Alligator Documentation

Release 0.5.1

Daniel Lindsley

January 01, 2015

Contents

Simple offline task queues. For Python.

“See you later, alligator.”

CHAPTER 1

Guide

1.1 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
```

1.1.1 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 can also install via other package managers:

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

# On Ubuntu
$ sudo aptitude install redis
```

Beanstalk

Support for beanstalk is coming in a future release.

1.2 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 Pyramid, pure WSGI applications, etc.

1.2.1 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).

1.2.2 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')
```

1.2.3 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
    )
```

```python
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.

1.2.4 Put the Task on the Queue

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.

```python
from alligator import Gator

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

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).

Then the only other change is to how we call `send_post_email`. Instead of calling it directly, we'll use `gator.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')
```

1.2.5 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.

1.2.6 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.

1.2.7 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.

1.2.8 Going Beyond

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

If you need to customize task behavior, using the `on_start/on_success/on_error` hook functions can go a long way, but you can also create your own `Task` classes.

If you need more scalability, you can use multiple queues (by passing `queue_name='...'` when initializing the `Gator` instance) & lots of workers to distribute/fan-out the tasks.

You can create custom backends to support other queues (by passing `backend_class=...` when initializing the `Gator` instance).

And you can use your own `Worker` subclasses to customize how you process tasks.

All these things are in the *Extending Alligator* docs, so when you need more advanced things, you should check that out.

Happy queuing!

1.3 Extending Alligator

1.3.1 Hook Methods

TBD

1.3.2 Custom Task Classes

TBD

1.3.3 Multiple Queues

TBD

1.3.4 Custom Backend Clients

TBD

1.3.5 Different Workers

TBD

1.4 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.

1.4.1 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.

1.4.2 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*

1.4.3 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.

1.5 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

2.1 alligator.constants

A set of constants included with `alligator`.

2.1.1 Task Constants

WAITING = `0`

SUCCESS = `1`

FAILED = `2`

DELAYED = `3`

RETRYING = `4`

2.1.2 Queue Constants

ALL = `all`

2.2 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.

2.3 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

`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

`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

2.4 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_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_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.

2.5 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

2.6 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.

`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

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

Index