Fennel

Matt Westcott

Aug 20, 2020

CONTENTS

CHAPTER ONE

FEATURES

Fennel is a task queue for Python 3.7+ based on Redis Streams with a Celery-like API.

- Supports both sync (e.g. Django, Flask) and async (e.g. Starlette, FastAPI) code.
- Sane defaults: at least once processing semantics, tasks acknowledged on completion.
- Automatic retries with exponential backoff for fire-and-forget jobs.
- Clear task statuses available (e.g. sent, executing, success).
- Automatic task discovery (defaults to using `**/tasks.py`).
- Exceptionally small and understandable codebase.

Note: This is an *alpha* release. The project is under development, breaking changes are likely.

CHAPTER

TWO

INSTALLATION

```
pip install fennel
```

CHAPTER

THREE

BASIC USAGE

Run Redis and then write your code in `tasks.py`:

```
from fennel import App

app = App(name='myapp', redis_url='redis://127.0.0.1')


@app.task
def foo(n):
    return n


# Enqueue a task to be executed in the background by a fennel worker process.
foo.delay(7)
```

Meanwhile, run the worker:

```
$ fennel worker --app tasks:app
```

CHAPTER FOUR

ASYNCHRONOUS API

Fennel also supports an async API. If your code is running in an event loop (e.g. via Starlette or FastAPI), you will want to use the async interface instead:

```python
from fennel import App

app = App(name='myapp', redis_url='redis://127.0.0.1', interface='async')


@app.task
async def bar(x):
    return x


await bar.delay(5)
```

CHAPTER

FIVE

CONTENTS

5.1 Guide

Fennel is a task queue library for Python.

It enables you to register your functions as tasks, which you can then enqueue in Redis. In the background, worker processes will pull tasks from the queue and execute them. This follows the same basic pattern established by Celery and other task queue systems.

5.1.1 Interfaces

We support both sync and async interfaces to the fennel system. This means you can use it in traditional web frameworks (e.g Django, Flask), but also newer frameworks built on top of asyncio (e.g. Starlette, FastAPI, Quart). We default to `interface='sync'`, but you can select your interface as an option in your app configuration.

Sync

```python
import time

from fennel import App
from fennel.client import gather

app = App(name='myapp', redis_url='redis://127.0.0.1')


@app.task
def foo(n):
    time.sleep(n)
    return n


x = foo.delay(4)  # Enqueue a task to be executed in the background.
x.get()           # Waits for completion and returns 4.
```

Async

```
import asyncio

from fennel import App
from fennel.client.aio import gather

app = App(name='myapp', redis_url='redis://127.0.0.1', interface='async')


@app.task
async def foo(n):
    await asyncio.sleep(n)
    return n


x = await foo.delay(4)  # Enqueue a task to be executed in the background.
await x.get()           # Waits for completion and returns 4.
```

5.1.2 Two use-cases

1. Fire-and-forget

The most common way to use a task queue is to fire off background tasks while processing requests in a web application. You have some work that needs to happen (e.g. generating image thumbnails, sending an email), but you don't want the user to wait for it to complete before returning a response. In this case, it's important that the task succeeds, but your code will not be waiting to ensure that it does. If failures happen, you may want to automatically retry the task, or be notified through your monitoring system.

This is the default scenario expected by Fennel. We support it by automatically retrying tasks which fail, up to `fennel.settings.Settings.default_retries` times. Individual tasks can be configured via the `retries` kwarg:

```
@app.task(retries=3)
def foo(n):
    time.sleep(n)
    return n
```

Retries will occur on a schedule provided by the `retry_backoff` function. By default, Fennel will use an exponential backoff algorithm with jitter to avoid overloading the workers in case a large number of failures happen simulatenously. (See `fennel.utils.backoff()` for more details.)

If all retries are exhausted and the task still fails, it will be placed in the 'dead-letter queue', see *Error handling* and *The dead-letter queue* below for details.

2. Compose parallel pipelines

There is a second way to use a task queue: when you have a large amount of work you want to perform in parallel, perhaps on dedicated high-performance machines. In this case your code may want to wait for all tasks to complete before moving on to the next batch of work.

This scenario is also supported by Fennel. You should set `retries=0` on your task (or `default_retries=0` in your app instance). The waiting primitives we supply are:

1. *gather*, when you want all tasks to complete and collect the results.
2. *wait*, to wait for a specific duration before timing out.

Sync:

```
@app.task
sync def foo(n):
    time.sleep(n)
    return n


results = [foo.delay(i) for i in range(6)]

# Waits for completion and returns [1, 2, 3, 4, 5, 6].
gather(results)

# Instead, waits for 10 seconds, returns two sets of Futures.
done, pending = wait(results, timeout=10)
```

Async:

```
@app.task
async def foo(n):
    await asyncio.sleep(n)
    return n


results = [await foo.delay(i) for i in range(6)]

# Waits for completion and returns [1, 2, 3, 4, 5, 6]
await gather(results)

# Instead, waits for 10 seconds, returns two sets of Futures.
done, pending = await wait(results, timeout=10)
```

5.1.3 Error handling

Fennel considers a task to have failed if any exception is raised during its execution.

If a task has retries enabled, it will be scheduled according by the `retry_backoff` function. By default, Fennel will use an exponential backoff algorithm with jitter to avoid overloading the workers in case a large number of failures happen simulatenously (see *`fennel.utils.backoff()`* for more details). When retries are exhausted the task enters the dead-letter queue.

If you attempt to retrieve the result of a task that has failed, fennel will raise *`fennel.exceptions.TaskFailed`* with the original exception information attached:

```
>>> @app.task(retries=0)
>>> async def foo(n):
...     raise Exception("baz")
...
>>> x = await foo.delay(3)
...
>>> try:
...     result = await x.get()
>>> except TaskFailed as e:
...     assert e.original_type == "Exception"
...     assert e.original_args == ["baz"]
```

5.1.4 The dead-letter queue

The DLQ hold tasks which have failed and exhausted all their retry attempts. They now require manual intervention, for instance you may need to redeploy your applicaiton code to fix a bug before you replay the failed tasks.

You can read, replay, or purge the contents of the DLQ as follows:

```
$ fennel dlq read --app mymodule:myapp
$ fennel dlq replay --app mymodule:myapp
$ fennel dlq purge --app mymodule:myapp
```

If you need more granular control, the Fennel client library also provides functions to interact with the DLQ programmatically. For example you can replay all jobs matching certain criteria (using the async client):

```
>>> from fennel.client.aio import replay_dead
...
>>> from myapp.tasks import app  # <-- Your Fennel app instance
...
>>> replay_dead(app, filter=lambda job: job.task == "tasks.mytask")
[<Job>, ...]
```

To understand how jobs are represented internally, see *`fennel.job`*.

5.1.5 Workers

Workers are launched via the CLI:

```
$ fennel worker --app mymodule:myapp
```

You must specify the Python module and Fennel application instance whose tasks the worker will execute. See the *CLI* page for more information.

5.1.6 Logging

Fennel supports structured logging out of the box. You can choose whether to use a human-readable format, or JSON via `fennel.settings.Settings.log_format`

5.1.7 Limitations

1. Task args and kwargs must be JSON-serialisable.
2. Return values (if results storage is enabled) must be JSON-serialisable.
3. Processing order is not guaranteed (if you want to ensure all events for a given key are processed in-order, see https://github.com/mjwestcott/runnel).
4. Tasks will be processed at least once (we acknowledge the underlying messages when a task returns without an exception, so any failures before then will happen again when retried).

This is similar to the approach taken by Celery, Dramatiq, and task queues in other languages. As a result, you are advised to follow these best-practices:

- Keep task arguments and return values small (e.g. send the user_id not the User model instance)
- Ensure that tasks are idempotent – if you process them more than once, the same result will occur.

Also, Redis is not a highly durable database system – it's durability is configurable and limited. You are advised to read the related parts of the Redis documentation.

This is a notable section of the Streams Intro:

> - AOF must be used with a strong fsync policy if persistence of messages is important in your application.
> - By default the asynchronous replication will not guarantee that XADD commands or consumer groups state changes are replicated: after a failover something can be missing depending on the ability of slaves to receive the data from the master.
> - The WAIT command may be used in order to force the propagation of the changes to a set of slaves. However note that while this makes very unlikely that data is lost, the Redis failover process as operated by Sentinel or Redis Cluster performs only a best effort check to failover to the slave which is the most updated, and under certain specific failures may promote a slave that lacks some data.
>
> So when designing application using Redis streams and consumer groups, make sure to understand the semantical properties your application should have during failures, and configure things accordingly, evaluating if it is safe enough for your use case.

5.2 Installation

```
pip install fennel
```

Fennel is tested on Python 3.7+

5.3 Motivation

Python needs an async/await compatible task queue library.

(And Celery is perhaps past its prime.)

5.4 Architecture

5.4.1 Fundamentals

Fennel's architecture is similar to other job queue systems like Celery, Dramatiq, RQ:

```
         +-Redis--------------------+
         |                          |
         |  * The Job Queue         |
         |  * The Dead-Letter Queue |
         |  * The Schedule          |
         |  * Results Storage       |
         |  * Job Metadata          |
         |  * Worker State          |
         |                          |
         +--------------------------+
             ^                 |
             | send            | receive
             | jobs            | jobs
             |                 |
             |                 v
+---------------------+   +---------------------+
|                     |   |                     |
|  Your Application   |   |    Fennel Worker    |
|                     |   |                     |
+---------------------+   +---------------------+
```

When your application sends jobs via *fennel.client.Task.delay()*, they are persisted in Redis. Meanwhile a background worker process is waiting to receive jobs and execute them using the Python function decorated with *fennel.App.task()*.

In the normal course of events, the job will be added to a Redis Stream (to notify workers) and a Redis Hash (to store metadata such as the current status and number of retries to perform). When execution is finished, the return value will be persisted in a Redis List (to allow workers to block awaiting it's arrival) and set to expire after a configurable duration (`fennel.settings.Settings.result_ttl`).

In case of execution failure (meaning an exception is raised), if the job is configured for retries it will be scheduled in a Redis Sorted Set (so workers can poll to discover jobs whose ETA has elapsed). If retries are exhausted, the job will be added to the dead-letter queue (another Redis Stream). From there, manual intervention is required to either purge or replay the job.

5.4.2 Redis Streams

Under the hood, Fennel uses Redis Streams as the fundamental 'queue' data structure. This provides useful functionality for distributing jobs to individual workers and keeping track of which tasks have been read and later acknowledged.

Our use of Streams is arguably non-standard. The expectation is that messages accumulate in the stream, which is periodically trimmed to some maximum length governed by memory limits. In our case, we don't need to maintain a long history of messages in memory and we don't want the trim operation to remove any unacknowledged meessages, so we take advantage of the *XDEL* operation and delete messages when they are acknowledged, like a traditional job queue.

5.4.3 The Worker

Workers are launched via the *CLI*. Below is a diagram representing a worker with the settings `processes=2` and `concurrency=8`:

```
+-Worker---------------------------------------------------------------------+
|                                                                             |
|   +-Executor----------------------+     +-Executor----------------------+   |
|   |                               |     |                               |   |
|   |    8x consumer coroutines     |     |    8x consumer coroutines     |   |
|   |                               |     |                               |   |
|   |    1x heartbeat coroutine     |     |    1x heartbeat coroutine     |   |
|   |                               |     |                               |   |
|   |    1x maintenance coroutine   |     |    1x maintenance coroutine   |   |
|   |                               |     |                               |   |
|   |    1x scheduler coroutine     |     |    1x scheduler coroutine     |   |
|   |                               |     |                               |   |
|   +-------------------------------+     +-------------------------------+   |
|                                                                             |
+-----------------------------------------------------------------------------+
```

The worker process itself simply spawns 2 executor processes and monitors their health. The executors themselves run 8 consumer coroutines which are responsible for waiting to receive jobs from the queue and then executing them. If the job is a coroutine function, it is awaited in the running asyncio event loop, otherwise it is run in a *ThreadPoolExecutor* so as not to block the loop.

The other coroutines maintain the health of the system by publishing heartbeats, polling for scheduled jobs, and responding to the death of other workers or executors.

CPU-bound tasks benefit from multiple processes. We default to running `multiprocessing.cpu_count()` executors for this reason. IO-bound tasks will benefit from high executor concurrency and we default to running 8 consumer coroutines in each executor.

5.4.4 Job Lifecycle

Python functions become tasks when they are decorated with *`fennel.App.task()`*. When they are enqueued using *`fennel.client.Task.delay()`*, they become jobs in the Fennel queue.

Jobs transition between a number of statuses according to the logic below:

```
                                                              +-----------+
                                                              |           |
                                                              |           |
                                                          5   |  SUCCESS  |
```

(continues on next page)

(continued from previous page)

```
+-----------+      +-----------+      +-----------+    +---->|           |
|           |      |           |      |           |    |     |           |
|           |  1   |           |  2   |           |    |     +-----------+
|  UNKNOWN  |----->|   SENT    |----->| EXECUTING |----+
|           |      |           |      |           |    |
|           |      |           |      |           |    |     +-----------+
+-----------+      +-----------+      +-----------+    +---->|           |
                                          |   ^           6  |           |
                                          |   |              |   DEAD    |
                                        3 |   | 4            |           |
                                          |   |              |           |
                                          v   |              +-----------+
                                      +-----------+
                                      |           |
                                      |           |
                                      |   RETRY   |
                                      |           |
                                      |           |
                                      +-----------+
```

1. Client code sends a job to the queue via `fennel.client.Task.delay()`.
2. A worker reads the job from the queue and begins executing it.
3. Execution fails (an exception was raised) and the job's max_retries has not been exceeded. The job is placed in the schedule (a Redis sorted set), which workers periodically poll.
4. A job is pulled from the schedule and execution is attempted again. (This can repeat many times.)
5. Execution succeeds (no exceptions raised).
6. Execution fails (an exception was raised) and retries have been exhausted, so the job is now in the dead-letter queue where it will remain until manual intervention (via the CLI or client code).

Job status can be retrieved via the AsyncResult object:

```
>>> import time
>>> from fennel import App
...
>>> app = App(name='myapp')
...
>>> @app.task
>>> def foo(n):
...     time.sleep(n)
...     return n
...
>>> x = foo.delay(4)
>>> x.status()
SENT
>>> # Wait a few moments.
>>> x.status()
EXECUTING
>>> # Wait for completion.
>>> x.get()
4
>>> x.status()
SUCCESS
```

5.5 CLI

5.5.1 fennel

```
fennel [OPTIONS] COMMAND [ARGS]...
```

dlq

Interact with the dead-letter queue. Choices for the *action* argument:

* read - Print all tasks from the dead-letter queue to stdout.
* replay - Move all tasks from the dead-letter queue back to the main task queue for reprocessing.
* purge - Remove all tasks from the dead-letter queue forever.

```
fennel dlq [OPTIONS] [read|replay|purge]
```

Options

-a, --app <application>
    Required

Arguments

ACTION
    Required argument

info

Print a JSON-encoded summary of application state.

```
fennel info [OPTIONS]
```

Options

-a, --app <application>
    Required

task

Print a JSON-encoded summary of job information.

```
fennel task [OPTIONS]
```

Options

-a, --app <application>
    Required

-u, --uuid <uuid>
    Required

worker

Run the worker.

```
fennel worker [OPTIONS]
```

Options

-a, --app <application>
    Required A colon-separated string identifying the *fennel.App* instance for which to run a worker.

    If a file foo.py exists at the current working directory with the following contents:

```
>>> from fennel import App
>>>
>>> app = App(name="myapp", redis_url="redis://127.0.0.1:6379")
>>>
>>> @app.task
>>> def f():
>>>     pass
```

    Then pass foo:app as the *app* option: $ fennel worker --app=foo:app

-p, --processes <processes>
    How many executor processes to run in each worker. Default multiprocessing.cpu_count()

-c, --concurrency <concurrency>
    How many concurrent consumers to run (we make at least this many Redis connections) in each executor process. The default, 8, can handle 160 req/s in a single worker process if each task is IO-bound and lasts on average 50ms. If you have long running CPU-bound tasks, you will want to run multiple executor processes. Default 8

5.6 API Reference

5.6.1 fennel

class `fennel.App`(*name: str*, ***kwargs*)

The app is the main abstraction provided by Fennel. Python functions are decorated via `@app.task` to enable background processing. All settings are configured on this object.

Parameters

- **name** (*str*) – Used to identify this application, e.g. to set which tasks a worker will execute.
- **kwargs** – Any settings found in *fennel.settings.Settings*

Examples

```
>>> from fennel import App
...
>>> app = App(
...     name='myapp',
...     redis_url='redis://127.0.0.1',
...     default_retries=3,
...     results_enabled=True,
...     log_level='info',
...     log_format='json',
...     autodiscover='**/tasks.py',
...     interface='sync',
... )
...
>>> @app.task(retries=1)
>>> def foo(x):
...     return x
...
>>> x = foo.delay(7)  # Execute in the background.
>>> x
AsyncResult(uuid=Tjr75jM3QDOHoLTLyrsY1g)
>>> x.get()  # Wait for the result.
7
```

If your code is running in an asynchronous event loop (e.g. via Starlette, FastAPI, Quart), you will want to use the async interface instead:

```
>>> import asyncio
...
>>> app = App(name='foo', interface='async')
...
>>> @app.task
>>> async def bar(x)
...     await asyncio.sleep(x)
...     return x
...
>>> x = await bar.delay(5)
>>> await x.status()
SENT
>>> await x.get()
5
```

task (*func: Callable = None*, *, *name=None*, *retries=<object object>*) → Any

A decorator to register a function with the app to enable background processing via the task queue.

The worker process (see *fennel.worker.worker*) will need to discover all registered tasks on startup. The means all the modules containing tasks need to be imported. Fennel will import modules found via `fennel.settings.Settings.autodiscover`, which by default is `'**/tasks.py'`.

Parameters

- **func** (*Callable*) – The decorated function.
- **name** (*str*) – The representation used to uniquely identify this task.
- **retries** (*int*) – The number of attempts at execution after a task has failed (meaning raised any exception).

Examples

Exposes an interface similar to Celery:

```
>>> @app.task(retries=1)
>>> def foo(x):
...     return x
```

Tasks can be enqueued for processing via:

```
>>> foo.delay(8)
AsyncResult(uuid=q_jb6KaUT-G4tOAoyQ0yaA)
```

The can also be called normally, bypassing the Fennel system entirely:

```
>>> foo(3)
3
```

By default, tasks are 'fire-and-forget', meaning we will not wait for their completion. They will be executed by worker process and will be retried automatically on failure (using exponential backoff), so we assume tasks are idempotent.

You can also wait for the result:

```
>>> x = foo.delay(4)
>>> x.status()
SENT
>>> x.get(timeout=10)
4
```

If instead you have many tasks and wish to wait for them to complete you can use the waiting primitives provided (you will want to ensure all tasks have retries=0, which you can set by default with an app setting):

```
>>> from fennel.client import gather, wait
>>> results = [foo.delay(x) for x in range(10)]
>>> gathered = gather(results)  # Or:
>>> done, pending = wait(results, timeout=2)
```

If your application is running in an event loop you can elect to use the async interface for your fennel app (see `fennel.settings.Settings.interface`), which uses *aioredis* under the hood to enqueue items, retrieve results, etc, so you will need to await those coroutines:

```
>>> app = App(name='foo', interface='async')
>>>
>>> @app.task
>>> async def bar(x)
...     await asyncio.sleep(x)
>>>
>>> x = await bar.delay(1)
>>> await x.status()
SUCCESS
```

5.6.2 fennel.settings

class fennel.settings.**Settings**

Settings can be configured via environment variables or keyword arguments for the fennel.App instance (which take priority).

Examples

For environment variables, the prefix is `FENNEL_`, for instance:

```
FENNEL_REDIS_URL=redis://127.0.0.1:6379
FENNEL_DEFAULT_RETRIES=3
FENNEL_RESULTS_ENABLED=true
```

Or via App kwargs:

```
>>> from fennel import App
...
>>> app = App(
...     name='myapp',
...     redis_url='redis://127.0.0.1',
...     default_retries=3,
...     results_enabled=True,
...     log_level='info',
...     log_format='json',
...     autodiscover='**/tasks.py',
...     interface='sync',
... )
```

Parameters

- **redis_url** (*str*) – Redis URL. Default `'redis://127.0.0.1:6369'`
- **interface** (*str*) – Which client interface should we use – sync or async? Default `'sync'`
- **processes** (*int*) – How many executor processes to run in each worker. Default `multiprocessing.cpu_count()`
- **concurrency** (*int*) – How many concurrent consumers to run (we make at least this many Redis connections) in each executor process. The default, 8, can handle 160 req/s in a single executor process if each task is IO-bound and lasts on average 50ms. If you have

long running CPU-bound tasks, you will want to run multiple executor processes (and set heartbeat_timeout to greater than your maximum expected task duration). Default `8`

- **`default_retries`** (*`int`*) – How many times to retry a task in case it raises an exception during execution. With 10 retries and the default *`fennel.utils.backoff()`* function, this will be approximately 30 days of retries. Default `10`
- **`retry_backoff`** (*`Callable`*) – Which algorithm to use to determine the retry schedule. The default is exponential backoff via *`fennel.utils.backoff()`*.
- **`read_timeout`** (*`int`*) – How many milliseconds to wait for messages in the main task queue. Default `4000`
- **`prefetch_count`** (*`int`*) – How many messages to read in a single call to *XREADGROUP*. Default `1`
- **`heartbeat_timeout`** (*`float`*) – How many seconds before an executor is considered dead if heartbeats are missed. If you have long-running CPU-bound tasks, this value should be greater than your maximum expected task duration. Default `60`
- **`heartbeat_interval`** (*`float`*) – How many seconds to sleep between heartbeats are stored for each executor process. Default `6`
- **`schedule_interval`** (*`float`*) – How many seconds to sleep between polling for scheduled tasks. Default `4`
- **`maintenance_interval`** (*`float`*) – How many seconds to sleep between running the maintenance script. Default `8`
- **`task_timeout`** (*`int`*) – How long to wait for results to be computed when calling .get(), seconds. Default `10`
- **`grace_period`** (*`int`*) – How many seconds to wait for in-flight tasks to complete before forcefully exiting. Default: `30`
- **`restults_enabled`** (*`bool`*) – Whether to store results. Can be disabled if your only use-case is ‘fire-and-forget’. Default `True`
- **`results_ttl`** (*`int`*) – How long before expiring results in seconds. Default `3600` (one hour).
- **`log_format`** (*`str`*) – Whether to pretty print a human-readable log (“console”) or JSON (“json”). Default `'console'`
- **`log_level`** (*`str`*) – The minimum log level to emit. Default `'debug'`
- **`autodiscover`** (*`str`*) – The pattern for `pathlib.Path.glob()` to find modules containing task-decorated functions, which the worker must import on startup. Will be called relative to current working directory. Can be set to the empty string to disable. Default `'**/tasks.py'`

5.6.3 fennel.worker

`fennel.worker.worker.`**`start`**(*app*, *exit='signal'*)

The main entrypoint for the worker.

The worker will create and monitor *N* `fennel.worker.Executor` processes. Each *Executor* will spawn *M* coroutines via an asyncio event loop. *N* and *M* are controlled by `fennel.settings.Settings.processes` and `fennel.settings.Settings.concurrency` respectively.

CPU-bound tasks benefit from multiple processes. IO-bound tasks will benefit from high executor concurrency.

Parameters

- **app** (`fennel.App`) – The application instance for which to start a background worker.
- **exit** (*str*) – The exit strategy. *EXIT_SIGNAL* is used when the worker should only stop on receipt of a interrupt or termination signal. *EXIT_COMPLETE* is used in tests to exit when all tasks from the queue have completed.

Notes

signal.SIGINT and *signal.SIGTERM* are handled by gracefully shutting down, which means giving the executor processes a chance to finish their current tasks.

class `fennel.worker.executor.`**`Executor`**(*app*)

The *Executor* is responsible for reading jobs from the Redis queue and executing them.

Heartbeats are sent from the executor periodically (controlled by `fennel.settings.Settings.heartbeat_interval`). If they are missing for more than `fennel.settings.Settings.heartbeat_timeout` seconds, the executor will be assumed dead and all of its pending messages will be reinserted to the stream by another worker's maintenance function.

Parameters **app** (`fennel.App`) – The application instance for which to start an *Executor*.

`start`(*exit: str = 'signal'*, *queue: multiprocessing.context.BaseContext.Queue = None*) → None

Begin the main executor loop.

Parameters

- **exit** (*str*) – The exit strategy. *EXIT_SIGNAL* is used when the worker should only stop on receipt of a interrupt or termination signal. *EXIT_COMPLETE* is used in tests to exit when all tasks from the queue have completed.
- **queue** (*multiprocessing.Queue*) – A *QueueHandler* will be used to send logs to this queue to avoid interleaving from multiple processes.

Notes

Intended to run via *`fennel.worker.worker.start()`* which will supervise multiple *Executor* processes.

signal.SIGINT and *signal.SIGTERM* are handled by gracefully shutting down, which means giving the executor processes a chance to finish their current tasks.

`is_running`()

5.6.4 fennel.client

A collection of synchronous classes and functions to interact with the Fennel system.

`fennel.client.`**`purge_dead`**(*app*, *filter=<function <lambda>>*, *batchsize=100*)
Iterate over the dead-letter queue and delete any jobs for which filter(job) evaluates to True. The default is to delete all jobs.

`fennel.client.`**`read_dead`**(*app*, *batchsize=100*)
Iterate over the dead-letter queue and return all job data.

`fennel.client.`**`replay_dead`**(*app*, *filter=<function <lambda>>*, *batchsize=100*)
Iterate over the dead-letter queue and replay any jobs for which filter(job) evaluates to True. The default is to replay all jobs.

`class` `fennel.client.`**`AsyncResult`**(*job: fennel.job.Job*, *app*)
A handle for a task that is being processed by workers via the task queue.

Conceptually similar to the *AsyncResult* from the mutliprocessing library.

`status`()
Return the status of the task execution.

Examples

```
>>> @app.task
>>> def bar(x)
...     time.sleep(x)
...     return x
...
>>> x = bar.delay(5)
>>> x.status()
SENT
>>> x.status()  # After roughly 5 seconds...
SUCCESS
```

`get`(*timeout: int = <object object>*) → Any
Wait for the result to become available and return it.

Raises

- ***`fennel.exceptions.TaskFailed`*** – If the original function raised an exception.
- ***`fennel.exceptions.Timeout`*** – If > *timeout* seconds elapse before a result is available.

Examples

```
>>> @app.task(retries=0)
>>> def foo(x):
...     return x
...
>>> x = foo.delay(7)
>>> x.get()  # Wait for the result.
7
```

> **Warning:** You must have results storage enabled (`fennel.settings.Settings.results_enabled`)
>
> If you have retries enabled, they may be rescheduled many times, so you may prefer to use retries=0 for tasks whose result you intend to wait for.

class `fennel.client.`**`Task`**(*name: str*, *func: Callable*, *retries: int*, *app*)

`delay`(**args: Any*, ***kwargs: Any*) → fennel.client.results.AsyncResult

Traditional Celery-like interface to enqueue a task for execution by the workers.

The *args* and *kwargs* will be passed through to the task when executed.

Examples

```
>>> @app.task
>>> def foo(x, bar=None):
...     time.sleep(x)
...     if bar == "mystr":
...         return False
...     return True
...
>>> foo.delay(1)
>>> foo.delay(2, bar="mystr")
```

`__call__`(**args: Any*, ***kwargs: Any*) → Any

Call the task-decorated function as a normal Python function. The fennel system will be completed by-passed.

Examples

```
>>> @app.task
>>> def foo(x):
...     return x
...
>>> foo(7)
7
```

`fennel.client.`**`gather`**(*results: Iterable[fennel.client.results.AsyncResult], task_timeout=10, return_exceptions=True*)

Multi-result version of .get() – wait for all tasks to complete and return all of their results in order.

Has the same semantics as *asyncio.gather*.

`fennel.client.`**`wait`**(*results: Iterable[fennel.client.results.AsyncResult], timeout: int, return_when='ALL_COMPLETED'*)

Wait for all tasks to complete and return two sets of Futures (done, pending).

Has the same semantics as *asyncio.wait*.

5.6.5 fennel.aio.client

A collection of asynchronous classes and functions, expected to be run in an asyncio-compatible event loop, to interact with the Fennel system.

async `fennel.client.aio.`**`purge_dead`**(*app*, *filter=<function <lambda>>*, *batchsize=100*)
Iterate over the dead-letter queue and delete any jobs for which filter(job) evaluates to True. The default is to delete all jobs.

async `fennel.client.aio.`**`read_dead`**(*app*, *batchsize=100*)
Iterate over the dead-letter queue and return all job data.

async `fennel.client.aio.`**`replay_dead`**(*app*, *filter=<function <lambda>>*, *batchsize=100*)
Iterate over the dead-letter queue and replay any jobs for which filter(job) evaluates to True. The default is to replay all jobs.

class `fennel.client.aio.`**`AsyncResult`**(*job: fennel.job.Job*, *app*)
A handle for a task that is being processed by workers via the task queue.

Conceptually similar to the AsyncResult from the mutliprocessing library.

async **`status`**()
Return the status of the task execution.

Examples

```
>>> @app.task
>>> async def bar(x)
...     await asyncio.sleep(x)
...     return x
...
>>> x = await bar.delay(5)
>>> await x.status()
SENT
>>> await x.status()  # After roughly 5 seconds...
SUCCESS
```

async **`get`**(*timeout: int = <object object>*) → Any
Wait for the result to become available and return it.

Raises

- ***`fennel.exceptions.TaskFailed`*** – If the original function raised an exception.
- ***`fennel.exceptions.Timeout`*** – If > *timeout* seconds elapse before a result is available.

Examples

```
>>> @app.task(retries=0)
>>> def foo(x):
...     return x
...
>>> x = await foo.delay(7)
>>> await x.get()  # Wait for the result.
7
```

> **Warning:** You must have results storage enabled (`fennel.settings.Settings.results_enabled`)
>
> If you have retries enabled, they may be rescheduled many times, so you may prefer to use retries=0 for tasks whose result you intend to wait for.

class `fennel.client.aio.`**`Task`**(*name: str*, *func: Callable*, *retries: int*, *app*)

async delay(**args: Any*, ***kwargs: Any*) → fennel.client.aio.results.AsyncResult
Enqueue a task for execution by the workers.

Similar to asyncio.create_task (but also works with non-async functions and runs on our Redis-backed task queue with distributed workers, automatic retry, and result storage with configurable TTL).

The *args* and *kwargs* will be passed through to the task when executed.

Examples

```
>>> @app.task(retries=1)
>>> async def foo(x, bar=None):
...     asyncio.sleep(x)
...     if bar == "mystr":
...         return False
...     return True
...
>>> await foo.delay(1)
>>> await foo.delay(2, bar="mystr")
```

__call__(**args: Any*, ***kwargs: Any*) → Any
Call the task-decorated function as a normal Python function. The fennel system will be completed bypassed.

Examples

```
>>> @app.task
>>> def foo(x):
...     return x
...
>>> foo(7)
7
```

async `fennel.client.aio.`**`gather`**(*results: Iterable[fennel.client.aio.results.AsyncResult], task_timeout=10, return_exceptions=True*)
Multi-result version of .get() – wait for all tasks to complete and return all of their results in order.

Has the same semantics as *asyncio.gather*.

async `fennel.client.aio.`**`wait`**(*results: Iterable[fennel.client.aio.results.AsyncResult], timeout: int, return_when='ALL_COMPLETED'*)
Wait for all tasks to complete and return two sets of Futures (done, pending).

Has the same semantics as *asyncio.wait*.

5.6.6 fennel.status

Jobs have a number of statuses through their lifecycle. This module contains the constants. If you have enqueued a task for execution, then you can obtain its status as follows:

```
>>> x = mytask.delay()
>>> x.status()
EXECUTING
```

fennel.status.**UNKNOWN = 'UNKNOWN'**
The job's status is not stored in Redis. Presumably no action has been taken on the job.

fennel.status.**SENT = 'SENT'**
The job has been sent to Redis, but execution has not yet started.

fennel.status.**EXECUTING = 'EXECUTING'**
A worker has received the job from the queue and has begun executing it.

fennel.status.**SUCCESS = 'SUCCESS'**
Execution was successful and the job's result is ready (if results storage is enabled).

fennel.status.**RETRY = 'RETRY'**
Execution was not successful (an exception was raised) and a retry is scheduled to occur in the future.

fennel.status.**DEAD = 'DEAD'**
Execution was not successful (an exception was raised) and retries have been exhausted, so the job is now in the dead-letter queue where it will remain until manual intervention (via the CLI or client code).

5.6.7 fennel.exceptions

exception fennel.exceptions.**FennelException**

exception fennel.exceptions.**TaskFailed**(*original_type: str*, *original_args: List*)
This exception is returned by worker processes which experienced an exception when executing a task.

Parameters

- **original_type** (*str*) – The name of the original exception, e.g. `'ValueError'`.
- **original_args** (*List*) – The arguments given to the original exception, e.g. `['Not found']`

Examples

```
>>> @app.task(retries=0)
>>> async def foo(n):
...     raise Exception("baz")
...
>>> x = await foo.delay(3)
>>> try:
...     result = await x.get()
>>> except TaskFailed as e:
...     assert e.original_type == "Exception"
...     assert e.original_args == ["baz"]
```

exception fennel.exceptions.**ResultsDisabled**
Raised when `results_enabled=False` and code attempts to access a tasks result via `.get()`.

exception fennel.exceptions.**UnknownTask**
Raised by a worker process if it is unable to find a Python function corresponding to the task it has read from the queue.

exception fennel.exceptions.**Timeout**
Raised by client code when a given timeout is exceeded when waiting for results to arrive.

exception fennel.exceptions.**JobNotFound**
Raised by client code when attempting to retrieve job information that cannot be found in Redis.

exception fennel.exceptions.**Chaos**
Used in tests to ensure failures are handled properly.

exception fennel.exceptions.**Completed**
Used internally to shutdown an Executor if the exit condition is completing all tasks.

5.6.8 fennel.utils

fennel.utils.**backoff**(*retries: int*, *jitter: bool = True*) → int
Compute duration (seconds) to wait before retrying using exponential backoff with jitter based on the number of retries a message has already experienced.

The minimum returned value is 1s The maximum returned value is 604800s (7 days)

With max_retries=9, you will have roughly 30 days to fix and redeploy the the task code.

Parameters

- **retries** (*int*) – How many retries have already been attemped.
- **jitter** (*bool*) – Whether to add random noise to the return value (recommended).

Notes

https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

5.6.9 fennel.job

class fennel.job.**Job**(*task: str*, *args: List*, *kwargs: Dict*, *tries: int = 0*, *max_retries: int = 9*, *exception: Dict = <factory>*, *return_value: Any = None*, *status: str = 'UNKNOWN'*, *uuid: str = <factory>*)
The internal representation of a job.

Parameters

- **task** (*str*) – The name of the task. By default will use `f"{func.__module__}.{func.__qualname__}"`, where *func* is the Python callable.
- **args** (*List*) – The job's args.
- **kwargs** (*Dict*) – The job's kwargs.
- **tries** (*int*) – The number of attempted executions.
- **max_retries** (*int*) – The maximum number of retries to attempt after failure.
- **exception** (*Dict*) – Exception information for the latest failure, contains 'original_type' (str, e.g. 'ValueError') and 'original_args' (List, e.g. ['Not found']).
- **return_value** (*Any*) – The return value of the Python callable when execution succeeds.

- **`status`** (*`str`*) – One of *`fennel.status`*, the current lifecycle stage.
- **`uuid`** (*`str`*) – Base64-encoded unique identifier.

5.7 Changelog

5.7.1 v0.3.0 (2020-08-20)

- Added configurable grace period before shutting down
- Exceptions now have a common superclass
- Switched async Redis driver from aioredis to aredis
- Adopted AnyIO for better async primitives

5.7.2 v0.2.4 (2020-07-03)

- Fixed multiprocessing bug for thread listener

5.7.3 v0.2.3 (2020-07-02)

- Bump pydantic major version

5.7.4 v0.2.2 (2020-07-02)

- Bump structlog major version

5.7.5 v0.2.1 (2020-07-02)

- Improved testing for CPU-bound tasks

5.7.6 v0.2.0 (2020-06-14)

- Added Python 3.8 support

5.7.7 v0.1.2 (2019-10-06)

- Fixed typo maintenence -> maintenance

5.7.8 v0.1.1 (2019-10-03)

- Fixed CLI and autodiscovery bugs

5.7.9 v0.1.0 (2019-10-03)

- Initial release

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX