) – Keyword arguments to control the task execution

Returns An `Options` context manager instance

pop()

Pops a task off the front of the queue & runs it.

Typically, you'll favor using a `Worker` to handle processing the queue (to constantly consume). However, if you need to custom-process the queue in-order, this method is useful.

Ex:

```
# Tasks were previously added, maybe by a different process or
# machine...
finished_topmost_task = gator.pop()
```

Returns The completed `Task` instance

push(*task*, *func*, **args*, ***kwargs*)

Pushes a configured task onto the queue.

Typically, you'll favor using the `Gator.task` method or `Gator.options` context manager for creating a task. Call this only if you have specific needs or know what you're doing.

If the `Task` has the `async = False` option, the task will be run immediately (in-process). This is useful for development and in testing.

Ex:

```
task = Task(async=False, retries=3)
finished = gator.push(task, increment, incr_by=2)
```

Parameters

- **task** (A `Task` instance) – A mostly-configured task
- **func** (*callable*) – The callable with business logic to execute
- **args** (*list*) – Positional arguments to pass to the callable task
- **kwargs** (*dict*) – Keyword arguments to pass to the callable task

Returns The `Task` instance

task (*func*, **args*, ***kwargs*)

Pushes a task onto the queue.

This will instantiate a `Gator.task_class` instance, configure the callable & its arguments, then push it onto the queue.

You'll typically want to use either this method or the `Gator.options` context manager (if you need to configure the `Task` arguments, such as retries, async, task_id, etc.)

Ex:

```
on_queue = gator.task(increment, incr_by=2)
```

Parameters

- **func** (*callable*) – The callable with business logic to execute
- **args** (*list*) – Positional arguments to pass to the callable task
- **kwargs** (*dict*) – Keyword arguments to pass to the callable task

Returns The `Task` instance

class `alligator.gator.`**Options** (*gator*, ***kwargs*)

task (*func*, **args*, ***kwargs*)

Pushes a task onto the queue (with the specified options).

This will instantiate a `Gator.task_class` instance, configure the task execution options, configure the callable & its arguments, then push it onto the queue.

You'll typically call this method when specifying advanced options.

Parameters

- **func** (*callable*) – The callable with business logic to execute
- **args** (*list*) – Positional arguments to pass to the callable task
- **kwargs** (*dict*) – Keyword arguments to pass to the callable task

Returns The `Task` instance

alligator.tasks

class `alligator.tasks.`**`Task`**(*task_id=None*, *retries=0*, *async=True*, *on_start=None*, *on_success=None*, *on_error=None*, *depends_on=None*)

classmethod **`deserialize`**(*data*)

Given some data from the queue, deserializes it into a `Task` instance.

The data must be similar in format to what comes from `Task.serialize` (a JSON-serialized dictionary). Required keys are `task_id`, `retries` & `async`.

Parameters **`data`** (*`string`*) – A JSON-serialized string of the task data

Returns A populated task

Return type A `Task` instance

`run`()

Runs the task.

This fires the `on_start` hook function first (if present), passing the task itself.

Then it runs the target function supplied via `Task.to_call` with its arguments & stores the result.

If the target function succeeded, the `on_success` hook function is called, passing both the task & the result to it.

If the target function failed (threw an exception), the `on_error` hook function is called, passing both the task & the exception to it. Then the exception is re-raised.

Finally, the result is returned.

`serialize`()

Serializes the `Task` data for storing in the queue.

All data must be JSON-serializable in order to be stored properly.

Returns A JSON strong of the task data.

`to_call`(*func*, **args*, ***kwargs*)

Sets the function & its arguments to be called when the task is processed.

Ex:

```
task.to_call(my_function, 1, 'c', another=True)
```

Parameters

- **`func`** (*`callable`*) – The callable with business logic to execute
- **`args`** (*`list`*) – Positional arguments to pass to the callable task
- **`kwargs`** (*`dict`*) – Keyword arguments to pass to the callable task

`to_canceled`()

Sets the task's status as "canceled".

Useful for the `on_start/on_success/on_failed` hook methods for figuring out what the status of the task is.

`to_failed`()

Sets the task's status as "failed".

Useful for the `on_start/on_success/on_failed` hook methods for figuring out what the status of the task is.

to_retrying()
Sets the task's status as "retrying".

Useful for the `on_start/on_success/on_failed` hook methods for figuring out what the status of the task is.

to_success()
Sets the task's status as "success".

Useful for the `on_start/on_success/on_failed` hook methods for figuring out what the status of the task is.

to_waiting()
Sets the task's status as "waiting".

Useful for the `on_start/on_success/on_failed` hook methods for figuring out what the status of the task is.

alligator.utils

alligator.utils.**determine_module**(*func*)
Given a function, returns the Python dotted path of the module it comes from.

Ex:

```
from random import choice
determine_module(choice) # Returns 'random'
```

Parameters **func** (*function*) – The callable

Returns Dotted path string

alligator.utils.**determine_name**(*func*)
Given a function, returns the name of the function.

Ex:

```
from random import choice
determine_name(choice) # Returns 'choice'
```

Parameters **func** (*function*) – The callable

Returns Name string

alligator.utils.**import_attr**(*module_name*, *attr_name*)
Given a dotted Python path & an attribute name, imports the module & returns the attribute.

If not found, raises `UnknownCallableError`.

Ex:

```
choice = import_attr('random', 'choice')
```

Parameters

- **module_name** (*string*) – The dotted Python path
- **attr_name** (*string*) – The attribute name

Returns attribute

alligator.utils.**import_module**(*module_name*)

Given a dotted Python path, imports & returns the module.

If not found, raises `UnknownModuleError`.

Ex:

```
mod = import_module('random')
```

Parameters **module_name** (*string*) – The dotted Python path

Returns module

alligator.workers

class alligator.workers.**Worker**(*gator*, *max_tasks=0*, *to_consume='all'*, *nap_time=0.1*)

ident()

Returns a string identifier for the worker.

Used in the printed messages & includes the process ID.

interrupt()

Prints an interrupt message to stdout.

result(*result*)

Prints the received result from a task to stdout.

Parameters **result** – The result of the task

run_forever()

Causes the worker to run either forever or until the `Worker.max_tasks` are reached.

starting()

Prints a startup message to stdout.

stopping()

Prints a shutdown message to stdout.

CHAPTER 3

Release Notes

alligator 0.8.0

date 2015-01-02

This release adds the abililty to cancel a task before it is processed.

Features

- Added task cancellation. (SHA:*981503a*, SHA:*c3b26ee* & SHA:*34fe5ab*)

Bugfixes

- None

alligator 0.7.1

date 2015-01-02

This release adds lots of documentation & release notes, as well as bugfixes to make `Workers` less unnecessarily busy & fixes argument handling in `latergator.py`.

Features

- Tutorial improvements. (SHA:*981503a*, SHA:*c3b26ee* & SHA:*34fe5ab*)
- Added the Best Practices docs. (SHA:*0d9ea2e*)
- Filled in the Extending docs. (SHA:*b0cf130*, SHA:*1d472d5* & SHA:*8eb158c*)

Bugfixes

- Eventually fixed Travis CI support. (SHA:*f264b59*, SHA:*a8833d8*, SHA:*3f74b25*, SHA:*3968cef* & SHA:*f6358e4*)
- Fixed how unnecessarily busy `Worker.run_forever()` was. (SHA:*7b242fc*)
- Fixed the argument handling in `latergator.py`. (SHA:*18c5e0a*)

alligator 0.7.0

date 2015-01-01

This release added support for the Beanstalk queue, bringing the supported backends to three!

Features

- Added Beanstalk support. (SHA:*a7edf0c*)
- Added docs for the Beanstalk support. (SHA:*5c49994*)

Bugfixes

- None

alligator 0.6.0

date 2015-01-01

This was a bugfix release that added Travis CI support & corrected Python 3 errors in the Redis backend.

Features

- Added Travis CI integration for testing. (SHA:*a570391* & SHA:*cef31d3*)

Bugfixes

- Fixed the Python 3 support in the Redis client. (SHA:*d255427* & SHA:*58cdf3c*)

alligator 0.5.1

date 2015-01-01

This was a bugfix release that corrected some packaging issues with Alligator 0.5.0.

Features

- None

Bugfixes

- Fixed a packaging error that resulted in none of the backends being included. (SHA:*0e00b3b*)

alligator 0.5.0

date 2015-01-01

The initial public release of Alligator!

Features

- Working codes.
- Fully passing tests.
- Locmem support.
- Redis support.
- Install, tutorial & API docs.

Bugfixes

None

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index