Pyncette

Release 0.6.2

Jun 28, 2020

Contents

CHAPTER 1

Overview

docs	
tests	
package	

A reliable distributed scheduler with pluggable storage backends

- Free software: MIT license

1.1 Installation

Minimal installation (just SQLite persistence):

```
pip install pyncette
```

Full installation (Redis and PostgreSQL persistence and Prometheus metrics exporter):

```
pip install pyncette[redis,postgres,prometheus]
```

You can also install the in-development version with:

```
pip install https://github.com/tibordp/pyncette/archive/master.zip
```

1.2 Documentation

https://pyncette.readthedocs.io

1.3 Usage example

Simple in-memory scheduler (does not persist state)

```
from pyncette import Pyncette, Context

app = Pyncette()

@app.task(schedule='* * * * *')
async def foo(context: Context):
    print('This will run every minute')

if __name__ == '__main__':
    app.main()
```

Persistent distributed cron using Redis (coordinates execution with parallel instances and survives restarts)

```
from pyncette import Pyncette, Context
from pyncette.redis import redis_repository

app = Pyncette(repository_factory=redis_repository, redis_url='redis://localhost')

@app.task(schedule='* * * * * */10')
async def foo(context: Context):
    print('This will run every 10 seconds')

if __name__ == '__main__':
    app.main()
```

See the *examples* directory for more examples of usage.

1.4 Development

To run integration tests you will need Redis and PostgreSQL Server running locally.

To run the all tests run:

```
tox
```

To run just the unit tests (excluding integration tests):

```
tox -e py37  # or py38
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER 2

Installation

At the command line:

```
pip install pyncette
```

For installing with Redis peristence:

```
pip install pyncette[redis]
```

For installing with PostgreSQL peristence:

```
pip install pyncette[postgres]
```

For installing with Prometheus metrics exporter:

```
pip install pyncette[prometheus]
```

CHAPTER 3

Usage

The core unit of execution in Pyncette is a `Task`. Each task is a Python coroutine that specifies what needs to be executed.

```
from pyncette import Pyncette, Context

app = Pyncette()

@app.task(interval=datetime.timedelta(seconds=2))
async def successful_task(context: Context) -> None:
    print("This will execute every second")

if __name__ == "__main__":
    app.main()
```

3.1 Running the main loop

The usual use case is that Pyncette runs as its own process, so the standard way to start the main loop is with *main()* method of the *Pyncette*. This sets up the logging to standard output and signal handler allowing for graceful shutdown (first SIGINT initiates the graceful shutdown and the second one terminates the process).

If Pyncette is run alongside other code or for customization, *create()* can be used to initialize the runtime environment and then the main loop can be run with *run()*:

```
import asyncio
from pyncette import Pyncette

app = Pyncette()

...

async with app.create() as app_context:
    await app_context.run()
```

3.2 Specifying the schedule

There are two ways a schedule can be specified, one is with the cron-like syntax (uses `croniter` under the hood to support the calculation):

```
@app.task(schedule="* * * * *")
async def every_minute(context: Context):
    ...

@app.task(schedule="* * * * * */10")
async def every_10_seconds(context: Context):
    ...

@app.task(schedule="20 4 * * * *")
async def every_day_at_4_20_am(context: Context):
    ...
```

The other way is with an interval:

```
@app.task(interval=datetime.timedelta(seconds=12))
async def every_12_seconds(context: Context):
    ...
```

3.3 Customizing tasks

Pyncette supports multiple different execution modes which provide different levels of reliability guarantees, depending on the nature of the task.

The default task configuration:

- When the task is scheduled for execution, it is locked for 60 seconds
- If the task execution succeeds, the next execution is scheduled and the task is unlocked
- If the task execution fails (exception is raised), the lock is not released, so it will be retried after the lease expires.
- If the task execution exceeds the lease duration, it will be executed again (so there could be two executions at the same time)

3.3.1 Best-effort tasks

If the task is run in a best-effort mode, locking will not be employed, and the next execution will be scheduled immediately when it becomes ready.:

```
from pyncette import ExecutionMode

@app.task(interval=datetime.timedelta(seconds=10), execution_mode=ExecutionMode.AT_
↪MOST_ONCE)
async def every_10_seconds(context: Context):
    print("Ping")
```

Caution: If best effort is used, there is no way to retry a failed execution, and exceptions thrown by the task will only be logged.

3.3.2 Failure behavior

Failure behavior can be specified with `failure_mode` parameter:

```
from pyncette import ExecutionMode

@app.task(interval=datetime.timedelta(seconds=10), failure_mode=FailureMode.UNLOCK)
async def every_10_seconds(context: Context):
    print("Ping")
```

- `FailureMode.NONE` the task will stay locked until the lease expires. This is the default.
- `FailureMode.UNLOCK` the task will be immediately unlocked if an exception is thrown, so it will be retried on the next tick.
- `FailureMode.COMMIT` treat the exception as a success and schedule the next execution in case the exception is thrown.

3.3.3 Timezone support

Pyncette is timezone-aware, the timezone for a task can be specified by `timezone` parameter:

```
from pyncette import ExecutionMode

@app.task(schedule="0 12 * * *", timezone="Europe/Dublin")
async def task1(context: Context):
    print(f"Hello from Dublin!")

@app.task(schedule="0 12 * * *", timezone="UTC+12")
async def task2(context: Context):
    print(f"Hello from !")
```

The accepted values are all that `dateutil.tz.gettz()` accepts.

3.3.4 Task parameters

The *`task()`* decorator accepts an arbitrary number of additional parameters, which are available through the `context` parameter

```
from pyncette import ExecutionMode

# If we use multiple decorators on the same coroutine, we must explicitely provide␣
↪the name
@app.task(name="task1", interval=datetime.timedelta(seconds=10), username="abra")
@app.task(name="task2", interval=datetime.timedelta(seconds=20), username="kadabra")
@app.task(name="task3", interval=datetime.timedelta(seconds=30), username="alakazam")
async def task(context: Context):
    print(f"{context.args['username']}")
```

This allows for parametrized tasks with multiple decorators, this is an essential feature needed to support *Dynamic tasks*.

Note: There is a restriction that all the values of the parameters must be JSON-serializable, since they are persisted in storage when dynamic tasks are used.

3.4 Middlewares

If you have common logic that should execute around every task invocation, middlewares can be used. Good examples of middlewares are ones used for logging and metrics.

```
app = Pyncette()

@app.middleware
async def retry(context: Context, next: Callable[[], Awaitable[None]]):
    # Example only, prefer to rely on Pyncette to drive task retry logic
    for _ in range(5):
        try:
            await next()
            return
        except Exception as e:
            pass
    raise Exception(f"Task {context.task.name} failed too many times.")

@app.middleware
async def logging(context: Context, next: Callable[[], Awaitable[None]]):
    logger.info(f"Task {context.task.name} started")
    try:
        await next()
    except Exception as e:
        logger.error(f"Task {context.task.name} failed", e)
        raise

@app.middleware
async def db_transaction(context: Context, next: Callable[[], Awaitable[None]]):
    context.db.begin_transaction()
    try:
        await next()
    except Exception:
        context.db.rollback()
        raise
    else:
        context.db.commit()
```

Middlewares execute in order they are defined.

3.5 Fixtures

Fixtures provide a convenient way for injecting dependencies into tasks, and specifying the set-up and tear-down code. They can be though of as application-level middlewares. For example, let's say we want to inject the database and a logfile as dependencies to all our tasks:

```
app = Pyncette()

@app.fixture()
async def db(app_context: PyncetteContext):
    db = await database.connect(...)
    try:
        yield db
    finally:
        await db.close()
```

(continues on next page)

(continued from previous page)

```
@app.fixture(name="super_log_file")
async def logfile(app_context: PyncetteContext):
    with open("log.txt", "a") as file:
        yield file

@app.task(interval=datetime.timedelta(seconds=2))
async def successful_task(context: Context) -> None:
    context.super_log_file.write("Querying the database")
    results = await context.db.query(...)
    ...
```

The lifetime of a fixture is that of a Pyncette application, i.e. the setup code for all fixtures runs before the first tick and the tear-down code runs after the graceful shutdown is initiated and all the pending tasks have finished. Like middlewares, fixtures execute in the order they are defined (and in reverse order on shutdown).

3.6 Persistence

By default Pyncette runs without persistence. This means that the schedule is mainteined in-memory and there is no coordination between multiple instances of the app.

Enabling persistence allows the aplication to recover from restarts as well as the ability to run multiple instances of an app concurrently without duplicate executions of tasks.

3.6.1 SQLite

SQLite is the default peristence engine.

```
from pyncette import Pyncette, Context

app = Pyncette(sqlite_database="pyncette.db")

@app.task(schedule='* * * * * */10')
async def foo(context: Context):
    print('This will run every 10 seconds')

if __name__ == '__main__':
    app.main()
```

3.6.2 Redis

Redis can be enabled by passing *redis_repository()* as repository_factory parameter to the *Pyncette* constructor.

```
from pyncette import Pyncette, Context
from pyncette.redis import redis_repository

app = Pyncette(repository_factory=redis_repository, redis_url='redis://localhost')
```

Optionally, the tasks can be namespaced if the Redis server is shared among different Pyncette apps:

```
app = Pyncette(repository_factory=redis_repository, redis_url='redis://localhost',␣
↪redis_namespace='my_super_app')
```

3.6.3 PostgreSQL

Redis can be enabled by passing *postgres_repository()* as repository_factory parameter to the *Pyncette* constructor.

```
from pyncette import Pyncette, Context
from pyncette.postgres import postgres_repository

app = Pyncette(
    repository_factory=postgres_repository,
    postgres_url='postgres://postgres@localhost/pyncette'
    postgres_table_name='pyncette_tasks'
)
```

The table will be automatically initialized on startup if it does not exists.

3.7 Dynamic tasks

Pyncette supports a use case where the tasks are not necessarily known in advance with *schedule_task()*.

```
@app.dynamic_task()
async def hello(context: Context) -> None:
    print(f"Hello {context.args['username']}")

async with app.create() as app_context:
    await asyncio.gather(
        app_context.schedule_task(hello, "bill_task", schedule="0 * * * *", username=
↪"bill"),
        app_context.schedule_task(hello, "steve_task", schedule="20 * * * *",␣
↪username="steve"),
        app_context.schedule_task(hello, "john_task", schedule="40 * * * *", username=
↪"john"),
    )
    await app_context.run()
```

When persistence is used, the schedules and task parameters of the are persisted alongside the execution data, which allows the tasks to be registered and unregistered at will.

An example use case is a web application where every user can have something happen at their chosen schedule. Polling is relatively efficient, since the concrete instances of the dynamic class are only loaded from the storage if the are already due, instead of being polled all the time.

The task instances can be removed by *unschedule_task()*

```
...

async with app.create() as app_context:
    await app_context.schedule_task(hello, "bill_task", schedule="0 * * * *",␣
↪username="bill")
    await app_context.unschedule_task(hello, "bill_task")
    await app_context.run()
```

Note: If the number of dynamic tasks is large, it is a good idea to limit the batch size:

```
app = Pyncette(
    repository_factory=redis_repository,
    redis_url='redis://localhost',
    batch_size=10
)
```

This will cause that only a specified number of dynamic tasks are scheduled for execution during a single tick, as well as allow potential multiple instances of the same app to load balance effectively.

3.8 Once-off dynamic tasks

Dynamic tasks can also be scheduled to execute only once at a specific date.

```
@app.dynamic_task()
async def task(context: Context) -> None:
    print(f"Hello {context.task.name}!")

async with app.create() as app_context:
    await app_context.schedule_task(task, "y2k38", execute_at=datetime(2038, 1, 19, 3,
↪ 14, 7));
    await app_context.schedule_task(task, "tomorrow", execute_at=datetime.now() +␣
↪timedelta(days=1));

    # This will execute once immediately, since it is already overdue
    await app_context.schedule_task(task, "overdue", execute_at=datetime.now() -␣
↪timedelta(days=1));
    await app_context.run()
```

Once-off tasks have the same reliability guarantees as recurrent tasks, which is controlled by *execution_mode* and *failure_mode* parameters, but in case of success, they will not be scheduled again.

3.9 Performance

Tasks are executed in parallel. If you have a lot of long running tasks, you can set `concurrency_limit` in *`Pyncette`* constructor, as this ensures that there are at most that many executing tasks at any given time. If there are no free slots in the semaphore, this will serve as a back-pressure and ensure that we don't poll additional tasks until some of the currently executing ones finish, enabling the pending tasks to be scheduled on other instances of your app. Setting `concurrency_limit` to 1 is equivalent of serializing the execution of all the tasks.

CHAPTER 4

pyncette package

4.1 Submodules

4.1.1 pyncette.errors module

exception pyncette.errors.**PyncetteException**
Bases: Exception

Base exception for Pyncette

4.1.2 pyncette.model module

class pyncette.model.**Context**
Bases: object

Task execution context. This class can have dynamic attributes.

class pyncette.model.**ExecutionMode**
Bases: enum.Enum

The execution mode for a Pyncette task.

AT_LEAST_ONCE = 0

AT_MOST_ONCE = 1

class pyncette.model.**FailureMode**
Bases: enum.Enum

What should happen when a task fails.

COMMIT = 2

NONE = 0

UNLOCK = 1

class pyncette.model.**FixtureFunc**(**args*, ***kwargs*)
Bases: typing_extensions.Protocol

class pyncette.model.**MiddlewareFunc**(**args*, ***kwargs*)
Bases: typing_extensions.Protocol

class pyncette.model.**PollResponse**(*result: ResultType, scheduled_at: datetime.datetime, lease: Optional[Lease]*)
Bases: object

The result of a task poll

class pyncette.model.**QueryResponse**(*tasks: List[Tuple['pyncette.task.Task', Lease]], has_more: bool*)
Bases: object

The result of a task query

class pyncette.model.**ResultType**
Bases: enum.Enum

Status returned by polling the task

LEASE_MISMATCH = 4

LOCKED = 3

MISSING = 0

PENDING = 1

READY = 2

class pyncette.model.**TaskFunc**(**args*, ***kwargs*)
Bases: typing_extensions.Protocol

4.1.3 pyncette.postgres module

class pyncette.postgres.**PostgresRepository**(*pool: asyncpg.pool.Pool*, ***kwargs*)
Bases: *pyncette.repository.Repository*

commit_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None
Commits the task, which signals a successful run.

initialize() → None

poll_dynamic_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → pyncette.model.QueryResponse
Queries the dynamic tasks for execution

poll_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: Optional[NewType.<locals>.new_type] = None*) → pyncette.model.PollResponse
Polls the task to determine whether it is ready for execution

register_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None
Registers a dynamic task

unlock_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None
Unlocks the task, making it eligible for retries in case execution failed.

unregister_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None
Deregisters a dynamic task implementation

pyncette.postgres.**postgres_repository**(***kwargs*) → AsyncIterator[pyncette.postgres.PostgresRepository]

Factory context manager for Redis repository that initializes the connection to Postgres

4.1.4 pyncette.prometheus module

class pyncette.prometheus.**MeteredRepository**(*metric_set: pyncette.prometheus.OperationMetricSet*, *inner_repository: pyncette.repository.Repository*)

Bases: *pyncette.repository.Repository*

A wrapper for repository that exposes metrics to Prometheus

commit_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None

Commits the task, which signals a successful run.

poll_dynamic_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → pyncette.model.QueryResponse

Queries the dynamic tasks for execution

poll_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: Optional[NewType.<locals>.new_type] = None*) → pyncette.model.PollResponse

Polls the task to determine whether it is ready for execution

register_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None

Registers a dynamic task

unlock_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None

Unlocks the task, making it eligible for retries in case execution failed.

unregister_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None

Deregisters a dynamic task implementation

class pyncette.prometheus.**OperationMetricSet**(*operation_name: str, labels: List[str]*)

Bases: object

Collection of Prometheus metrics representing a logical operation

measure(***labels*) → AsyncIterator[None]

An async context manager that measures the execution of the wrapped code

pyncette.prometheus.**prometheus_fixture**(*app_context: pyncette.pyncette.PyncetteContext*) → AsyncIterator[None]

pyncette.prometheus.**prometheus_middleware**(*context: pyncette.model.Context*, *next: Callable[[], Awaitable[None]]*) → None

Middleware that exposes task execution metrics to Prometheus

pyncette.prometheus.**use_prometheus**(*app: pyncette.pyncette.Pyncette*, *measure_repository: bool = True*, *measure_ticks: bool = True*, *measure_tasks: bool = True*) → None

Decorate Pyncette app with Prometheus metric exporter.

Parameters

- **measure_repository** – Whether to measure repository operations
- **measure_ticks** – Whether to measure ticks
- **measure_tasks** – Whether to measure individual task executions

pyncette.prometheus.**with_prometheus_repository**(*repository_factory: pyncette.repository.RepositoryFactory*) → pyncette.repository.RepositoryFactory

Wraps the repository factory into one that exposes the metrics via Prometheus

4.1.5 pyncette.pyncette module

class pyncette.pyncette.**Pyncette**(*repository_factory: pyncette.repository.RepositoryFactory = <function sqlite_repository>*, *executor_cls: Type[CT_co] = <class 'pyncette.executor.DefaultExecutor'>*, *poll_interval: datetime.timedelta = datetime.timedelta(seconds=1)*, ***kwargs*)

Bases: object

Pyncette application.

create() → AsyncIterator[PyncetteContext]
Creates the execution context.

dynamic_task(***kwargs*) → Callable[[pyncette.model.TaskFunc], pyncette.model.TaskFunc]
Decorator for marking the coroutine as a dynamic task

fixture(*name: Optional[str] = None*) → Callable[[pyncette.model.FixtureFunc], pyncette.model.FixtureFunc]
Decorator for marking the generator as a fixture

main() → None
Convenience entrypoint for console apps, which sets up logging and signal handling.

middleware(*func: pyncette.model.MiddlewareFunc*) → pyncette.model.MiddlewareFunc
Decorator for marking the function as a middleware

task(***kwargs*) → Callable[[pyncette.model.TaskFunc], pyncette.model.TaskFunc]
Decorator for marking the coroutine as a task

use_fixture(*fixture_name: str*, *func: pyncette.model.FixtureFunc*) → None

use_middleware(*func: pyncette.model.MiddlewareFunc*) → None

class pyncette.pyncette.**PyncetteContext**(*app: pyncette.pyncette.Pyncette*, *repository: pyncette.repository.Repository*, *executor: pyncette.executor.DefaultExecutor*)

Bases: object

Execution context of a Pyncette app

initialize(*root_context: pyncette.model.Context*) → None

last_tick

run() → None
Runs the Pyncette's main event loop.

schedule_task(*task: pyncette.task.Task*, *instance_name: str*, ***kwargs*) → pyncette.task.Task
Schedules a concrete instance of a dynamic task

shutdown() → None
Initiates graceful shutdown, terminating the main loop, but allowing all executing tasks to finish.

unschedule_task(*task: pyncette.task.Task*, *instance_name: Optional[str] = None*) → None
Removes the concrete instance of a dynamic task

4.1.6 pyncette.redis module

class pyncette.redis.**RedisRepository**(*redis_client: aioredis.commands.Redis*, ***kwargs*)
Bases: *pyncette.repository.Repository*

Redis-backed store for Pyncete task execution data

commit_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None
Commits the task, which signals a successful run.

poll_dynamic_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → pyncette.model.QueryResponse
Queries the dynamic tasks for execution

poll_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: Optional[NewType.<locals>.new_type] = None*) → pyncette.model.PollResponse
Polls the task to determine whether it is ready for execution

register_scripts() → None
Registers the Lua scripts used by the implementation ahead of time

register_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None
Registers a dynamic task

unlock_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None
Unlocks the task, making it eligible for retries in case execution failed.

unregister_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None
Deregisters a dynamic task implementation

pyncette.redis.**redis_repository**(***kwargs*) → AsyncIterator[RedisRepository]
Factory context manager for Redis repository that initializes the connection to Redis

4.1.7 pyncette.repository module

class pyncette.repository.**Repository**
Bases: abc.ABC

Abstract base class representing a store for Pyncette tasks

commit_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None
Commits the task, which signals a successful run.

poll_dynamic_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → pyncette.model.QueryResponse
Queries the dynamic tasks for execution

poll_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: Optional[NewType.<locals>.new_type] = None*) → pyncette.model.PollResponse
Polls the task to determine whether it is ready for execution

register_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None
Registers a dynamic task

unlock_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None
Unlocks the task, making it eligible for retries in case execution failed.

unregister_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None
Deregisters a dynamic task implementation

class pyncette.repository.**RepositoryFactory**(**args*, ***kwargs*)
Bases: typing_extensions.Protocol

A factory context manager for creating a repository

4.1.8 pyncette.healthcheck module

pyncette.healthcheck.**default_healthcheck**(*app_context: pyncette.pyncette.PyncetteContext*) → bool

pyncette.healthcheck.**use_healthcheck_server**(*app: pyncette.pyncette.Pyncette*, *port: int = 8080*, *bind_address: str = None*, *healthcheck_handler: Callable[[pyncette.pyncette.PyncetteContext], Awaitable[bool]] = <function default_healthcheck>*) → None
Decorate Pyncette app with a healthcheck endpoint served as a HTTP endpoint.

Parameters

- **app** – Pyncette app
- **port** – The local port to bind to
- **bind_address** – The local address to bind to

Healthcheck_handler A coroutine that determines health status

4.1.9 pyncette.sqlite module

class pyncette.sqlite.**SqliteRepository**(*connection: aiosqlite.core.Connection*, ***kwargs*)
Bases: *pyncette.repository.Repository*

commit_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None
Commits the task, which signals a successful run.

initialize() → None

poll_dynamic_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → pyncette.model.QueryResponse
Queries the dynamic tasks for execution

poll_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: Optional[NewType.<locals>.new_type] = None*) → pyncette.model.PollResponse
Polls the task to determine whether it is ready for execution

register_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None
Registers a dynamic task

unlock_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → None
Unlocks the task, making it eligible for retries in case execution failed.

unregister_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*) → None
Deregisters a dynamic task implementation

pyncette.sqlite.**sqlite_repository**(***kwargs*) → AsyncIterator[pyncette.sqlite.SqliteRepository]

Factory context manager for Sqlite repository that initializes the connection to Sqlite

4.1.10 pyncette.task module

class pyncette.task.**Task**(*name: str*, *func: pyncette.model.TaskFunc*, *dynamic: bool = False*, *parent_task: Optional[Task] = None*, *schedule: Optional[str] = None*, *interval: Optional[datetime.timedelta] = None*, *execute_at: Optional[datetime.datetime] = None*, *timezone: Optional[str] = None*, *fast_forward: bool = False*, *failure_mode: pyncette.model.FailureMode = <FailureMode.NONE: 0>*, *execution_mode: pyncette.model.ExecutionMode = <ExecutionMode.AT_LEAST_ONCE: 0>*, *lease_duration: datetime.timedelta = datetime.timedelta(seconds=60)*, ***kwargs*)

Bases: object

The base unit of execution

as_spec() → Dict[str, Any]
Serializes all the attributes to task spec

canonical_name
A unique identifier for a task instance

get_next_execution(*utc_now: datetime.datetime, last_execution: Optional[datetime.datetime]*) → Optional[datetime.datetime]

instantiate(*name: str*, ***kwargs*) → pyncette.task.Task
Creates a concrete instance of a dynamic task

instantiate_from_spec(*task_spec: Dict[str, Any]*) → pyncette.task.Task
Deserializes all the attributes from task spec

4.2 Module contents

class pyncette.**Pyncette**(*repository_factory: pyncette.repository.RepositoryFactory = <function sqlite_repository>*, *executor_cls: Type[CT_co] = <class 'pyncette.executor.DefaultExecutor'>*, *poll_interval: datetime.timedelta = datetime.timedelta(seconds=1)*, ***kwargs*)

Bases: object

Pyncette application.

create() → AsyncIterator[PyncetteContext]
Creates the execution context.

dynamic_task(***kwargs*) → Callable[[pyncette.model.TaskFunc], pyncette.model.TaskFunc]
Decorator for marking the coroutine as a dynamic task

fixture(*name: Optional[str] = None*) → Callable[[pyncette.model.FixtureFunc], pyncette.model.FixtureFunc]
Decorator for marking the generator as a fixture

main() → None
Convenience entrypoint for console apps, which sets up logging and signal handling.

middleware(*func: pyncette.model.MiddlewareFunc*) → pyncette.model.MiddlewareFunc
Decorator for marking the function as a middleware

task(***kwargs*) → Callable[[pyncette.model.TaskFunc], pyncette.model.TaskFunc]
Decorator for marking the coroutine as a task

use_fixture(*fixture_name: str*, *func: pyncette.model.FixtureFunc*) → None

use_middleware(*func: pyncette.model.MiddlewareFunc*) → None

class pyncette.**ExecutionMode**
Bases: enum.Enum

The execution mode for a Pyncette task.

AT_LEAST_ONCE = 0

AT_MOST_ONCE = 1

class pyncette.**FailureMode**
Bases: enum.Enum

What should happen when a task fails.

COMMIT = 2

NONE = 0

UNLOCK = 1

class pyncette.**Context**
Bases: object

Task execution context. This class can have dynamic attributes.

class pyncette.**PyncetteContext**(*app: pyncette.pyncette.Pyncette*, *repository: pyncette.repository.Repository*, *executor: pyncette.executor.DefaultExecutor*)
Bases: object

Execution context of a Pyncette app

initialize(*root_context: pyncette.model.Context*) → None

last_tick

run() → None
Runs the Pyncette's main event loop.

schedule_task(*task: pyncette.task.Task*, *instance_name: str*, ***kwargs*) → pyncette.task.Task
Schedules a concrete instance of a dynamic task

shutdown() → None
Initiates graceful shutdown, terminating the main loop, but allowing all executing tasks to finish.

unschedule_task(*task: pyncette.task.Task*, *instance_name: Optional[str] = None*) → None
Removes the concrete instance of a dynamic task

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

Pyncette could always use more documentation, whether as part of the official Pyncette docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/tibordp/pyncette/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *pyncette* for local development:

1. Fork pyncette (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:tibordp/pyncette.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. Running integration tests requires a Redis server running on localhost.
6. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

7. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (see [tox documentation](https://tox.readthedocs.io/en/latest/example/basic.html#parallel-mode)):

```
tox --parallel auto
```

[1] If you don't have all the necessary python versions available locally you can rely on Github Actions - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 6

Authors

- Tibor Djurica Potpara - https://www.ojdip.net

CHAPTER 7

Changelog

7.1 0.6.1 (2020-04-02)

- Optimize the task querying on Postgres backend
- Fix: ensure that there are no name colissions between concrete instances of different dynamic tasks
- Improve fairness of polling tasks under high contention.

7.2 0.6.0 (2020-03-31)

- Added PostgreSQL backend
- Added Sqlite backend and made it the default (replacing *InMemoryRepository*)
- Refactored test suite to cover all conformance/integration tests on all backends
- Refactored Redis backend, simplifying the Lua scripts and improving exceptional case handling (e.g. tasks disappearing between query and poll)
- Main loop only sleeps for the rest of remaining *poll_interval* before next tick instead of the full amount
- General bug fixes, documentation changes, clean up

7.3 0.5.0 (2020-03-27)

- Fixes bug where a locked dynamic task could be executed again on next tick.
- poll_task is now reentrant with regards to locking. If the lease passed in matches the lease on the task, it behaves as though it were unlocked.

7.4 0.4.0 (2020-02-16)

- Middleware support and optional metrics via Prometheus
- Improved the graceful shutdown behavior
- Task instance and application context are now available in the task context
- Breaking change: dynamic task parameters are now accessed via *context.args['name']* instead of *context.name*
- Improved examples, documentation and packaging

7.5 0.2.0 (2020-01-08)

- Timezone support
- More efficient poling when Redis backend is used

7.6 0.1.1 (2020-01-08)

- First release that actually works.

7.7 0.0.0 (2019-12-31)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

U

W