Tamarco Documentation

Release 0.1.0

System73

Apr 08, 2020

CONTENTS

CHAPTER

ONE

TAMARCO

Microservices framework designed for asyncio and Python.

1.1 Features

- Lifecycle management.
- Standardized settings via etcd.
- Automatic logging configuration. Support for sending logs to an ELK stack.
- Application metrics via Prometheus.
- Designed for asyncio.
- Messaging patterns. The framework comes with support for AMQP and Kafka via external resources. The AMQP resource has implemented publish/subscribe, request/response and push/pull patterns.
- Custom encoders and decoders.
- Pluging oriented architecture. Anyone can create a new resource to add new functionality. External resources are integrated into the framework transparently for the user.
- Graceful shutdown.

1.2 Resources

The framework allows to write external resources and integrate them in the lifecycle of a microservice easily. List with the available resources:

- Metrics
- Registry
- Status
- Profiler
- Memory analizer

- HTTP
- Kafka
- AMQP
- Postgres (not released yet)
- Influxdb (not released yet)
- Redis (not released yet)
- Websocket (not released yet)

Let us know if you have written a resource.

1.3 Documentation

The documentation is available here.

1.4 Examples

There are several examples in the `examples` folder.

To run them, install tamarco, launch the docker-compose (not necessary for all the examples) and run it.

```
pip install tamarco
docker-compose up -d
python examples/http_resource/microservice.py
```

1.5 Requirements

Support for Python >= 3.6.

CHAPTER

TWO

TUTORIALS

2.1 Quick Start

1. Install Tamarco

To install Tamarco, simply run this command in your terminal of choice. Allowed Python versions are Python >= 3.6. Recommended version is Python 3.7:

```
$ pip3 install tamarco
```

2.1.1 Start a project

Use Tamarco to start a new project. Use the following command and fill the data requested:

```
$ tamarco start_project
```

2.1.2 Write your microservice

Start writing the microservice code inside the project folder, in *microservice.py*.

2.1.3 Run the microservice

Run the microservice with the following command:

```
$ python3 app.py
```

2.2 Write your first microservice

In this section, we will create a simple microservice that inserts data to a Postgres table.

2.2.1 Installation

For this example, we need the Tamarco framework and the Postgres resource plugin. Optionally, you can create a virtual environment before installing the packages:

```
$ virtualenv virtualenv -p python3.6
$ . virtualenv/bin/activate
$ pip3 install tamarco tamarco-postgres
```

2.2.2 Using Tamarco code generation

Tamarco provides the generation of a microservice skeleton using cookiecutter. templates. To use this feature, go to the path where you want to create the microservice and type:

```
$ tamarco start_project
```

This command will ask you a few questions to get a minimum service configuration and will generate the code in a new folder named with the chosen *project_name*. The main script file is called *microservice.py* and for simplification, we will code all our example in this file.

More information about the microservice code generation: microservice_cookiecutter_template.

2.2.3 Our microservice step by step

The code generated in *microservice.py* is very simple:

```
from tamarco.core.microservice import Microservice


class MyMicroservice(Microservice):
    name = "my_awesome_project_name"


def main():
    ms = MyMicroservice()
    ms.run()
```

In the previous code, we can see that our service inherits from the Tamarco base class *Microservice*. This class will be the base of all the microservices and it is responsible for starting all the resources and at the same time stop all the resources properly when the microservice exits. It has several execution stages in its lifecycle. For more information see: microservice_base_class.

The next step is to declare the Postgres resource we want to use:

```
from tamarco.core.microservice import Microservice
from tamarco-postgres import PostgresClientResource


class MyMicroservice(Microservice):
    name = "my_awesome_project_name"
    postgres = PostgresClientResource()
```

In a production environment, we normally get the service settings/configuration from a storage service like etcd, but to simplify, now we set the required configuration using an internal function. More info about the Tamarco settings in: *A walk around the settings*.

```
from tamarco.core.microservice import Microservice
from tamarco-postgres import PostgresClientResource

```

(continues on next page)

(continued from previous page)

```
class MyMicroservice(Microservice):
    name = "my_awesome_project_name"
    postgres = PostgresClientResource()

    def __init__(self):
        super().__init__()
        self.settings.update_internal({
            "system": {
                "deploy_name": "my_first_microservice",
                "logging": {
                    "profile": "DEVELOP",
                },
                "resources": {
                    "postgres": {
                        "host": "127.0.0.1",
                        "port": 5432,
                        "user": "postgres"
                    }
                }
            }
        })
```

Our service already knows where to connect to the database, so, we have to create the table and make the queries. Tamarco provides a decorator (*@task*) to convert a method in an asyncio task. The task is started and stopped when the microservice starts and stops respectively:

```
from tamarco.core.microservice import Microservice, task
from tamarco-postgres import PostgresClientResource

class MyMicroservice(Microservice):
    name = "my_awesome_project_name"
    postgres = PostgresClientResource()

    def __init__(self):
        super().__init__()
        self.settings.update_internal({
            "system": {
                "deploy_name": "my_first_microservice",
                "logging": {
                    "profile": "DEVELOP",
                },
                "resources": {
                    "postgres": {
                        "host": "127.0.0.1",
                        "port": 5432,
                        "user": "postgres"
                    }
                }
            }
        })

    @task
    async def postgres_query(self):
        create_query = '''
            CREATE TABLE my_table (
                id INT PRIMARY KEY NOT NULL,
```

(continues on next page)

(continued from previous page)

```
                name TEXT NOT NULL
            );
        '''
        insert_query = "INSERT INTO my_table (id, name) VALUES (1, 'John Doe');"
        select_query = "SELECT * FROM my_table"

        try:
            await self.postgres.execute(create_query)
            await self.postgres.execute(insert_query)
            response = await self.postgres.fetch(select_query)
        except Exception:
            self.logger.exception("Error executing query")
        else:
            self.logger.info(f"Data: {response}")
```

NOTICE that we imported *task* from tamarco.core.microservice!!

2.2.4 Running our microservice

Firstly, we need a running Postgres, so we can launch a docker container:

```
$ docker run -d -p 5432:5432 postgres
```

In the root of our project, there is the service entry point: *app.py*. You can execute this file and check the result (don't forget to activate the virtualenv if you have one):

```
$ python app.py
```

CHAPTER

THREE

HOW-TO GUIDES

3.1 How to install Tamarco

Tamarco is compatible with Python >= 3.6. Recommended version is Python 3.7.

To install Tamarco, simply run this command in your terminal of choice:

```
$ pip3 install tamarco
```

3.2 How to make doc

`bash $ make docs `

The documentation is generated in *docs/_build/html/*.

3.3 How to setup the logging

3.3.1 The profile

Two different profiles are allowed:

- DEVELOP. The logging level is set to debug.
- PRODUCTION. The logging level is set to info.

The profile setting needs to be in capital letters.

```
system:
  logging:
    profile: <DEVELOP or PRODUCTION>
```

3.3.2 Stdout

The logging by stdout can be enabled or disabled:

It comes with the

```
system:
  logging:
    stdout: true
```

3.3.3 File handler

Write all logs in files with a *RotatingFileHandler*. It is enabled when the system/logging/file_path exits, saving the logs in the specified location.

```
system:
  logging:
    file_path: <file_path>
```

3.3.4 Logstash

Logstash is the log collector used by Tamarco, it collects, processes, enriches and unifies all the logs sent by different components of an infrastructure. Logstash supports multiple choices for the log ingestion, we support three of them simply by activating the corresponding settings:

Logstash UDP handler

Send logs to Logstash using a raw UDP socket.

```
system:
  logging:
    logstash:
      enabled: true
      host: 127.0.0.1
      port: 5044
      fqdn: false
      version: 1
```

Logstash Redis handler

Send logs to Logstash using the Redis pubsub pattern.

```
system:
  logging:
    redis:
      enabled: true
      host: 127.0.0.1
      port: 6379
      password: my_password
      ssl: false
```

Logstash HTTP handler

Send logs to Logstash using HTTP requests.

```
system:
  logging:
    http:
      enabled: true
      url: http://127.0.0.1
      user:
      password:
```

(continues on next page)

(continued from previous page)

```
      max_time_seconds: 15
      max_records: 100
```

The logs are sent in bulk, the max_time_seconds is the maximum time without sending the logs, the max_records configures the maximum number of logs in a single HTTP request (The first condition triggers the request).

3.4 How to setup a metric backend

The Microservice class comes by default with the metrics resource, this means that the microservice is going to read the configuration without any explicit code in your microservice.

3.4.1 Prometheus

Prometheus, unlike other metric backends, follows a pull-based (over HTTP) architecture at the metric collection. It means that the microservices just have the responsibility of exposing the metrics via an HTTP server and Prometheus collects the metrics requesting them to the microservices.

It is the supported metric backend with a more active development right now.

The metrics resource uses other resource named tamarco_http_report_server, that it is an HTTP server, to expose the application metrics. The metrics always are exposed to the */metrics* endpoint. To expose the Prometheus metrics the microservices should be configured as follows:

```
system:
  resources:
    metrics:
      collect_frequency: 10
      handlers:
        prometheus:
          enabled: true
    tamarco_http_report_server:
      host: 127.0.0.1
      port: 5747
```

With this configuration, a microservice is going to expose the Prometheus metrics at http://127.0.0.1:5747/metrics.

The collect frequency defines the update period in seconds of the metrics in the HTTP server.

The microservice name is automatically added as metric suffix to the name of the metrics. Example: A summary named http_response_time in a microservice named billing_api is going to be named billing_api_http_response_time in the exposed metrics.

3.4.2 Carbon

Only the plaintext protocol sent directly via a TCP socket is supported.

To configure a carbon handler:

```
system:
  resources:
    metrics:
      handlers:
        carbon:
```

(continues on next page)

(continued from previous page)

```
          enabled: true
          host: 127.0.0.1
          port: 2003
    collect_frequency: 15
```

The collect frequency defines the period in seconds where the metrics are collected and sent to carbon.

3.4.3 File

It is an extension of the carbon handler, instead of sending the metrics to carbon it just appends the metrics to a file. The format is the following: *<metric path> <metric value> <metric timestamp>*.

To configure the file handler:

```
system:
  resources:
    metrics:
      handlers:
        file:
          enabled: true
          path: /tmp/tamarco_metrics
    collect_frequency: 15
```

The collect frequency defines the period in seconds where the metrics are collected and written to a file.

3.4.4 Stdout

It is an extension of the carbon handler, instead of sending the metrics to carbon it just writes the metrics in the stdout. The format is the following: *<metric path> <metric value> <metric timestamp>*.

To configure the file handler:

```
system:
  resources:
    metrics:
      handlers:
        stdout:
          enabled: true
    collect_frequency: 15
```

The collect frequency defines the period in seconds where the metrics are collected and written to a file.

3.5 How to setup a setting backend

There are some ways to set up the settings, etcd is the recommended backend for a centralized configuration. The YML and file and dictionary are useful for development.

3.5.1 etcd

etcd is the recommended backend for a centralized configuration. All the configuration of the system can be in etcd, but before being able to read it, we should specify to the microservices how to access an etcd.

The following environment variables need to be properly configured to use etcd:

- TAMARCO_ETCD_HOST: Needed to setup the etcd as setting backend.
- TAMARCO_ETCD_PORT: Optional variable, by default is 2379.
- ETCD_CHECK_KEY: Optional variable, if set the microservice waits until the specified etcd key exits to initialize.

Avoids race conditions between the etcd and microservices initialization. Useful in orchestrators such docker-swarm where dependencies between components cannot be easily specified.

3.5.2 YML file

For enable the feature, the following environment variable must be set:

- TAMARCO_YML_FILE: Example: 'settings.yml'. Example of a YML file with the system configuration:

```yaml
system:
  deploy_name: test_tamarco
  logging:
    profile: DEVELOP
    file: false
    stdout: true
    redis:
      enabled: false
      host: "127.0.0.1"
      port: 7006
      password: ''
      ssl: false
  microservices:
    test:
      logging:
        profile: DEVELOP
        file: false
        stdout: true
  resources:
    metrics:
      collect_frequency: 15
    status:
      host: 127.0.0.1
      port: 5747
      debug: False
    amqp:
      host: 127.0.0.1
      port: 5672
      vhost: /
      user: microservice
      password: 1234
      connection_timeout: 10
      queues_prefix: "prefix"
```

3.5.3 Dictionary

It is possible to load the configuration from a dictionary:

```
import asyncio

from sanic.response import text

from tamarco.core.microservice import Microservice, MicroserviceContext, thread
from tamarco.resources.io.http.resource import HTTPClientResource, HTTPServerResource


class HTTPMicroservice(Microservice):
    name = 'settings_from_dictionary'
    http_server = HTTPServerResource()

    def __init__(self):
        super().__init__()
        self.settings.update_internal({
            'system': {
                'deploy_name': 'settings_documentation',
                'logging': {
                    'profile': 'PRODUCTION',
                },
                'resources': {
                    'http_server': {
                        'host': '127.0.0.1',
                        'port': 8080,
                        'debug': True
                    }
                }
            }
        })


ms = HTTPMicroservice()


@ms.http_server.app.route('/')
async def index(request):
    print('Requested /')
    return text('Hello world!')


def main():
    ms.run()


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

3.6 How to setup settings for a specific microservice

The settings under *system.microservice.<microservice_name>.<setting_paths_to_override>* overrides the general settings of *system.<setting_paths_to_override>* in the microservice named <microservice_name>.

In the following example, the microservice dog is going to read the logging profile "DEVELOP" and the other microservices are going to stay in the logging profile "PRODUCTION":

```
system:
  deploy_name: tamarco_doc
  logging:
    profile: PRODUCTION
    file: false
    stdout: true
  microservices:
    dog:
      logging:
        profile: DEVELOP
```

The microservice name is declared when the microservice class is defined:

```
class MicroserviceExample(Microservice):

    name = 'my_microservice_name'
```

3.7 How to setup settings for a resource

The resources are designed to automatically load their configuration using the setting resource.

The resources should be defined as an attribute of the microservice class:

```
class MyMicroservice(Microservice):
    name = 'settings_from_dictionary'

    recommendation_http_api = HTTPServerResource()
    billing_http_api = HTTPServerResource()

    def __init__(self):
        super().__init__()
        self.settings.update_internal({
            'system': {
                'deploy_name': 'settings_documentation',
                'logging': {
                    'profile': 'PRODUCTION',
                },
                'resources': {
                    'recommendation_http_api': {
                        'host': '127.0.0.1',
                        'port': 8080,
                        'debug': True
                    },
                    'billing_http_api': {
                        'host': '127.0.0.1',
                        'port': 9090,
                        'debug': False
                    }
                }
            }
        })
```

The resources load their configuration based on the name of the attribute used to bind the resource to the microservice. In the example, we have two HTTPServerResource in the same microservice and each one uses a different configuration.

The HTTPServerResource recommendations_api variable is going to find its configuration in the path 'system.resources.recommendation_api'.

You must be cautious about choosing the name when the instances are created. If several microservices use the same database, the name of the resource instance in the microservice must be the same in all microservices to load the same configuration.

3.8 How to use the logging resource

Tamarco uses the standard logging library, it only interferes doing an automatic configuration based in the settings.

The microservice comes with a logger ready to use:

```
import asyncio

from tamarco.core.microservice import Microservice, task


class MyMicroservice(Microservice):
    name = 'my_microservice_name'

    extra_loggers_names.append("my_extra_logger")

    @task
    async def periodic_log(self):
        logging.getlogger("my_extra_logger").info("Initializing periodic log")
        while True:
            await asyncio.sleep(1)
            self.logger.info("Sleeping 1 second")

if __name__ == "__main__":
    ms = MyMicroservice()
    ms.run()
```

Also can configured more loggers adding their names to *my_extra_logger* list of the Microservice class.

The logger bound to the microservice is the one named as the microservice, so you can get and use the logger whatever you want:

```
import logging

async def http_handler():
    logger = logging.getlogger('my_microservice_name')
    logger.info('Handling a HTTP request')
```

3.8.1 Logging exceptions

A very common pattern programming microservices is log exceptions. Tamarco automatically sends the exception tracing to Logstash and print the content by stdout when the exc_info flag is active. Only works with logging lines inside an except statement:

```
import asyncio

from tamarco.core.microservice import Microservice, task

```

(continues on next page)

(continued from previous page)

```
class MyMicroservice(Microservice):
    name = 'my_microservice_name'

    @task
    async def periodic_exception_log(self):
        while True:
            try:
                raise KeyError
            except:
                self.logger.warning("Unexpected exception.", exc_info=True)


if __name__ == "__main__":
    ms = MyMicroservice()
    ms.run()
```

3.8.2 Adding extra fields and tags

The fields extend the logging providing more extra information and the tags allow to filter the logs by this key.

A common pattern is to enrich the logs with some information about the context. For example: with a request identifier the trace can be followed by various microservices.

This fields and tags are automatically sent to Logstash when it is configured.

```
logger.info("logger line", extra={'tags': {'tag': 'tag_value'}, 'extra_field': 'extra_
↪field_value'})
```

3.8.3 Default logger fields

Automatically some extra fields are added to the logging.

- *deploy_name*: deploy name configured in *system/deploy_name*, it allows to distinguish logs of different deploys,

for example between staging, develop and production environments. * *levelname*: log level configured currently in the Microservice. * *logger*: logger name used when the logger is declared. * *service_name*: service name declared in the Microservice.

3.9 How to use metrics resource

All Tamarco meters implement the Flyweight pattern, this means that no matter where you instantiate the meter if two or more meters have the same characteristics they are going to be the same object. You don't need to be careful about using the same object in multiple places.

3.9.1 Counter

A counter is a cumulative metric that represents a single numerical value that only goes up. The counter is reseated when the server restart. A counter can be used to count requests served, events, tasks completed, errors occurred, etc.

```
cats_counter = Counter('cats', 'animals')
meows_counter = Counter('meows', 'sounds')
jumps_counter = Counter('jumps', 'actions')

class Cat:

    def __init__(self):
        cats_counter.inc()

    # It can work as a decorator, every time a function is called, the counter is␣
↪increased in one.
    @meows_counter
    def meow(self):
        print('meow')

    # Similarly it can be used as a decorator of coroutines.
    @jumps_counter
    async def jump(self):
        print("jump")
```

3.9.2 Gauge

A gauge is a metric that represents a single numerical value. Unlike the counter, it can go down. Gauges are typically used for measured values like temperatures, current memory usage, coroutines, CPU usage, etc. You need to take into account that this kind of data only save the last value when it is reported.

It is used similarly to the counter, a simple example:

```
ws_connections_metric = Gauge("websocket_connections", "connections")

class WebSocketServer:

    @ws_connections_metric
    def on_open(self):
        ...

    def on_close(self):
        ws_connections_metric.dec()
        ...
```

3.9.3 Summary

A summary samples observations over sliding windows of time and provides instantaneous insight into their distributions, frequencies, and sums). They are typically used to get feedback about quantities where the distribution of the data is important, as the processing times.

The default quantiles are: [0.5, 0.75, 0.9, 0.95, 0.99].

3.9.4 Timer

Gauge and Summary can be used as timers. The timer admits to be used as a context manager and as a decorator:

```
request_processing_time = Summary("http_requests_processing_time", "time")

@request_processing_time.timeit()
def http_handler(request):
    ...
```

```
import time

my_task_processing_time_gauge = Gauge("my_task_processing_time", "time")

with my_task_processing_time_gauge.timeit()
     my_task()
```

3.9.5 Labels

The metrics admit labels to attach additional information in a counter. For example, the status code of an HTTP response can be used as a label to monitoring the amount of failed requests.

A meter with labels:

```
http_requests_ok = Counter('http_requests', 'requests', labels={'status_code': 200})

def http_request_ping(request):
    http_requests_ok.inc()
    ...
```

To add a label to an already existent meter:

CHAPTER

FOUR

EXPLANATIONS

4.1 A walk around the settings

Tamarco is an automation framework for managing the lifecycle and resources of the microservices. The configuration has a critical role in the framework, all the other resources and components of the framework strongly depend on the settings.

When you have thousands of microservices running in production the way to provide the configuration of the system becomes critical. Some desirable characteristics of a microservice settings framework are:

1) The configuration should be centralized. A microservice compiled in a container should be able to run in different
   environments without any change in the code. For example, the network location of a database or its credentials aren't going to be the same in a production environment or a staging environment.
2) The configuration should be able to change in runtime without restarting the microservices. For example, you should
   be able to update the configuration of your WebSocket server without close the existing connections.
3) The configuration should have redundancy. One of the advantages of a microservice architecture is the facility to
   obtain redundancy in your services, you should be able to run the microservices in several machines if someone fails, the others should be able to work correctly. Nothing of this has a sense if your services aren't able to read the configuration, so to take the benefits of this architectural advantage, all critical services of the system must be redundant as well.

The external backend supported by this framework right now is etcd v2, we strongly recommend its use in production with Tamarco.

Other settings backends are available to develop:

- Dictionary
- File based (YML or JSON)

4.1.1 Settings structure

The settings can be configured from a simple YAML in etcd [Link of how to configure an etcd from a file]. A generic setting could be the following:

```
etcd_ready: true
system:
   deploy_name: tamarco_tutorial
```

(continues on next page)

(continued from previous page)

```
logging:
    profile: PRODUCTION
    file: false
    stdout: true
resources:
    amqp:
        host: 172.31.0.102
        port: 5672
        vhost: /
        user: guest
        password: guest
        connection_timeout: 10
        queues_prefix: ""
    kafka:
        bootstrap_servers: 172.31.0.1:9092,172.31.0.2:9092
microservices:
    http_server:
        application_cache_seconds: 10
```

The etcd_ready setting is written by the etcd configuration script when it finishes configuring all the other settings. This prevents the microservices from reading the settings before the environment is properly configured.

All the other tamarco settings are inside a root_path named "system". The settings under the root path are:

- Deploy_name. Name that identifies a deploy, used by default by logging and metrics resources with the purpose of
  distinct logs and metrics from different deploys. Possible use cases: allow to filter logs of deploys in different regions or by develop, staging and production with the same monitoring system.
- Logging: Configuration of the logging of the system, it is out of resources because this configuration can't be avoided since it is a core component, all the microservices and all resources emit logs. More information about the
  possible configuration in [TODO link to logging section].
- Resources: configurations of the resources of the system, it can be used by one or more microservices. See: setup_setting_for_a_resource.
- Microservice: configuration of the business logic of each microservice. This section also has a special property, all the other settings can be configured by in this section for a specific microservice. See:
  setup_setting_for_a_specific_microservice.

4.2 Microservice lifecycle

4.2.1 Start

When the microservice is initialized, the following steps are performed, automatically:

1. Start provisional logging with default parameters. Needed in case of some error before being able to read the final
   logging configuration from the settings.
2. Initialize the settings. All the other resources of the framework depend on being able to read the centralized configuration.
3. Initialize the logging with the proper settings. With the settings available, the next step is to be sure that all

the resources can send proper log messages in case of failure before starting them.

4. Call the pre_start of the microservice, that triggers the pre_start of the microservices. Operations that need to be performed before starting the microservice. For example, a HTTP server could need to render some templates before
   start the server. It is not advisable to perform I/O operations in the pre_start statement.
5. Call the start of the microservice, they are going to start all the resources. In the start statement the resources are expected to perform the initial I/O operations, start a server, connect to a database, etc.
6. Call the post_start of the microservice, it is going to call the post_start of all the resources. In this step all the resources should be working normally because they should be started in the previous step.

Tamarco builds a dependency graph of the order in that the resources should be initialized.

4.2.2 Status of a resource

All the resources should report their state, it can be one of the followings:

1. NOT_STARTED
2. CONNECTING
3. STARTED
4. STOPPING
5. STOPPED
6. FAILED

The status of all the resources are exposed via an HTTP API and used by the default restart policies to detect when a resource is failing.

4.2.3 Resource restart policies

The status resources come by default with the microservice and their responsibility is to apply the restart policies of the microservice and report the state of the resources via an HTTP API.

There are two settings to control automatically that a resource should do when it has a FAILED status:

```
system:
    resources:
        status:
            restart_policy:
                resources:
                    restart_microservice_on_failure: ['redis']
                    restart_resource_on_failure: ['kafka']
```

Where the microservice is identified by the name of the resource instance in the microservice class.

Keep in mind that the most recommended way is not to use these restart policies and implement a circuit breaker in each resource. But sometimes you could want a simpler solution and in some cases, the default restart policies can be an acceptable way to go.

4.2.4 Stop

The shut down of a microservice can be triggered by a restart policy (restart_microservice_on_failure), by a system signal, by a resource (not recommended, a resource shouldn't have the responsibility of stopping a service) or by business code.

A service only should be stopped calling the method *stop_gracefully* of the microservice instance.

The shut down is performed doing the following steps:

1. Call stop() method of the microservice, it is going to call the stop() of all the resources.
2. Call post_stop() method of the microservice, it is going to call the post_stop() method of all the resources.
3. The exit is going to be forced after 30 seconds if the microservice didn't finish the shut down in this time or some resource raises an exception stopping the service.

4.2.5 Overwrite lifecycle methods

The lifecycle methods are designed to be overwritten by the user, allowing to execute code at a certain point of the lifecycle. Just take into account that these methods are asynchronous and that the *super()* method should be called.

The available methods are:

- pre_start
- start
- post_start
- stop
- post_stop

```
from tamarco import Microservice

class LifecycleMicroservice(Microservice):

    async def pre_start(self):
        print("Before pre_start of the service")
        await super().pre_start()
        print("After pre_start of the service")

    async def start(self):
        print("Before start of the service")
        await super().start()
        print("After start of the service")

    async def post_start(self):
        print("Before post_start of the service")
        await super().start()
        print("After post_start of the service")

    async def stop(self):
        print("Before stop of the service")
        await super().stop()
        print("After stop of the service")

    async def post_stop(self):
        print("Before post_stop of the service")
```

(continues on next page)

(continued from previous page)

```
        await super().stop()
        print("After post_stop of the service")


def main():
    microservice = LifecycleMicroservice(Microservice)
    microservice.run()

def __name__ == '__main__':
    main()
```

4.3 Microservice base class

All the microservices must inherit from the Tamarco Microservice class. Let's take a deeper look into this class.

To launch the microservice, we use the *run* function:

```
.. code-block:: python
```

from tamarco.core.microservice import Microservice

class MyMicroservice(Microservice): [...]

ms = MyMicroservice() ms.run()

When we run the microservice, there is a certain order in the setup of the service and then the event loop is running until an unrecoverable error occurs, or it is stopped.

Setup steps:

1. Configure and load the microservice settings (and of its resources if used). 1. Configure and start the logging service. 1. Pre-start stage: run all the Tamarco resources *pre_start* methods (only the resources actually used by the microservice). This method can be overriden if we want to do some coding in this step. But don't forget to call to the Tamarco function too!

1. Start stage: run all the Tamarco resources *start* methods (only the resources actually used by the microservice). Also collects all the task declared in the microservice (using the *@task* decorator in a method) and launch them. Generally in this stage is when the database connections, or other services used by the resources are started. This *start* method can be overriden if we want to do some coding in this step. But don't forget to call to the Tamarco function too! 1. Post-start stage: run all the Tamarco resources *post_start* methods (only the resources actually used by the microservice). This method can be overriden if we want to do some coding in this step. But don't forget to call to the Tamarco function too! 1. Stop stage: run all the Tamarco resources *stop* methods (only the resources actually used by the microservice). In this stage all resources and tasks are stopped. This method can be overriden if we want to do some coding in this step. But don't forget to call to the Tamarco function too! 2. Post-stop stage: run all the Tamarco resources *post_stop* methods (only the resources actually used by the microservice). This step is useful if you want to make some instructions when the microservice stops. This *post_stop* method can be overriden if we want to do some coding in this step. But don't forget to call to the Tamarco function too!

4.4 Microservice cookicutter template

When you install the tamarco python package is available a _tamarco_ command. Calling this command you can create a new microservice skeleton answering before a few questions:

```
$ tamarco start_project
```

1. Project name: project name. In the same directory when you execute the tamarco command the script will create a folder with this name and all the initial files insite it. Used also in the docs and README files. 1. Project slug: project short name. Inside of the project name folder, a folder with this name is created and all the microservice logic code should be here. Used also in the docs files. 1. Full name: author's full name. Used in the docs files. 1. Email: author's email. Used in the docs files. 1. Version: initial project version. It will be copied to the setup.cfg file. 1. Project short description: this text will be in the initial README file created.

The project skeleton will be:

```
<project_name>
   |
   |- docs (folder with the files to generate Sphinx documentation)
   |
   |- tests (here will be store the microservice tests)
   |
   |- <project_slug>
   |      |
   |      |- logic (microservice business logic code)
   |      |
   |      |- resources (code related with the microservice resources: databases, ...)
   |      |
   |      |- meters.py (application meters: prometheus, ...)
   |      |
   |      |- microservice.py (microservice class inherited from Tamarco Microservice␣
↪class)
   |
   |- .coveragerc (coverage configuration file)
   |
   |- .gitignore
   |
   |- app.py (entrypoint file for the microservice)
   |
   |- Dockerfile
   |
   |- HISTORY.md
   |
   |- Makefile (run the tests, generate docs, create virtual environments, install␣
↪requirements, ...)
   |
   |- README.md
   |
   |- requirements.txt
   |
   |- setup.cfg (several python packages configurations: bumpversion, flake8, pytest,␣
↪...)
   |
```

CHAPTER

FIVE

REFERENCE

5.1 Core

class tamarco.core.microservice.**Microservice**

Main class of a microservice. This class is responsible for controlling the lifecycle of the microservice, also builds and provides the necessary elements that a resource needs to work.

The resources of a microservice should be declared in this class. The microservice automatically takes the ownership of all the declared resources.

async post_start()

Post start stage of lifecycle. This method can be overwritten by the user to add some logic in the start.

async post_stop()

Post stop stage of the lifecycle. This method can be overwritten by the user to add some logic to the shut down.

async pre_start()

Pre start stage of lifecycle. This method can be overwritten by the user to add some logic in the start.

run()

Run a microservice. It initializes the main event loop of asyncio, so this function only are going to end when the microservice ends its live cycle.

async start()

Start stage of lifecycle. This method can be overwritten by the user to add some logic in the start.

async stop()

Stop stage of the lifecycle. This method can be overwritten by the user to add some logic to the shut down. This method should close all the I/O operations opened by the resources.

async stop_gracefully()

Stop the microservice gracefully. Shut down the microservice. If after 30 seconds the microservice is not closed gracefully it forces a exit.

class tamarco.core.microservice.**MicroserviceContext**

"This class is used to use tamarco resources without using a full microservice, for example a script.

tamarco.core.microservice.**task**(*name_or_fn*)

Decorator to convert a method of a microservice in a asyncio task. The task is started and stopped when the microservice starts and stops respectively.

Parameters **name_or_fn** – Name of the task or function. If function the task name is the declared name of the function.

tamarco.core.microservice.**task_timer**(*interval=1000*, *one_shot=False*, *autostart=False*) → Union[collections.abc.Callable, Coroutine]

Decorator to declare a task that should repeated in time intervals.

Examples

```
>>> @task_timer()
>>> async def execute(*arg,**kwargs)
>>>     print('tick')
```

```
>>> @task_timer(interval=1000, oneshot=True, autostart=True)
>>> async def execute(*args,**kwargs)
>>>     print('tick')
```

Parameters

- **interval** (*int*) – Interval in milliseconds when the task is repeated.
- **one_shot** (*bool*) – Only runs the task once.
- **autostart** (*bool*) – Task is automatically initialized with the microservice.

tamarco.core.microservice.**thread**(*name_or_fn*)

Decorator to convert a method of a microservice in a thread. The thread is started and stopped when the microservice starts and stops respectively.

Parameters **name_or_fn** – Name of the thread or function. If function the thread name is the declared name of the function.

5.1.1 Logging

class tamarco.core.logging.logging.**Logging**

Class that handles the configuration of the standard logging of python using the microservice settings.

configure_settings(*settings*)

Sets the settings object (a SettingsView(f"{ROOT_SETTINGS}.logging")).

Parameters **settings** (*SettingsInterface*) – Settings object that have the logging settings.

static describe_dynamic_settings()

Describe all the class dynamic settings.

Returns Settings and their description.

Return type dict

static describe_static_settings()

Describe all the settings as a dictionary keys and their values are a setting short description. These settings are the static settings needed by the class.

Returns Settings and their description.

Return type dict

async start(*loggers*, *microservice_name*, *deploy_name*, *loop*)

Configure the standard python logging, adding handlers and loggers that uses that handlers.

Parameters

- **loggers** (*list*) – Names of the loggers you want to configure.
- **microservice_name** (*str*) – Name of the microservice that will use the logging.
- **deploy_name** (*str*) – Deploy name.
- **loop** – asyncio event loop.

5.1.2 Patterns

class tamarco.core.patterns.**Singleton**
Singleton pattern implementation.

This pattern restricts the instantiation of a class to one object.

class tamarco.core.patterns.**Proxy**(*obj*)
Proxy pattern to be used as a pointer. When the value of _obj changes, the reference to the proxy remains.

static make_method(*name*)
Create a new method to getting the value of the attribute *name*.

Parameters **name** (*string*) – Attribute name.

Returns New __getattribute__ method to get a value from the _obj object.

Return type function

class tamarco.core.patterns.**Flyweight**(*name*, *bases*, *dct*)
Metaclass that implements the Flyweight pattern.

It is like a Singleton but only for the instances with the same key. The key is first parameter that you pass to the class when you create the object.

This class is conceived for the internal use of the Tamarco metrics library.

Example:

```
>>> class Metric(metaclass=Flyweight):
>>>     def __init__(self, metric_id):
>>>         self.metric_id = metric_id
>>>
>>> http_requests_1 = Metric('http_requests')
>>> http_requests_2 = Metric('http_requests')
>>>
>>> http_requests_1 == http_requests_2
True
```

class tamarco.core.patterns.**FlyweightWithLabels**(*name*, *bases*, *dct*)
Metaclass that extends the pattern of the Flyweight pattern with labels.

This class is conceived for the internal use of the Tamarco metrics library.

Example:

```
>>> class Metric(metaclass=FlyweightWithLabels):
>>>     def __init__(self, metric_id, labels=None):
>>>         self.metric_id = metric_id
>>>         self.labels = labels if labels else {}
>>>
>>> requests_http_get_1 = Metric('request', labels={'protocol': 'http', 'method':
↪'get'})
>>> requests_http_post_1 = Metric('request', labels={'protocol': 'http', 'method
↪': 'post'})
```

(continues on next page)

(continued from previous page)

```
>>>
>>> requests_http_get_2 = Metric('request', labels={'protocol': 'http', 'method':
↪'get'})
>>> requests_http_post_2 = Metric('request', labels={'protocol': 'http', 'method
↪': 'post'})
>>>
>>> requests_http_get_1 == requests_http_get_2
True
>>> requests_http_post_1 == requests_http_post_2
True
```

5.1.3 Settings

exception tamarco.core.settings.settings.**SettingNotFound**(*key*)

class tamarco.core.settings.settings.**Settings**

Core settings class, here is the unique True of settings all the settings values are cached by this class in his internal_backend, all of the other settings are views of the data that this class holds.

The external backend is where the settings should be originally loaded, the internal backend acts as cache to avoid making many requests to the external backend.

async bind(*loop*)

Binds the settings to one event loop.

Parameters **loop** – Main asyncio event loop.

async cancel_watch_tasks()

Cancel all the pending watcher tasks of the settings in the etcd backend.

async delete(*key*)

Delete a setting.

Parameters **key** (*str*) – Path to the setting.

async get(*key*, *default=<class 'tamarco.core.settings.backends.interface._EmptyArg'>*)

Get a setting value for a key.

Parameters

- **key** (*str*) – Path to the setting.
- **default** – Default value in the case that it doesn't exists.

Raises ***SettingNotFound*** – The setting can't be resolved and it hasn't default value.

Returns Setting value.

async get_external(*key*, *default=<class 'tamarco.core.settings.backends.interface._EmptyArg'>*)

Get the setting from the external backend updating the internal one with the value of the external.

Parameters

- **key** (*str*) – Path to the setting.
- **default** – Default value in case that the setting doesn't exists in the external backend.

Returns Setting value.

register_promised_setting(*key*, *promised_setting*)

Register a SettingProxy to be resolved when the settings are loaded.

Parameters

- **key** (*str*) – setting key to register.
- **promised_setting** – setting proxy to register.

async set(*key*, *value*)
Set a setting value.

Parameters

- **key** (*str*) – Path to the setting.
- **value** – Value to be set in the setting key.

async start()
Start the settings. First loads the settings from the external settings backend (etcd or yaml file) once the internal and external settings backends are ready, the promised settings (when_loaded_settings) are resolved and the proxies start to holds the settings values.

async stop()
Perform all the needed tasks in order to stop the Settings.

update_internal(*dict_settings*)
Update the internal cache with new settings.

Parameters **dict_settings** (*dict*) – Settings to add to the internal backend.

async update_internal_settings(*key*, *value*)
Update an specific internal setting.

Parameters

- **key** (*str*) – Path to the setting.
- **value** – Setting value.

async watch(*key*, *callback*)
Schedule a callback for when a setting is changed in the etcd backend.

Parameters

- **key** (*str*) – Path to the setting.
- **callback** – function or coroutine to be called when the setting changes, it should have with two input arguments, one for the setting path and other for the setting value.

async watch_and_update(*key*)
Watch one specific settings and maintain it updated in the internal settings.

Parameters **key** (*str*) – Path to the setting.

exception tamarco.core.settings.settings.**SettingsNotLoadedYet**

class tamarco.core.settings.settings.**SettingsView**(*settings*, *prefix*, *microservice_name=None*)
View/chroot/jail/box of main settings class. Used in the resources to provide them with their subset of settings.

async cancel_watch_tasks()
Cancel all the pending watcher tasks of the settings in the etcd backend.

async delete(*key*, *raw=False*)
Delete a setting.

Parameters

- **key** (*str*) – Path to the setting.

- **raw** – If True no prefix is used so is not a view.

async get(*key*, *default=<class 'tamarco.core.settings.backends.interface._EmptyArg'>*, *raw=False*)
Get setting.

Parameters

- **key** (*str*) – Path to the setting.
- **default** – Default value in case that the setting doesn't exists in the external backend.
- **raw** – if True no prefix is used so is not a view.

async set(*key*, *value*, *raw=False*)
Set a setting value.

Parameters

- **key** (*str*) – Path to the setting.
- **default** – Default value in the case that it doesn't exists.
- **raw** – If True no prefix is used so is not a view.

Returns Setting value.

async update_internal_settings(*key*, *value*)
Update internal settings.

Parameters

- **key** (*str*) – Path to the setting.
- **value** – Setting value.

async watch(*key*, *callback*, *raw=False*)
Schedule a callback for when a setting is changed in the etcd backend.

Parameters

- **key** (*str*) – Path to the setting.
- **callback** – Callback to run whenever the *key* changes.
- **raw** – If True no prefix is used so is not a view.

5.2 Resources

class tamarco.resources.bases.**BaseResource**
Define the basic interface of a resource. All the tamarco resources should inherit from this class.

Resource start call chain:

1. bind
2. configure_settings
3. pre_start
4. start
5. post_start

Resource stop call chain:

1. stop

2. post_stop

async bind(*microservice*, *name*)
Build method, the microservice binds all its resources. Microservice starts and stops the resources.

Parameters

- **microservice** (Microservice) – Microservice instance managing the resource.
- **name** (*str*) – Name of the resource instance in the microservice class.

async configure_settings(*settings*)
Build method, the microservice provides the settings class of each resource. The resource should read the settings via this object.

Parameters **settings** (SettingsView) – Settings view of the resource.

async post_start()
Post start stage of the resource lifecycle.

async post_stop()
Post stop stage of the resource lifecycle.

async pre_start()
Pre start stage of the resource lifecycle.

async start()
Start stage of the resource lifecycle.

async status() → dict
Return information about the state of the resource.

async stop()
Stop stage of the resource lifecycle.

class tamarco.resources.bases.**DatabaseResource**(**args*, ***kwargs*)

async start(*clean_database=False*, *register_scripts=True*)
Start stage of the resource lifecycle.

async status()
Return information about the state of the resource.

async stop()
Stop stage of the resource lifecycle.

class tamarco.resources.bases.**IOResource**(*inputs: List = None*, *outputs: List = None*)
Extended resource that manages I/O streams, like Kafka and AMQP.

add_input(*input_to_add*)
Add one input.

Parameters **input_to_add** (*InputBase*) – Input to add.

add_output(*output*)
Add one output.

Parameters **output** (*OutputBase*) – Output to add.

CHAPTER

SIX

CONTRIBUTION GUIDE

Welcome to the project!! First of all we want to thank you, we would like to have new collaborators and contributions.

This project is governed by the Tamarco Code of Conduct and we expect that all our members follow them.

6.1 Your first contribution

There are so many ways to help, improve the documentation, write tutorials or examples, improve the docstrings, make tests, report bugs, etc.

You can take a look at the tickets with the tag `good first issue`.

6.2 Running tests and linters

All the contributions must have at least unit tests.

Make sure that the test are in the correct place. We have separated the tests in two categories, the unit tests (`test/unit`) and the functional tests (`test/functional`). Inside each folder the test should follow the same structure than the main package. For example, a unit test of `tamarco/core/microservice.py` should be placed in `tests/unit/core/test_microservice.py`.

Functional tests are considered those that do some kind of I/O, such as those that need third party services (AMQP, Kafka, Postgres, ...), open servers (http and websocket resource), manage files or wait for an event. The goal is maintain unit tests that can be passed quickly during development.

Most of the functional tests need docker and docker-compose installed in the system to use some third party services.

Before summit a pull request, please check that all the tests and linters are passing.

```
make test
make linters
```

6.3 Code review process

The project maintainers will leave the feedback.

- You need at least two approvals from core developers.
- The tests and linters should pass in the CI.
- The code must have at least the 80% of coverage.

CHAPTER

SEVEN

CONTRIBUTOR COVENANT CODE OF CONDUCT

7.1 Our Pledge

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

7.2 Our Standards

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

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

Examples of unacceptable behavior by participants include:

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

7.3 Our Responsibilities

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

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

7.4 Scope

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

7.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at opensource@system73.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

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

7.6 Attribution

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

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

R

S

T

U

W