Pyncette

Release 0.10.1

May 09, 2023

Contents

1 Overview **1**
- 1.1 Installation . . . 1
- 1.2 Documentation . . . 2
- 1.3 Usage example . . . 2
- 1.4 Use cases . . . 2
- 1.5 Supported backends . . . 3
- 1.6 Development . . . 3

2 Installation **5**

3 Usage **7**
- 3.1 Running the main loop . . . 7
- 3.2 Specifying the schedule . . . 8
- 3.3 Customizing tasks . . . 8
- 3.4 Middlewares . . . 10
- 3.5 Fixtures . . . 11
- 3.6 Persistence . . . 11
- 3.7 Heartbeating . . . 12
- 3.8 Dynamic tasks . . . 12
- 3.9 Once-off dynamic tasks . . . 13
- 3.10 Performance . . . 13

4 Backends **15**
- 4.1 SQLite . . . 15
- 4.2 Redis . . . 15
- 4.3 PostgreSQL . . . 16
- 4.4 MySQL . . . 16
- 4.5 Amazon DynamoDB . . . 16

5 Advanced usage **19**
- 5.1 Partitioned dynamic tasks . . . 19

6 pyncette package **21**
- 6.1 Submodules . . . 21
- 6.2 Module contents . . . 30

7 Contributing **33**

7.1 Bug reports . . . 33
7.2 Documentation improvements . . . 33
7.3 Feature requests and feedback . . . 33
7.4 Development . . . 34

8 Authors **37**

9 Changelog **39**
9.1 0.10.1 (2023-05-09) . . . 39
9.2 0.10.0 (2023-05-08) . . . 39
9.3 0.8.1 (2021-04-08) . . . 39
9.4 0.8.0 (2021-04-05) . . . 39
9.5 0.7.0 (2021-03-31) . . . 40
9.6 0.6.1 (2020-04-02) . . . 40
9.7 0.6.0 (2020-03-31) . . . 40
9.8 0.5.0 (2020-03-27) . . . 40
9.9 0.4.0 (2020-02-16) . . . 40
9.10 0.2.0 (2020-01-08) . . . 41
9.11 0.1.1 (2020-01-08) . . . 41
9.12 0.0.0 (2019-12-31) . . . 41

10 Indices and tables **43**

Python Module Index **45**

Index **47**

CHAPTER 1

Overview

docs	
tests	
package	

A reliable distributed scheduler with pluggable storage backends for Async Python.

- Free software: MIT license

1.1 Installation

Minimal installation (just SQLite persistence):

```
pip install pyncette
```

Full installation (all the backends and Prometheus metrics exporter):

```
pip install pyncette[all]
```

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 Use cases

Pyncette is designed for reliable (at-least-once or at-most-once) execution of recurring tasks (think cronjobs) whose lifecycles are managed dynamically, but can work effectively for non-reccuring tasks too.

Example use cases:

- You want to perform a database backup every day at noon
- You want a report to be generated daily for your 10M users at the time of their choosing
- You want currency conversion rates to be refreshed every 10 seconds
- You want to allow your users to schedule non-recurring emails to be sent at an arbitrary time in the future

Pyncette might not be a good fit if:

- You want your tasks to be scheduled to run (ideally) once as soon as possible. It is doable, but you will be better served by a general purpose reliable queue like RabbitMQ or Amazon SQS.

- You need tasks to execute at sub-second intervals with low jitter. Pyncette coordinates execution on a per task-instance basis and this corrdination can add overhead and jitter.

1.5 Supported backends

Pyncette comes with an implementation for the following backends (used for persistence and coordination) out-of-the-box:

- SQLite (included)
- Redis (`pip install pyncette[redis]`)
- PostgreSQL (`pip install pyncette[postgres]`)
- MySQL 8.0+ (`pip install pyncette[mysql]`)
- Amazon DynamoDB (`pip install pyncette[dynamodb]`)

Pyncette imposes few requirements on the underlying datastores, so it can be extended to support other databases or custom storage formats / integrations with existing systems. For best results, the backend needs to provide:

- Some sort of serialization mechanism, e.g. traditional transactions, atomic stored procedures or compare-and-swap
- Efficient range queries over a secondary index, which can be eventually consistent

1.6 Development

To run integration tests you will need Redis, PostgreSQL, MySQL and Localstack (for DynamoDB) running locally.

To run the all tests run:

```
tox
```

Alternatively, there is a Docker Compose environment that will set up all the backends so that integration tests can run seamlessly:

```
docker-compose up -d
docker-compose run --rm shell
tox
```

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

```
tox -e py310  # or your Python version of choice
```

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 MySQL peristence:

```
pip install pyncette[mysql]
```

For installing with Amazon DynamoDB peristence:

```
pip install pyncette[dynamodb]
```

For installing with PostgreSQL peristence:

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

For installing with Prometheus metrics exporter:

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

For a full installation with all the extras:

```
pip install pyncette[all]
```

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 Disabling a task

Tasks can be disabled by passing an `enabled=False` in the parameters. This can be used for example to conditionally enable tasks only on certain instances.

```
@app.task(schedule="* * * * *", enabled=False)
async def task1(context: Context):
    print(f"This will never run.")
```

Tasks can be disabled also in the initialization code:

```
from pyncette import Pyncette, Context

app = Pyncette()

@app.task(schedule="* * * * *")
async def task1(context: Context):
    print(f"This will never run.")

async with app.create() as app_context:
    task1.enabled = False
    await app_context.run()
```

3.3.5 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()
```

(continues on next page)

(continued from previous page)

```
    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()

@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.

See *Backends* for instructions on how to configure persistence for a database of your choice.

3.7 Heartbeating

If have tasks that have an unpredictable run time, it can be hard to come up with an appropriate lease duration in advance. If set too short, lease will expire, leading to duplicate task execution and if too long, there can be insufficient protection against unhealthy workers.

A way to mitigate is to use heartbeating. Heartbeating will periodically extend the lease on the task as long as task is still running. Pyncette supports two approaches to heartbeating:

- Cooperative heartbeating: your task periodically calls `context.heartbeat()` to extend the lease
- Automatic heartbeating: your task is decorated with *with_heartbeat()* and it heartbeats automatically in the background for as long as the task is executing.

Beware that automatic heartbeating can potentially be dangerous if, for example, your task is stuck in an infinite loop or an I/O operation that does not have a proper time out. In this case the lease can be kept alive indefinitely and the task will not make any progress. Cooperative heartbeating may be more verbose, but offers a greater degree of control.

If `context.heartbeat()` is called when the lease is already lost, the call will raise *LeaseLostException*, allowing you to bail out early, since another instance is likely already processing the same task.

```
from pyncette.utils import with_heartbeat

@app.task(schedule='* * * * * */10')
@with_heartbeat()
async def foo(context: Context):
    # The task will be kept alive by the heartbeat
    await asyncio.sleep(3600)

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

3.8 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 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.

3.9 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.

3.10 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.

Depending on the backend used, having a dynamic task with a very large number of instances can lead to diminished performance. See *Partitioned dynamic tasks* for a way to address this issue.

CHAPTER 4

Backends

By default Pyncette runs without persistence. This means that the schedule is maintained 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.

4.1 SQLite

SQLite is the default peristence engine and is included in the base Python package.

```
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()
```

4.2 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')
```

4.3 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 unless `postgres_skip_table_create` is set to `True`.

4.4 MySQL

MySQL can be configured by passing *mysql_repository()* as `repository_factory` parameter to the *Pyncette* constructor.

The MySQL backend requires MySQL version 8.0+.

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

app = Pyncette(
    repository_factory=mysql_repository,
    mysql_host="localhost",
    mysql_database="pyncette",
    mysql_user="pyncette",
    mysql_password="password",
    mysql_table_name='pyncette_tasks'
)
```

The table will be automatically initialized on startup if it does not exists unless `mysql_skip_table_create` is set to `True`.

Caution: MySQL backend currently does not work with Python 3.10 due to an issue with an upstream library.

4.5 Amazon DynamoDB

Amazon DynamoDB backend can be configured with *dynamodb_repository()*.

```
from pyncette import Pyncette, Context
from pyncette.dynamodb import dynamodb_repository

app = Pyncette(
    repository_factory=dynamodb_repository,
    dynamodb_region_name="eu-west-1",
    dynamodb_table_name="pyncette",
)
```

DynamoDB repository will use ambient credentials, such as environment variables, `~/.aws/config` or EC2 metadata service if e.g. running on EC2 or a Kubernetes cluster with kiam/kube2iam.

For convenience, an appropriate DynamoDB table will be automatically created on startup if it does not exist. The created table uses on-demand pricing model. If you would like to customize this behavior, you can manually create the table beforehand and pass `dynamodb_skip_table_create=True` in parameters.

Expected table schema should look something like this

```
{
    "AttributeDefinitions": [
        { "AttributeName": "partition_id", "AttributeType": "S" },
        { "AttributeName": "ready_at", "AttributeType": "S" },
        { "AttributeName": "task_id", "AttributeType": "S" }
    ],
    "KeySchema": [
        { "AttributeName": "partition_id", "KeyType": "HASH" },
        { "AttributeName": "task_id", "KeyType": "RANGE" }
    ],
    "LocalSecondaryIndexes": [
        {
            "IndexName": "ready_at",
            "KeySchema": [
                { "AttributeName": "partition_id", "KeyType": "HASH" },
                { "AttributeName": "ready_at", "KeyType": "RANGE" }
            ],
            "Projection": {
                "ProjectionType": "ALL"
            }
        }
    ]
}
```

CHAPTER 5

Advanced usage

5.1 Partitioned dynamic tasks

Certain backends, like Redis and Amazon DynamoDB have a natural partitioning to them. Generally, when using dynamic tasks, the task name is used as a partition key. For example, in DynamoDB, each dynamic task instance is associated with one row/document, but they all share the same partition id.

Similarly for Redis, each task instance record is stored in its own key, but the index that sets them in order of next execution is stored in a single key, so a single large task will not benefit from a clustered Redis setup.

If there is a very large number of dynamic task instances associated with a single task or they are polled very frequently, this can lead to hot partitions and degraded performance. There can also be limits as to how many task instances can even be stored in a single partition. For DynamoDB, the limit is 10GB.

Pyncette supports transparent partitioning of tasks through *partitioned_task()* decorator.

```
from pyncette import Pyncette, Context

app = Pyncette()

@app.partitioned_task(partition_count=32)
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()
```

This splits the dynamic task into 32 partitions and the task instances are automatically assigned to them based on the hash of the task instance name.

The default partition selector uses SHA1 hash of the instance name, but a custom selector can be provided:

```
def custom_partition_selector(partition_count: int, task_id: str) -> int:
    return hash(task_id) % partition_count  # Do not use this, as the hash() is not␣
↪stable

@app.partitioned_task(
    partition_count=32,
    partition_selector=custom_partition_selector
)
async def hello(context: Context) -> None:
    print(f"Hello {context.args['username']}")
```

5.1.1 Choosing the partition count

Care must be taken when selecting a pertition count, as it is not easy to change it later after tasks have already been scheduled. Changing a partition count will generally map task instances to a different partition, making them not run and also making it impossible to unschedule them through *`unschedule_task()`*.

There is also a tradeoff as the time complexity as a single Pyncette poll grows linearly with the total number of tasks (or their partitions). Setting the number of partitions too high can lead to diminished performance due to the polling overhead.

It is possible to configure Pyncette to only poll certain partitions using the `enabled_partitions` parameter. This will allow the tasks to be scheduled and unscheduled by any application instance, but only the partitions selected will be polled. You may use this if you have a large number of instances for a given task in order to spread the load evenly among them.

```
@app.partitioned_task(
    partition_count=8,
    # Partitions 4, 5, 6 and 7 will not be polled
    enabled_partitions=[0, 1, 2, 3]
)
async def hello(context: Context) -> None:
    print(f"Hello {context.args['username']}")
```

CHAPTER 6

pyncette package

6.1 Submodules

6.1.1 pyncette.dynamodb module

class pyncette.dynamodb.**DynamoDBRepository**(*dynamo_resource: Any*, *skip_table_create: bool*, *partition_prefix: str*, ***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.

extend_lease(*utc_now: datetime.datetime*, *task: Task*, *lease: Lease*) → Lease | None
Extends the lease on the task. Returns the new lease if lease was still valid.

initialize() → None

poll_dynamic_task(*utc_now: datetime.datetime*, *task: Task*, *continuation_token: ContinuationToken | None = None*) → QueryResponse
Queries the dynamic tasks for execution

poll_task(*utc_now: datetime.datetime*, *task: Task*, *lease: Lease | None = None*) → 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.dynamodb.**dynamodb_repository**(*, *dynamodb_endpoint: str | None = None*, *dynamodb_region_name: str | None = None*, *dynamodb_skip_table_create: bool = False*, *dynamodb_partition_prefix: str = ''*, ***kwargs*) → AsyncIterator[DynamoDBRepository]

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

6.1.2 pyncette.errors module

exception pyncette.errors.**LeaseLostException**(*task: Task*)

Bases: *pyncette.errors.PyncetteException*

Signals that the lease on the task was lost

exception pyncette.errors.**PyncetteException**

Bases: Exception

Base exception for Pyncette

6.1.3 pyncette.mysql module

class pyncette.mysql.**MySQLRepository**(*pool: aiomysql.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.

extend_lease(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → Optional[NewType.<locals>.new_type]

Extends the lease on the task. Returns the new lease if lease was still valid.

initialize() → None

poll_dynamic_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *continuation_token: Optional[NewType.<locals>.new_type] = None*) → 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.mysql.**mysql_repository**(*, *mysql_host: str*, *mysql_user: str*, *mysql_database: str*, *mysql_password: Optional[str] = None*, *mysql_port: int = 3306*, ***kwargs*) → AsyncIterator[pyncette.mysql.MySQLRepository]

Factory context manager that initializes the connection to MySQL

6.1.4 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.Protocol

class pyncette.model.**Heartbeater**(**args*, ***kwargs*)
Bases: typing.Protocol

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

class pyncette.model.**NextFunc**(**args*, ***kwargs*)
Bases: typing.Protocol

class pyncette.model.**PartitionSelector**(**args*, ***kwargs*)
Bases: typing.Protocol

class pyncette.model.**PollResponse**(*result: ResultType*, *scheduled_at: datetime.datetime*, *lease: Lease | None*)
Bases: object

The result of a task poll

class pyncette.model.**QueryResponse**(*tasks: list[tuple[pyncette.task.Task, Lease]]*, *continuation_token: ContinuationToken | None*)
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.Protocol

6.1.5 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.

extend_lease(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → Optional[NewType.<locals>.new_type]
Extends the lease on the task. Returns the new lease if lease was still valid.

initialize() → None

poll_dynamic_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *continuation_token: Optional[NewType.<locals>.new_type] = None*) → 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 repository that initializes the connection to Postgres

6.1.6 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.

extend_lease(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → Optional[NewType.<locals>.new_type]
Extends the lease on the task. Returns the new lease if lease was still valid.

poll_dynamic_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *continuation_token: Optional[NewType.<locals>.new_type] = None*) → 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

6.1.7 pyncette.pyncette module

class pyncette.pyncette.**Pyncette**(*repository_factory: pyncette.repository.RepositoryFactory = <function sqlite_repository>*, *executor_cls: type = <class 'pyncette.executor.DefaultExecutor'>*, *poll_interval: datetime.timedelta = datetime.timedelta(seconds=1)*, ***kwargs*)
Bases: object

Pyncette application.

create(*context_items: dict[str, Any] | None = None*) → AsyncIterator[PyncetteContext]
Creates the execution context.

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

fixture(*name: str | None = None*) → Decorator[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

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

task(***kwargs*) → Callable[[pyncette.model.TaskFunc], pyncette.task.Task]
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

add_to_context(*name: str*, *value: Any*) → None
Adds a value with a given key to task context

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: Task*, *instance_name: str | None = None*) → None
Removes the concrete instance of a dynamic task

6.1.8 pyncette.redis module

class pyncette.redis.**RedisRepository**(*redis_client: redis.asyncio.client.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.

extend_lease(*utc_now: datetime.datetime*, *task: Task*, *lease: Lease*) → Lease | None
Extends the lease on the task. Returns the new lease if lease was still valid.

poll_dynamic_task(*utc_now: datetime.datetime*, *task: Task*, *continuation_token: ContinuationToken | None = None*) → QueryResponse
Queries the dynamic tasks for execution

poll_task(*utc_now: datetime.datetime*, *task: Task*, *lease: Lease | None = None*) → 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[pyncette.redis.RedisRepository]
Factory context manager for Redis repository that initializes the connection to Redis

6.1.9 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.

extend_lease(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *lease: NewType.<locals>.new_type*) → Optional[NewType.<locals>.new_type]
Extends the lease on the task. Returns the new lease if lease was still valid.

poll_dynamic_task(*utc_now: datetime.datetime*, *task: pyncette.task.Task*, *continuation_token: Optional[NewType.<locals>.new_type] = None*) → 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.Protocol

A factory context manager for creating a repository

6.1.10 pyncette.executor module

class pyncette.executor.**DefaultExecutor**(***kwargs*)
Bases: contextlib.AbstractAsyncContextManager

Manages the spawned tasks running in background

spawn_task(*task: Awaitable*) → None

class pyncette.executor.**SynchronousExecutor**(***kwargs*)
Bases: contextlib.AbstractAsyncContextManager

spawn_task(*task: Awaitable*) → None

6.1.11 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: Optional[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

6.1.12 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.

extend_lease(*utc_now: datetime.datetime, task: pyncette.task.Task, lease: NewType.<locals>.new_type*) → Optional[NewType.<locals>.new_type]
Extends the lease on the task. Returns the new lease if lease was still valid.

initialize() → None

poll_dynamic_task(*utc_now: datetime.datetime, task: pyncette.task.Task, continuation_token: Optional[NewType.<locals>.new_type] = None*) → 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

6.1.13 pyncette.task module

class pyncette.task.**PartitionedTask**(***, *partition_count: int*, *partition_selector: PartitionSelector = <function _default_partition_selector>*, *enabled_partitions: list[int] | None = None*, ***kwargs*)
Bases: *pyncette.task.Task*

get_partitions() → list

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

class pyncette.task.**Task**(***, *name: str*, *func: TaskFunc*, *enabled: bool = True*, *dynamic: bool = False*, *parent_task: Task | None = None*, *schedule: str | None = None*, *interval: datetime.timedelta | None = None*, *execute_at: datetime.datetime | None = None*, *timezone: str | None = None*, *fast_forward: bool = False*, *failure_mode: FailureMode = <FailureMode.NONE: 0>*, *execution_mode: 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
Serializes all the attributes to task spec

canonical_name
A unique identifier for a task instance

enabled

get_next_execution(*utc_now: datetime.datetime*, *last_execution: datetime.datetime | None*) → datetime.datetime | None

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

instantiate_from_spec(*task_spec: dict*) → pyncette.task.Task
Deserializes all the attributes from task spec

6.1.14 pyncette.utils module

pyncette.utils.**with_heartbeat**(*lease_remaining_ratio: float = 0.5*, *cancel_on_lease_lost: bool = False*) → Callable[[pyncette.model.TaskFunc], pyncette.model.TaskFunc]

Decorate the task to use automatic heartbeating in background.

Parameters

- **lease_remaining_ratio** – Number between 0 and 1. The ratio between elapsed time and the lease duration when heartbeating will be performed. Default is 0.5.
- **cancel_on_lease_lost** – Whether the task should be cancelled if lease expires. Default is False.

6.2 Module contents

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

Bases: object

Pyncette application.

create(*context_items: dict[str, Any] | None = None*) → AsyncIterator[PyncetteContext]
Creates the execution context.

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

fixture(*name: str | None = None*) → Decorator[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

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

task(***kwargs*) → Callable[[pyncette.model.TaskFunc], pyncette.task.Task]
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

add_to_context(*name: str*, *value: Any*) → None
Adds a value with a given key to task context

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: Task*, *instance_name: str | None = None*) → None
Removes the concrete instance of a dynamic task

CHAPTER 7

Contributing

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

7.1 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.

7.2 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.

7.3 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 :)

7.4 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.

4. Running integration tests assumes that there will be Redis, PostgreSQL, MySQL and Localstack (for DynamoDB) running on localhost. Alternatively, there is a Docker Compose environment that will set up all the backends so that integration tests can run seamlessly:

```
docker-compose up -d
docker-compose run --rm shell
```

5. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

6. Pyncette uses *black* and *isort* to enforce formatting and import ordering. If you want to auto-format the code, you can do it like this:

```
tox -e check
```

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.

If you run into issues setting up a local environment or testing the code locally, feel free to submit the PR anyway and GitHub Actions will test it for you.

7.4.1 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. Update documentation when there's new API, functionality etc.
2. Add a note to `CHANGELOG.rst` about the changes.
3. Add yourself to `AUTHORS.rst`.

7.4.2 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.wiki/en/latest/user_guide.html#parallel-mode)):

```
tox --parallel auto
```

CHAPTER 8

Authors

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

CHAPTER 9

Changelog

9.1 0.10.1 (2023-05-09)

- Include missing lua files in the built wheel

9.2 0.10.0 (2023-05-08)

- Drop support for Python 3.7
- Add support for Python 3.11
- Modernize Python package structure and linters
- Fix a few bugs and type annotations

9.3 0.8.1 (2021-04-08)

- Improve performance for calculation of the next execution time
- Add ability for repositories to pass a pagination token
- Add *add_to_context()* to inject static data to context
- Clean up documentation and add additional examples

9.4 0.8.0 (2021-04-05)

- Added Amazon DynamoDB backend
- Added MySQL backend

- Added support for partitioned dynamic tasks

9.5 0.7.0 (2021-03-31)

- Added support for automatic and cooperative lease heartbeating
- PostgreSQL backend can now skip automatic table creation
- Improved signal handling
- CI: Add Codecov integration
- Devenv: Run integration tests in Docker Compose

9.6 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.

9.7 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

9.8 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.

9.9 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

9.10 0.2.0 (2020-01-08)

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

9.11 0.1.1 (2020-01-08)

- First release that actually works.

9.12 0.0.0 (2019-12-31)

- First release on PyPI.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

pyncette, 30
pyncette.dynamodb, 21
pyncette.errors, 22
pyncette.executor, 28
pyncette.healthcheck, 28
pyncette.model, 23
pyncette.mysql, 22
pyncette.postgres, 24
pyncette.prometheus, 24
pyncette.pyncette, 25
pyncette.redis, 26
pyncette.repository, 27
pyncette.sqlite, 28
pyncette.task, 29
pyncette.utils, 30

Index

A

add_to_context() (*pyncette.pyncette.PyncetteContext method*), 26
add_to_context() (*pyncette.PyncetteContext method*), 31
as_spec() (*pyncette.task.Task method*), 29
AT_LEAST_ONCE (*pyncette.ExecutionMode attribute*), 30
AT_LEAST_ONCE (*pyncette.model.ExecutionMode attribute*), 23
AT_MOST_ONCE (*pyncette.ExecutionMode attribute*), 30
AT_MOST_ONCE (*pyncette.model.ExecutionMode attribute*), 23

C

canonical_name (*pyncette.task.Task attribute*), 29
COMMIT (*pyncette.FailureMode attribute*), 31
COMMIT (*pyncette.model.FailureMode attribute*), 23
commit_task() (*pyncette.dynamodb.DynamoDBRepository method*), 21
commit_task() (*pyncette.mysql.MySQLRepository method*), 22
commit_task() (*pyncette.postgres.PostgresRepository method*), 24
commit_task() (*pyncette.prometheus.MeteredRepository method*), 24
commit_task() (*pyncette.redis.RedisRepository method*), 26
commit_task() (*pyncette.repository.Repository method*), 27
commit_task() (*pyncette.sqlite.SqliteRepository method*), 28
Context (*class in pyncette*), 31
Context (*class in pyncette.model*), 23
create() (*pyncette.Pyncette method*), 30
create() (*pyncette.pyncette.Pyncette method*), 25

D

default_healthcheck() (*in module pyncette.healthcheck*), 28
DefaultExecutor (*class in pyncette.executor*), 28
dynamic_task() (*pyncette.Pyncette method*), 30
dynamic_task() (*pyncette.pyncette.Pyncette method*), 26
dynamodb_repository() (*in module pyncette.dynamodb*), 21
DynamoDBRepository (*class in pyncette.dynamodb*), 21

E

enabled (*pyncette.task.Task attribute*), 29
ExecutionMode (*class in pyncette*), 30
ExecutionMode (*class in pyncette.model*), 23
extend_lease() (*pyncette.dynamodb.DynamoDBRepository method*), 21
extend_lease() (*pyncette.mysql.MySQLRepository method*), 22
extend_lease() (*pyncette.postgres.PostgresRepository method*), 24
extend_lease() (*pyncette.prometheus.MeteredRepository method*), 24
extend_lease() (*pyncette.redis.RedisRepository method*), 26
extend_lease() (*pyncette.repository.Repository method*), 27
extend_lease() (*pyncette.sqlite.SqliteRepository method*), 28

F

FailureMode (*class in pyncette*), 30
FailureMode (*class in pyncette.model*), 23
fixture() (*pyncette.Pyncette method*), 30
fixture() (*pyncette.pyncette.Pyncette method*), 26
FixtureFunc (*class in pyncette.model*), 23

G

get_next_execution() (*pyncette.task.Task method*), 29

get_partitions() (*pyncette.task.PartitionedTask method*), 29

H

Heartbeater (*class in pyncette.model*), 23

I

initialize() (*pyncette.dynamodb.DynamoDBRepository method*), 21
initialize() (*pyncette.mysql.MySQLRepository method*), 22
initialize() (*pyncette.postgres.PostgresRepository method*), 24
initialize() (*pyncette.pyncette.PyncetteContext method*), 26
initialize() (*pyncette.PyncetteContext method*), 31
initialize() (*pyncette.sqlite.SqliteRepository method*), 28
instantiate() (*pyncette.task.PartitionedTask method*), 29
instantiate() (*pyncette.task.Task method*), 29
instantiate_from_spec() (*pyncette.task.Task method*), 29

L

last_tick (*pyncette.pyncette.PyncetteContext attribute*), 26
last_tick (*pyncette.PyncetteContext attribute*), 31
LEASE_MISMATCH (*pyncette.model.ResultType attribute*), 23
LeaseLostException, 22
LOCKED (*pyncette.model.ResultType attribute*), 23

M

main() (*pyncette.Pyncette method*), 30
main() (*pyncette.pyncette.Pyncette method*), 26
measure() (*pyncette.prometheus.OperationMetricSet method*), 25
MeteredRepository (*class in pyncette.prometheus*), 24
middleware() (*pyncette.Pyncette method*), 30
middleware() (*pyncette.pyncette.Pyncette method*), 26
MiddlewareFunc (*class in pyncette.model*), 23
MISSING (*pyncette.model.ResultType attribute*), 23
mysql_repository() (*in module pyncette.mysql*), 22
MySQLRepository (*class in pyncette.mysql*), 22

N

NextFunc (*class in pyncette.model*), 23
NONE (*pyncette.FailureMode attribute*), 31
NONE (*pyncette.model.FailureMode attribute*), 23

O

OperationMetricSet (*class in pyncette.prometheus*), 25

P

partitioned_task() (*pyncette.Pyncette method*), 30
partitioned_task() (*pyncette.pyncette.Pyncette method*), 26
PartitionedTask (*class in pyncette.task*), 29
PartitionSelector (*class in pyncette.model*), 23
PENDING (*pyncette.model.ResultType attribute*), 23
poll_dynamic_task() (*pyncette.dynamodb.DynamoDBRepository method*), 21
poll_dynamic_task() (*pyncette.mysql.MySQLRepository method*), 22
poll_dynamic_task() (*pyncette.postgres.PostgresRepository method*), 24
poll_dynamic_task() (*pyncette.prometheus.MeteredRepository method*), 24
poll_dynamic_task() (*pyncette.redis.RedisRepository method*), 27
poll_dynamic_task() (*pyncette.repository.Repository method*), 27
poll_dynamic_task() (*pyncette.sqlite.SqliteRepository method*), 28
poll_task() (*pyncette.dynamodb.DynamoDBRepository method*), 21
poll_task() (*pyncette.mysql.MySQLRepository method*), 22
poll_task() (*pyncette.postgres.PostgresRepository method*), 24
poll_task() (*pyncette.prometheus.MeteredRepository method*), 25
poll_task() (*pyncette.redis.RedisRepository method*), 27
poll_task() (*pyncette.repository.Repository method*), 27
poll_task() (*pyncette.sqlite.SqliteRepository method*), 28
PollResponse (*class in pyncette.model*), 23
postgres_repository() (*in module pyncette.postgres*), 24
PostgresRepository (*class in pyncette.postgres*), 24
prometheus_fixture() (*in module pyncette.prometheus*), 25

prometheus_middleware() (*in module pyncette.prometheus*), 25
Pyncette (*class in pyncette*), 30
Pyncette (*class in pyncette.pyncette*), 25
pyncette (*module*), 30
pyncette.dynamodb (*module*), 21
pyncette.errors (*module*), 22
pyncette.executor (*module*), 28
pyncette.healthcheck (*module*), 28
pyncette.model (*module*), 23
pyncette.mysql (*module*), 22
pyncette.postgres (*module*), 24
pyncette.prometheus (*module*), 24
pyncette.pyncette (*module*), 25
pyncette.redis (*module*), 26
pyncette.repository (*module*), 27
pyncette.sqlite (*module*), 28
pyncette.task (*module*), 29
pyncette.utils (*module*), 30
PyncetteContext (*class in pyncette*), 31
PyncetteContext (*class in pyncette.pyncette*), 26
PyncetteException, 22

Q

QueryResponse (*class in pyncette.model*), 23

R

READY (*pyncette.model.ResultType attribute*), 23
redis_repository() (*in module pyncette.redis*), 27
RedisRepository (*class in pyncette.redis*), 26
register_scripts() (*pyncette.redis.RedisRepository method*), 27
register_task() (*pyncette.dynamodb.DynamoDBRepository method*), 21
register_task() (*pyncette.mysql.MySQLRepository method*), 22
register_task() (*pyncette.postgres.PostgresRepository method*), 24
register_task() (*pyncette.prometheus.MeteredRepository method*), 25
register_task() (*pyncette.redis.RedisRepository method*), 27
register_task() (*pyncette.repository.Repository method*), 27
register_task() (*pyncette.sqlite.SqliteRepository method*), 29
Repository (*class in pyncette.repository*), 27
RepositoryFactory (*class in pyncette.repository*), 27
ResultType (*class in pyncette.model*), 23
run() (*pyncette.pyncette.PyncetteContext method*), 26
run() (*pyncette.PyncetteContext method*), 31

S

schedule_task() (*pyncette.pyncette.PyncetteContext method*), 26
schedule_task() (*pyncette.PyncetteContext method*), 31
shutdown() (*pyncette.pyncette.PyncetteContext method*), 26
shutdown() (*pyncette.PyncetteContext method*), 31
spawn_task() (*pyncette.executor.DefaultExecutor method*), 28
spawn_task() (*pyncette.executor.SynchronousExecutor method*), 28
sqlite_repository() (*in module pyncette.sqlite*), 29
SqliteRepository (*class in pyncette.sqlite*), 28
SynchronousExecutor (*class in pyncette.executor*), 28

T

Task (*class in pyncette.task*), 29
task() (*pyncette.Pyncette method*), 30
task() (*pyncette.pyncette.Pyncette method*), 26
TaskFunc (*class in pyncette.model*), 24

U

UNLOCK (*pyncette.FailureMode attribute*), 31
UNLOCK (*pyncette.model.FailureMode attribute*), 23
unlock_task() (*pyncette.dynamodb.DynamoDBRepository method*), 21
unlock_task() (*pyncette.mysql.MySQLRepository method*), 22
unlock_task() (*pyncette.postgres.PostgresRepository method*), 24
unlock_task() (*pyncette.prometheus.MeteredRepository method*), 25
unlock_task() (*pyncette.redis.RedisRepository method*), 27
unlock_task() (*pyncette.repository.Repository method*), 27
unlock_task() (*pyncette.sqlite.SqliteRepository method*), 29
unregister_task() (*pyncette.dynamodb.DynamoDBRepository method*), 21
unregister_task() (*pyncette.mysql.MySQLRepository method*), 22
unregister_task() (*pyncette.postgres.PostgresRepository method*), 24
unregister_task() (*pyncette.prometheus.MeteredRepository method*), 25
unregister_task() (*pyncette.redis.RedisRepository method*),

27
unregister_task() (*pyncette.repository.Repository method*), 27
unregister_task() (*pyncette.sqlite.SqliteRepository method*), 29
unschedule_task() (*pyncette.pyncette.PyncetteContext method*), 26
unschedule_task() (*pyncette.PyncetteContext method*), 31
use_fixture() (*pyncette.Pyncette method*), 30
use_fixture() (*pyncette.pyncette.Pyncette method*), 26
use_healthcheck_server() (*in module pyncette.healthcheck*), 28
use_middleware() (*pyncette.Pyncette method*), 30
use_middleware() (*pyncette.pyncette.Pyncette method*), 26
use_prometheus() (*in module pyncette.prometheus*), 25

W

with_heartbeat() (*in module pyncette.utils*), 30
with_prometheus_repository() (*in module pyncette.prometheus*), 25