async-service Documentation

Release 0.1.0-alpha.11

The Ethereum Foundation

Sep 28, 2020

General

CHAPTER 1

async-service

Lifecycle management for async applications

1.1 Goals

This library provides strong lifecycle management for asynchronous applications.

Features:

- Provide a well defined service lifecycle
- Allow orchestration of services via:
  - Tasks
  - Daemon Tasks
  - Child Services
- Well-behaved cancellation of services

1.2 Further reading

Here are a couple more useful links to check out.

- *Guides*
- Source Code on GitHub

1.2.1 Introduction

async-service

Lifecycle management for async applications

Goals

This library provides strong lifecycle management for asynchronous applications.

Features:

- Provide a well defined service lifecycle
- Allow orchestration of services via:
  - Tasks
  - Daemon Tasks
  - Child Services
- Well-behaved cancellation of services

Further reading

Here are a couple more useful links to check out.

- *Guides*
- Source Code on GitHub

1.2.2 Release Notes

Async_Service 0.1.0-alpha.10 (2020-09-24)

Features

- Turn off verbose logging about task lifecycle by default. To re-enable it, set the environment variable `ASYNC_SERVICE_VERBOSE_LOG=1`. (#75)
- In py3.8, annotate asyncio tasks with a name, so that asyncio logs show more than _run_and_manage_task() when there's an issue like a coro that takes too long. (#76)
- Raise an exception when more than 1000 child tasks are concurrently running. It slows down the event loop too much. (#77)

Internal Changes - for async-service Contributors

- Pull in updates from project template, for latest release notes, Makefile, etc. (#78)

v0.1.0-alpha.1

- Launched repository, claimed names for pip, RTD, github, etc

1.2.3 Guides

Services

This library provides strong lifecycle management for asynchronous applications.

All application logic must be encapslated in a `Service` class which implements a `run()` method.

```
from async_service import Service, run_asyncio_service

class MyApplication(Service):
    def run(self):
        print("I'm a service")

await run_asyncio_service(MyApplication())
```

You can also run services in the background while we do other things.

```
from async_service import Service, background_asyncio_service

class MyApplication(Service):
    def run(self):
        print("I'm a service")

async with background_asyncio_service(MyApplication()):
    # do things while it runs
    ...
# service will be finished here.
```

Lifecycle of a Service

Each service has a well defined lifecycle.

```
+---------+
| STARTED |
+---------+
     |
     v
+---------+
| RUNNING |
+---------+
     |
     v
+----------+
| FINISHED |
+----------+
```

- **Started:**
  - The `run()` method has been scheduled to run.
- **Running:**
  - The service has started and is still running (has not been cancelled and has not finished)
- **Finished**
  - The service has stopped. All background tasks have either completed or been cancelled.

Cancellation

Calling `cancel()` will trigger cancellation of the service and all child tasks and child services. A service that has been cancelled will still register as "running" until all child tasks have been cancelled and the service registers as "finished".

Managers

The `ManagerAPI` is responsible for running a service and managing the service lifecycle. It also exposes all of the APIs for inspecting a running service or waiting for the service to reach a specific state.

```
from async_service import background_asyncio_service

from my_application import MyApplicationService

async with background_asyncio_service(MyApplicationService()) as manager:
    # wait for the service to be started
    await manager.wait_started()

    # check if the service has started
    if manager.is_started:
        ...

    # check if the service is running
    if manager.is_running:
        ...

    # check if the service has been cancelled
    if manager.is_cancelled:
        ...

    # check if the service is finished
    if manager.is_finished:
        ...

    # wait for the service to finishe completely
    await manager.wait_finished()
```

The `ManagerAPI` also allows us to control the service.

```
from async_service import background_asyncio_service

from my_application import MyApplicationService

async with background_asyncio_service(MyApplicationService()) as manager:
    # Cancel the service
    manager.cancel()

    # Cancel the service AND wait for it to be finished
    await manager.stop()
```

Tasks

Asynchrounous applications will typically need to run multiple things concurrently which implies running things in the *background*.

This is done using the `manager` attribute which exposes the `run_task()` method.

```
from async_service import Service, run_asyncio_service

async def fetch_url(url):
    ...

class MyService(Service):

    async def run(self):
        for url in URLS_TO_FETCH:
            self.manager.run_task(fetch_url, url)
```

The example above shows a service that concurrently fetches multiple URLS concurrently. These *tasks* will be scheduled and run in the background. The service will run until all of the background tasks are finished or the service encounters an error in one of the tasks.

If a task raises an exception it will trigger cancellation of the service. Upon exiting, all errors that were encountered while running the service will be re-raised.

For slighly nicer logging output we can provide a `name` as a keyword argument to *~async_service.abc._InternalManagerAPI.run_task* which will be used in logging messages.

Daemon Tasks

A *"Daemon"* tasks is one that is intended to run for the full lifecycle of the service. This can be done by passing `daemon=True` into the call to `run_task()`.

```
from async_service import Service, run_asyncio_service

class MyService(Service):
    async def do_long_running_thing(self):
        while True:
            ...

    async def run(self):
        # The following two statements are equivalent.
        self.manager.run_task(self.do_long_running_thing, daemon=True)
        self.manager.run_daemon_task(self.do_long_running_thing)
```

Alternatively we can use `run_daemon_task()`.

A *"Daemon"* task which finishes before the service is shuts down will trigger cancellation and result in the `DaemonTaskExit` exception to be raised.

Child Services

Child services are like tasks, except that they are other services that we want to run within a running service.

```
from async_service import Service, run_asyncio_service

class ChildService(Service):
    async def run(self):
        ...

class ParentService(Service):
```

(continues on next page)

(continued from previous page)

```
    async def run(self):
        child_manager = self.manager.run_child_service(ChildService())
```

Child services are run using the `run_child_service()` method which returns the manager for the child service.

There is also a `run_daemon_child_service()` method behaves the same as `run_daemon_task()` in that if the child service finishes before the parent service has finished, it will raise a `DaemonTaskExit` exception.

Task Shutdown

Note: This behavior is currently only guaranteed when using the `asyncio` based service manager.

As a service spawns background tasks, the manager keeps track of them as a DAG. The **root** of the DAG is always the `run()` method with each new background task being a child of whatever parent coroutine spawned it.

When the service is cancelled, these tasks are cancelled by traversing the task DAG starting at the leaves and working up towards the root. This provides a guarantee that if the `run()` method spawns multiple backound tasks, that the background tasks will be cancelled before the `run()` method is cancelled.

External Service APIs

Sometimes we may want to expose an API from a `Service` for external callers such that the call should only work if the service is running, and calls should fail or be terminated if the service is cancelled or finishes.

This can be done with the `external_asyncio_api()` and `external_trio_api()` decorators.

```
from async_service import Service, background_asyncio_service, external_asyncio_api

class MyService(Service):
    async def run(self):
        ...

    @external_asyncio_api
    async def get_thing(self):
        ...

service = MyService()

# this will fail because the service isn't running yet
await service.get_thing()

async with background_asyncio_service(service) as manager:
    thing = await service.get_thing()

    # now cancel the service
    manager.cancel()

    # this will fail because the service is cancelled.
    thing = await service.get_thing()
```

Note: The `external_asyncio_api()` can only be used on coroutine functions.

When a method decorated with `external_asyncio_api()` fails it raises an `async_service.exceptions.LifecycleError` exception.

Cleanup logic

In the case that we need to run some logic **after** the service has finished running but **before** the service has registered as finished we can do so with the following patterns. However, special care and consideration should be taken as the following patterns can result in the application hanging when we try to shut it down.

The basic idea is to use a `try/finally` expression in our main `Service.run()` method. Since services track and shutdown their tasks using a DAG, the code in the `finally` block is guaranteed to run after everything else has stopped.

```
from async_service import Service

class CleanupService(Service):
    async def run(self) -> None:
        try:
            ...  # do main service logic here
        finally:
            ...  # do cleanup logic here
```

For those running under `trio` it is worth noting that if the cleanup logic needs to `await` anything we will probably need to shield it from further cancellations.

```
from async_service import Service

class CleanupService(Service):
    async def run(self) -> None:
        try:
            ...  # do main service logic here
        finally:
            with trio.CancelScope(shield=True):
                ...  # do cleanup logic here
```

It is relatively trivial to implement a reusable pattern for doing cleanup.

```
from async_service import Service

class CleanupService(Service):
    async def run(self) -> None:
        try:
            ...  # do main service logic here
        except:
            await self.on_error()
            raise
        else:
            await self.on_success()
        finally:
            await self.on_finally()

    async def on_success(self) -> None:
        pass

    async def on_error(self) -> None:
        pass
```

(continues on next page)

(continued from previous page)

```
    async def on_finally(self) -> None:
        pass
```

Stats

The `ManagerAPI` exposes a `stats()` method which returns a `Stats` object with basic stats about the running service.

```
async with background_asyncio_service(MyService) as manager:
    stats = manager.stats

    print(f"Total running tasks: {stats.total_count}")
    print(f"Finished tasks: {stats.finished_count}")
    print(f"Pending tasks: {stats.pending_count}")
```

1.2.4 API

ABC

ManagerAPI

InternalManagerAPI

ServiceAPI

Base

BaseManager

Service

Asyncio

AsyncioManager

background_asyncio_service

external_api

Trio

TrioManager

background_trio_service

external_api

Exceptions

DaemonTaskExit

LifecycleError

ServiceException

1.2.5 Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at piper@pipermerriam.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html