Taskhawk Documentation

Release 1.4.1-dev

Aniruddha Maru

Nov 16, 2018

Contents

TaskHawk is a replacement for celery that works on AWS SQS/SNS, while keeping things pretty simple and straightforward. Any unbound function can be converted into a TaskHawk task.

For inter-service messaging, see Hedwig.

Only Python 3.6+ is supported currently.

This project uses semantic versioning

CHAPTER 1

Quickstart

1.1 Quickstart

Getting started with Taskhawk is easy, but requires a few steps.

1.1.1 Installation

Install the latest taskhawk release via *pip*:

```
$ pip install taskhawk
```

You may also install a specific version:

```
$ pip install taskhawk==1.0.0
```

The latest development version can always be found on Github.

1.1.2 Configuration

Before you can use Taskhawk, you need to set up a few settings. For Django projects, simple use Django settings to configure Taskhawk, for non-Django projects, you must declare an environment variable called `SETTINGS_MODULE` that points to a module where settings may be found.

Required settings are:

```
AWS_ACCESS_KEY = <YOUR AWS KEY>
AWS_ACCOUNT_ID = <YOUR AWS ACCOUNT ID>
AWS_REGION = <YOUR AWS REGION>
AWS_SECRET_KEY = <YOUR AWS SECRET KEY>

TASKHAWK_QUEUE = <YOUR APP TASKHAWK QUEUE>
```

1.1.3 Provisioning

Taskhawk works on SQS and SNS as backing queues. Before you can publish tasks, you need to provision the required infra. This may be done manually, or, preferably, using Terraform. Taskhawk provides tools to make infra configuration easier: see Taskhawk Terraform Generator for further details.

1.1.4 Using Taskhawk

To use taskhawk, simply add the decorator *taskhawk.task()* to your function:

```
@taskhawk.task
def send_email(to: str, subject: str, from_email: str = None) -> None:
    # send email
```

And then dispatch your function asynchronously:

```
send_email.dispatch('example@email.com', 'Hello!', from_email='example@spammer.com')
```

Tasks are held in SQS queue until they're successfully executed, or until they fail a configurable number of times. Failed tasks are moved to a Dead Letter Queue, where they're held for 14 days, and may be examined for further debugging.

1.1.5 Priority

Taskhawk provides 4 priority queues to use, which may be customized per task, or per message. For more details, see *taskhawk.Priority*.

CHAPTER 2

Usage

2.1 Usage Guide

2.1.1 Tasks

Add *taskhawk.task()* decorator to convert any unbound function into an async task, as shown here:

```
@taskhawk.task
def send_email(to: str, subject: str, from_email: str = None) -> None:
    # send email
```

Optionally, pass in `priority=taskhawk.Priority.high` to mark the task as a high priority task.

Task name is automatically inferred from decorated function module and name, but you can also set it explicitly with `name` parameter.

If your task function accepts an kwarg called `metadata` (of type `dict`) or `**kwargs`, the function will be called with a `metadata` parameter as a dict with the following attributes:

id: task identifier. This represents a run of a task.

priority: the priority a task was dispatched with. This will be same as task's priority, unless priority was customized on dispatch.

receipt: SQS receipt for the task. This may be used to extend message visibility if the task is running longer than expected using `taskhawk.extend_visibility_timeout`.

timestamp: task dispatch epoch timestamp (milliseconds)

version: message format version. Currently can only be 1.

If your task function accepts an kwarg called `headers` (of type `dict`) or `**kwargs`, the function will be called with a `headers` parameter which is dict that the task was dispatched with.

2.1.2 Publisher

You can run tasks asynchronously like so:

```
send_email.dispatch('example@email.com', 'Hello!', from_email='example@spammer.com')
```

If you want to include a custom headers with the message (for example, you can include a `request_id` field for cross-application tracing), or you want to customize priority, you can customize a particular task invocation using chaining like so:

```
send_email.with_headers(request_id='1234')\
          .with_priority(taskhawk.Priority.high)\
          .dispatch('example@email.com')
```

2.1.3 Consumer

A consumer for SQS based workers can be started as following:

```
taskhawk.listen_for_messages(taskhawk.Priority.high)
```

This is a blocking function, so if you want to listen to multiple priority queues, you'll need to run these on separate processes (don't use threads since this library is **NOT** guaranteed to be thread-safe).

A consumer for Lambda based workers can be started as following:

```
taskhawk.process_messages_for_lambda_consumer(lambda_event)
```

where `lambda_event` is the event provided by AWS to your Lambda function as described here.

If your tasks exist in different modules, ensure that your modules are imported before calling Taskhawk listener functions since tasks need to be registered before they can receive messages.

2.1.4 Internals

Message format

Internally, all tasks are converted into a message that looks like this:

```
{
    "id": "b1328174-a21c-43d3-b303-964dfcc76efc",
    "metadata": {
        "priority": "high",
        "timestamp": 1460868253255,
        "version": "1.0"
    },
    "headers": {
        ...
    },
    "task": "tasks.send_email",
    "args": [
        "email@automatic.com",
        "Hello!"
    ],
    "kwargs": {
        "from_email": "spam@example.com"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

2.2 Configuration

Add appropriate configuration to the app. If not using a Django app, ensure that *SETTINGS_MODULE* is defined to the path of a module where all settings can be found.

AWS_REGION

AWS region

required; string

AWS_ACCOUNT_ID

AWS account id

required; string

AWS_ACCESS_KEY

AWS access key

required; string

AWS_CONNECT_TIMEOUT_S

AWS connection timeout

optional; int; default: 2

AWS_ENDPOINT_SNS

AWS endpoint for SNS. This may be used to customized AWS endpoints to assist with testing, for example, using localstack.

optional; string

AWS_ENDPOINT_SQS

AWS endpoint for SQS. This may be used to customized AWS endpoints to assist with testing, for example, using localstack.

optional; string

AWS_READ_TIMEOUT_S

AWS read timeout

optional; int; default: 2

AWS_SECRET_KEY

AWS secret key

required; string

AWS_SESSION_TOKEN

AWS session token that represents temporary credentials (for example, for Lambda apps)

optional; string

IS_LAMBDA_APP

Flag indicating if this is a Lambda app

optional; string; default: False

TASKHAWK_DEFAULT_HEADERS

A function that may be used to inject custom headers into every message, for example, request id. This hook is called right before dispatch, and any headers that are explicitly specified when dispatching may override these headers.

If specified, it's called with the following arguments:

```
default_headers(task=task)
```

where `task` is the task function, and its expected to return a dict of strings.

It's recommended that this function be declared with `**kwargs` so it doesn't break on new versions of the library.

optional; fully-qualified function name

TASKHAWK_PRE_PROCESS_HOOK

A function which can used to plug into the message processing pipeline *before* any processing happens. This hook may be used to perform initializations such as set up a global request id based on message headers. If specified, this will be called with the following arguments for SQS apps:

```
pre_process_hook(queue_name=queue_name, sqs_queue_message=sqs_queue_message)
```

where `sqs_queue_message` is of type `boto3.sqs.Message`. And for Lambda apps as so:

```
pre_process_hook(sns_record=record)
```

where `sns_record` is a `dict` of a single record with format as described in lambda_sns_format.

It's recommended that this function be declared with `**kwargs` so it doesn't break on new versions of the library.

optional; fully-qualified function name

TASKHAWK_POST_PROCESS_HOOK

Same as `TASKHAWK_PRE_PROCESS_HOOK` but executed after task processing.

TASKHAWK_QUEUE

The name of the taskhawk queue (exclude the `TASKHAWK-` prefix).

required; string

TASKHAWK_SYNC

Flag indicating if Taskhawk should work synchronously. This is similar to Celery's Eager mode and is helpful for integration testing.

optional; bool; default False

TASKHAWK_TASK_CLASS

The name of a class to use as Task class rather than the default `taskhawk.Task`. This may be used to customize the behavior of tasks.

optional; fully-qualified class name

2.3 API reference

taskhawk.**listen_for_messages**(*priority: taskhawk.models.Priority*, *num_messages: int = 1*, *visibility_timeout_s: int = None*, *loop_count: int = None*, *shutdown_event: threading.Event = None*) → None

Starts a taskhawk listener for message types provided and calls the task function with given *args* and *kwargs*.

If the task function accepts a param called *metadata*, it'll be passed in with a dict containing the metadata fields: id, timestamp, version, receipt.

The message is explicitly deleted only if task function ran successfully. In case of an exception the message is kept on queue and processed again. If the task function keeps failing, SQS dead letter queue mechanism kicks in and the message is moved to the dead-letter queue.

This function is blocking by default. It may be run for specific number of loops by passing *loop_count*. It may also be stopped by passing a shut down event object which can be set to stop the function.

Parameters

- **priority** – The priority queue to listen to
- **num_messages** – Maximum number of messages to fetch in one SQS API call. Defaults to 1
- **visibility_timeout_s** – The number of seconds the message should remain invisible to other queue readers. Defaults to None, which is queue default
- **loop_count** – How many times to fetch messages from SQS. Default to None, which means loop forever.
- **shutdown_event** – An event to signal that the process should shut down. This prevents more messages from being de-queued and function exits after the current messages have been processed.

taskhawk.**process_messages_for_lambda_consumer**(*lambda_event: dict*) → None

Process messages for a Taskhawk consumer Lambda app, and calls the task function with given *args* and *kwargs*

If the task function accepts a param called *metadata*, it'll be passed in with a dict containing the metadata fields: id, timestamp, version, receipt.

In case of an exception, the message is kept on Lambda's retry queue and processed again a fixed number of times. If the task function keeps failing, Lambda dead letter queue mechanism kicks in and the message is moved to the dead-letter queue.

taskhawk.**task**(**args*, *priority: taskhawk.models.Priority = <Priority.default: 1>*, *name: Optional[str] = None*) → Callable

Decorator for taskhawk task functions. Any function may be converted into a task by adding this decorator as such:

```
@taskhawk.task
def send_email(to: str, subject: str, from: str = None) -> None:
    ...
```

Additional methods available on tasks are described by *taskhawk.Task* class

class taskhawk.**Task**(*fn: Callable*, *priority: taskhawk.models.Priority*, *name: str*)

Represents a Taskhawk task. This class provides methods to dispatch tasks asynchronously, You can also chain customizations such as:

```
send_email.with_headers(request_id='1234')\
          .with_priority(taskhawk.Priority.high)\
          .dispatch('example@email.com')
```

These customizations may also be saved and re-used multiple times

```
send_email_high_priority = send_email.with_priority(taskhawk.Priority.high)

send_email_high_priority.dispatch('example@email.com')

send_email_high_priority.with_headers(request_id='1234')\
                        .dispatch('example@email.com')
```

dispatch(**args*, ***kwargs*) → None
Dispatch task for async execution

Parameters

- **args** – arguments to pass to the task
- **kwargs** – keyword args to pass to the task

with_headers(***headers*) → taskhawk.task_manager.AsyncInvocation
Create a task invocation that uses custom headers

Parameters **headers** – Arbitrary headers

Returns an invocation that uses custom headers

with_priority(*priority: taskhawk.models.Priority*) → taskhawk.task_manager.AsyncInvocation
Create a task invocation with custom priority

Parameters **priority** – Custom priority to attach to this invocation

Returns an invocation that uses custom priority

class taskhawk.**Priority**
Priority of a task. This may be used to differentiate batch jobs from other tasks for example.

High and low priority queues provide independent scaling knobs for your use-case.

default = 1
This is the default priority of a task if nothing is specified. In most cases, using just the default queue should work fine.

high = 2

low = 3

bulk = 4
Bulk queue will typically have different monitoring, and may be used for bulk jobs, such as sending push notifications to all users. This allows you to effectively throttle the tasks.

taskhawk.**requeue_dead_letter**(*priority: taskhawk.models.Priority*, *num_messages: int = 10*, *visibility_timeout: int = None*) → None
Re-queues everything in the Taskhawk DLQ back into the Taskhawk queue.

Parameters

- **priority** – The priority queue to listen to
- **num_messages** – Maximum number of messages to fetch in one SQS call. Defaults to 10.

- **visibility_timeout** – The number of seconds the message should remain invisible to other queue readers. Defaults to None, which is queue default

taskhawk.**extend_visibility_timeout**(*priority: taskhawk.models.Priority*, *receipt: str*, *visibility_timeout_s: int*) → None

Extends visibility timeout of a message on a given priority queue for long running tasks.

2.3.1 Exceptions

class taskhawk.**RetryException**

Special exception that does not log an exception when it is received. This is a retryable error.

class taskhawk.**LoggingException**(*message*, *extra=None*)

An exception that allows passing additional logging info. *extra* must be a dict that will be passed to *logging.exception* and can be used by a logging adaptor etc.

class taskhawk.**IgnoreException**

Indicates that this task should be ignored.

class taskhawk.**ValidationError**

Message failed JSON schema validation

class taskhawk.**ConfigurationError**

There was some problem with settings

class taskhawk.**TaskNotFound**

No task was found that can handle the given message. Ensure that you call *taskhawk.RegisterTask*.

2.4 Release Notes

Current version: v1.4.1-dev

2.4.1 v1.0

- Initial version

2.5 Taskhawk Migration Guide

2.5.1 CELERY → v1

Assuming publishers and workers are completely independent processes:

1. Remove all celery task decorators from your task functions and replace them with *taskhawk.task()*.
2. Remove all celery related settings from your project.
3. Provision infra required for taskhawk using and taskhawk_terraform_generator, or manually.
4. Add new processes for workers on each priority queue that your app publishes to (not all queues may be relevant for your app).
5. Deploy Taskhawk worker processes (not publishers).
6. Verify that Taskhawk workers pick up message by sending a test message.

7. Deploy publisher processes.
8. Let Celery queues drain to 0.
9. Terminate Celery worker processes.

If Celery workers also publish async tasks:

1. Remove all celery task decorators from your task functions and replace them with `taskhawk.task()`.
2. Remove all celery related settings from your project.
3. Provision infra required for taskhawk using and taskhawk_terraform_generator, or manually.
4. Add new processes for workers on each priority queue that your app publishes to (not all queues may be relevant for your app).
5. Deploy a test TaskHawk worker process.
6. Verify that Taskhawk workers pick up message by sending a test message.
7. Double publish to both Taskhawk and Celery in Celery workers.
8. Deploy Taskhawk worker processes (not other publishers).
9. Deploy other publisher processes.
10. Remove double publish in Celery workers.
11. Deploy Celery workers.
12. Let Celery queues drain to 0.
13. Terminate Celery worker processes.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index