aiotasks Documentation

Release 1.0.0

Daniel Garcia - cr0hn

May 27, 2020

Contents

1 What's aiotasks **3**

2 Contents **5**

2.1 Install aiotasks 5

2.2 Quick Start 5

2.3 Backends 8

2.4 Architecture 9

2.5 Contributing 9

2.6 API 9

3 Indices and tables **11**

Project site	http://github.com/cr0hn/aiotasks
Author	Daniel Garcia (cr0hn) - @ggdaniel
Documentation	http://aiotasks.readthedocs.org
Last Version	1.0.0
Python versions	3.5 or above

aiotasks: A Celery like task manager that distributes Asyncio coroutines

CHAPTER 1

What's aiotasks

aiotasks is an asynchronous & distributed task queue / jobs queue, implemented as coroutines and based on Python asyncio framework.

Based on the Celery Task Queue ideas, but distributing coroutines and doing focus in performance, non-blocking & event-driven concepts.

aiotasks doesn't pulling and doesn't has active waiting for incoming jobs, instead use asyncio framework to suspend the execution until any new data are received by a new broker notification.

Note: aiotasks is still under development. Not as active as I would like (for time limitations), but the project is in active development.

If you want to contribute, take a look to the TODO.md file.

CHAPTER 2

Contents

2.1 Install aiotasks

Installing aiotasks is very simple, depending on the release you want, you can install:

- stable version (**recomended**).
- development version.

2.1.1 Stable version

```
> python3.5 -m pip install aiotasks
```

2.1.2 Development version

```
> python3.5 -m pip install https://github
.com/cr0hn/aiotasks/archive/develop.zip
```

2.2 Quick Start

This is a quick start with aiotasks with the minimum information to start.

Note: You can find examples at project `examples` folder.

2.2.1 Running modes

You can run aiotasks as two ways:

- Launching an aiotasks manager in an independent console / process (like Celery does), and then sending any tasks to aiotasks through the broker.
- Running the standalone way: Launching the client and the server in a unique point and running both at the same time.

2.2.2 Running using the manager

Run the manager

```
> aiotasks -vvvv worker -A examples.launch_manager_tasks_and_launch_in_console
```

Send the tasks

```
> python examples/launch_manager_tasks_and_launch_in_console.py
```

2.2.3 Running standalone

```
> python examples/standalone_tasks_standalone.py
```

2.2.4 Defining tasks

This concept was ported from Celery. Defining any tasks is very simple, only need to decorate a function with *task* function.

```
from aiotasks import build_manager

manager = build_manager("redis://")

@manager.task()  # <-- DEFINITION OF TASK
async def task_01(num):  # <-- TASK SHOULD BE A **COROUTINE**
    print("Task 01 starting: {}".format(num))
    await asyncio.sleep(2, loop=manager.loop)
    print("Task 01 stopping")
```

2.2.5 Sending info to tasks

We can send tasks to the manager using methods:

Using *delay* method

```
from aiotasks import build_manager

manager = build_manager("redis://")

@manager.task()
async def task_01(num):
    await asyncio.sleep(0, loop=manager.loop)

async def generate_tasks():
    # Generates 5 tasks
    for x in range(5):
        await task_01.delay(x)  # <-- METHOD DELAY SEND A TASK

if __name__ == '__main__':
    manager.loop.run_until_complete(generate_tasks())
```

Using *send_task*

```
from aiotasks import build_manager, send_task

manager = build_manager("redis://")

@manager.task()
async def task_01(num):
    await asyncio.sleep(0, loop=manager.loop)

async def generate_tasks():
    # Generates 5 tasks
    for x in range(5):
        await send_task("task_01", args=(x, ))  # <-- SENDING A TASK

if __name__ == '__main__':
    manager.loop.run_until_complete(generate_tasks())
```

2.2.6 Sending info to tasks & wait for response

We can also send for a task job and wait for the response in a **non-blocking mode**:

```
from aiotasks import build_manager

manager = build_manager("redis://")

@manager.task()
async def task_01(num):
    await asyncio.sleep(0, loop=manager.loop)

async def generate_tasks():
    # Generates 5 tasks
    async with task_01.wait(x) as f:  # <-- NON-BLOCKING WAITING FOR RESPONSE
        print(f)

if __name__ == '__main__':
    manager.loop.run_until_complete(generate_tasks())
```

2.3 Backends

aiotasks supports various backends. All of them are event-driven queue systems.

Currently supported backends are:

- Redis
- Memory

Planned supported in near future:

- RabbitMQ (AMQP)
- ZeroMQ

Panned supported in more long future:

- etcd
- consul

2.3.1 How to specify a backend

The backends should be specified as a URI format. This is:

- BACKEND_TYPE://USER:PASSWORD@IP_OR_HOST:PORT/ANY_OTHER_INFO

2.3.2 Redis

To configure the Redis backend we must to specify the `BACKEND_TYPE` as **redis**, following this format: `redis://HOST:PORT/DB`

Examples

Connect to localhost and default Redis options:

```
from aiotasks import build_manager

manager = build_manager("redis://")

...
```

Custom Redis server, with the information:

- Redis password: **mypassword**.
- Custom host: **10.0.0.1**:
- Redis database: **12**.

```
from aiotasks import build_manager

manager = build_manager("redis://:mypassword@10.0.0.1/12")

...
```

2.3.3 Memory

Memory is a special backend type. It should be used for small workload systems or development environments.

This execution mode is useful to do small and local tasks. For example: If you're using aiohttp and you only want to send an email in a background way, you can use the standalone way and the memory backend.

The `BACKEND_TYPE` type is **memory** and the format is: `:memory://`.

Example

```
from aiotasks import build_manager

manager = build_manager("memory://")

...
```

2.4 Architecture

TODO

2.5 Contributing

Contributors are welcome. You can find a list ot TODO tasks in the TODO.md at the project file.

All contributors will be added to the CONTRIBUTORS.md file.

Thanks in advance if you're planning to contribute to the project! :)

2.6 API

TODO

CHAPTER 3

Indices and tables

- genindex
- modindex
- search