prometheus$_{a}syncDocumentation$

Release 22.2.0

Hynek Schlawack

May 14, 2022

CONTENTS

Release v22.2.0 (*What's new?*).

prometheus-async adds support for asynchronous frameworks to the official Python client for the *Prometheus* metrics and monitoring system.

Currently *asyncio* and *Twisted* on Python 3.7 and later are supported.

It works by wrapping the metrics from the official client:

```
import asyncio

from aiohttp import web
from prometheus_client import Histogram
from prometheus_async.aio import time

REQ_TIME = Histogram("req_time_seconds", "time spent in requests")

@time(REQ_TIME)
async def req(request):
      await asyncio.sleep(1)
      return web.Response(body=b"hello")
```

Even for *synchronous* applications, the metrics exposure methods can be useful since they are more powerful than the one shipped with the official client. For that, helper functions have been added that run them in separate threads (*asyncio*-only).

The source code is hosted on GitHub and the documentation on Read The Docs.

CHAPTER

ONE

USER'S GUIDE

1.1 Installation and Requirements

If you just want to instrument an *asyncio*-based application:

```
$ python -m pip install -U pip
$ python -m pip install prometheus-async
```

If you want to expose metrics using *aiohttp*:

```
$ python -m pip install -U pip
$ python -m pip install prometheus-async[aiohttp]
```

If you want to instrument a Twisted application:

```
$ python -m pip install -U pip
$ python -m pip install prometheus-async[twisted]
```

Warning

Please do not skip the update of *pip*, because *prometheus-async* uses modern packaging features and the installation will most likely fail otherwise.

1.2 asyncio Support

The asyncio-related APIs can be found within the `prometheus_async.aio` package.

1.2.1 Decorator Wrappers

All of these functions take a *prometheus_client* metrics object and can either be applied as a decorator to functions and methods, or they can be passed an `asyncio.Future` for a second argument.

coroutine prometheus_async.aio.**time**(*metric: Observer*) → Callable[[Callable[P, R]], Callable[P, R]]

coroutine prometheus_async.aio.**time**(*metric: Observer*, *future: Awaitable[T]*) → Awaitable[T]

Call `metric.observe(time)` with the runtime in seconds.

Works as a decorator as well as on `asyncio.Future`s.

Returns coroutine function (if decorator) or coroutine.

The most common use case is using it as a decorator:

```
import asyncio

from aiohttp import web
from prometheus_client import Histogram
from prometheus_async.aio import time

REQ_TIME = Histogram("req_time_seconds", "time spent in requests")

@time(REQ_TIME)
async def req(request):
    await asyncio.sleep(1)
    return web.Response(body=b"hello")
```

coroutine prometheus_async.aio.**count_exceptions**(*metric: Incrementer*, *, *exc: type[BaseException] = 'BaseException'*) → Callable[[Callable[P, R]], Callable[P, R]]

coroutine prometheus_async.aio.**count_exceptions**(*metric: Incrementer*, *future: Awaitable[T]*, *, *exc: type[BaseException] = 'BaseException'*) → Awaitable[T]

Call `metric.inc()` whenever *exc* is caught.

Works as a decorator as well as on `asyncio.Future`s.

Returns coroutine function (if decorator) or coroutine.

coroutine prometheus_async.aio.**track_inprogress**(*metric: Gauge*) → Callable[[Callable[P, R]], Callable[P, R]]

coroutine prometheus_async.aio.**track_inprogress**(*metric: Gauge*, *future: Awaitable[T]*) → Awaitable[T]

Call `metrics.inc()` on entry and `metric.dec()` on exit.

Works as a decorator, as well on `asyncio.Future`s.

Returns coroutine function (if decorator) or coroutine.

1.2.2 Metric Exposure

prometheus-async offers methods to expose your metrics using *aiohttp* under `prometheus_async.aio.web`:

coroutine prometheus_async.aio.web.**start_http_server**(*, *addr=''*, *port=0*, *ssl_ctx=None*, *service_discovery=None*)

Start an HTTP(S) server on *addr*:*port*.

If *ssl_ctx* is set, use TLS.

Parameters

- **addr** (*str*) – Interface to listen on. Leaving empty will listen on all interfaces.
- **port** (*int*) – Port to listen on.
- **ssl_ctx** (*ssl.SSLContext*) – TLS settings
- **service_discovery** (*ServiceDiscovery | None*) – see *Service Discovery*

Return type *MetricsHTTPServer*

Deprecated since version 18.2.0: The *loop* argument is a no-op now and will be removed in one year by the earliest.

Changed in version 21.1.0: The *loop* argument has been removed.

prometheus_async.aio.web.**start_http_server_in_thread**(*, *port=0*, *addr=''*, *ssl_ctx=None*, *service_discovery=None*)

Start an asyncio HTTP(S) server in a new thread with an own event loop.

Ideal to expose your metrics in non-asyncio Python 3 applications.

For arguments see *start_http_server()*.

Return type *ThreadedMetricsHTTPServer*

Warning

Please note that if you want to use *uWSGI* together with `start_http_server_in_thread()`, you have to tell *uWSGI* to enable threads using its configuration option or by passing it `--enable-threads`.

Currently the recommended mode to run *uWSGI* with `--master` is broken if you want to clean up using `atexit` handlers.

Therefore the usage of `prometheus_sync.aio.web` together with *uWSGI* is **strongly discouraged**.

---

async prometheus_async.aio.web.**server_stats**(*request*)

Return a web response with the plain text version of the metrics.

Return type `aiohttp.web.Response`

Useful if you want to install your metrics within your own application:

```
from aiohttp import web
from prometheus_async import aio

app = web.Application()
app.router.add_get("/metrics", aio.web.server_stats)
# your other routes go here.
```

class prometheus_async.aio.web.**MetricsHTTPServer**(*socket*, *runner*, *app*, *https*)

A stoppable metrics HTTP server.

Returned by *start_http_server()*. Do *not* instantiate it yourself.

Variables

- **socket** – Socket the server is listening on. namedtuple of either (`ipaddress.IPv4Address`, port) or (`ipaddress.IPv6Address`, port).
- **https** (*bool*) – Whether the server uses SSL/TLS.
- **url** (*str*) – A valid URL to the metrics endpoint.
- **is_registered** (*bool*) – Is the web endpoint registered with a service discovery system?

coroutine close()

Stop the server and clean up.

class prometheus_async.aio.web.**ThreadedMetricsHTTPServer**(*http_server*, *thread*, *loop*)

A stoppable metrics HTTP server that runs in a separate thread.

Returned by *start_http_server_in_thread()*. Do *not* instantiate it yourself.

Variables

- **socket** – Socket the server is listening on. namedtuple of `Socket(addr, port)`.
- **https** (*bool*) – Whether the server uses SSL/TLS.
- **url** (*str*) – A valid URL to the metrics endpoint.
- **is_registered** (*bool*) – Is the web endpoint registered with a service discovery system?

close()

Stop the server, close the event loop, and join the thread.

1.2.3 Service Discovery

Web exposure is much more useful if it comes with an easy way to integrate it with service discovery.

Currently *prometheus-async* only ships integration with a local *Consul* agent using *aiohttp*. We do **not** plan add more.

class prometheus_async.aio.sd.**ConsulAgent**(*, *name='app-metrics'*, *service_id=None*, *tags=()*, *token=None*, *deregister=True*)

Service discovery via a local Consul agent.

Pass as `service_discovery` into *prometheus_async.aio.web.start_http_server()*/ *prometheus_async.aio.web.start_http_server_in_thread()*.

Parameters

- **name** (*str*) – Application name that is used for the name and the service ID if not set.
- **service_id** (*str*) – Consul Service ID. If not set, *name* is used.
- **tags** (*tuple*) – Tags to use in Consul registration.
- **token** (*str*) – A consul access token.
- **deregister** (*bool*) – Whether to deregister when the HTTP server is closed.

Custom Service Discovery

Adding own service discovery methods is simple: all you need is to provide an instance with a coroutine `register(self, metrics_server, loop)` that registers the passed `metrics_server` with the service of your choicer and returns another coroutine that is called for de-registration when the metrics server is shut down.

Have a look at our implementations if you need inspiration or check out the `ServiceDiscovery` `typing.Protocol` in the `types` module

1.3 Twisted Support

The Twisted-related APIs can be found within the `prometheus_async.tx` package.

1.3.1 Decorator Wrappers

`prometheus_async.tx.time`(*metric: Observer*) → Callable[[Callable[P, D]], Callable[P, D]]

`prometheus_async.tx.time`(*metric: Observer*, *deferred: D*) → D

Call `metric.observe(time)` with runtime in seconds.

Can be used as a decorator as well as on `Deferred`s.

Works with both sync and async results.

Returns function or `Deferred`.

The fact it's accepting `Deferred`s is useful in conjunction with `twisted.web` views that don't allow to return a `Deferred`:

```
from prometheus_client import Histogram
from prometheus_async.tx import time
from twisted.internet.task import deferLater
from twisted.web.resource import Resource
from twisted.web.server import NOT_DONE_YET
from twisted.internet import reactor

REQ_TIME = Histogram("req_time_seconds", "time spent in requests")

class DelayedResource(Resource):
   def _delayedRender(self, request):
      request.write("<html><body>Sorry to keep you waiting.</body></html>")
      request.finish()

   def render_GET(self, request):
      d = deferLater(reactor, 5, lambda: request)
      time(REQ_TIME, d.addCallback(self._delayedRender))
      return NOT_DONE_YET
```

`prometheus_async.tx.count_exceptions`(*metric: Incrementer*, **, exc: type[BaseException] = <class 'BaseException'>*) → Callable[P, C]

`prometheus_async.tx.count_exceptions`(*metric: Incrementer*, *deferred: D*, **, exc: type[BaseException] = <class 'BaseException'>*) → D

Call `metric.inc()` whenever *exc* is caught.

Can be used as a decorator or on a `Deferred`.

Returns function (if decorator) or `Deferred`.

`prometheus_async.tx.track_inprogress`(*metric: Gauge*) → Callable[P, C]

`prometheus_async.tx.track_inprogress`(*metric: Gauge*, *deferred: D*) → D

Call `metrics.inc()` on entry and `metric.dec()` on exit.

Can be used as a decorator or on a `Deferred`.

Returns function (if decorator) or `Deferred`.

1.3.2 Metric Exposure

prometheus_client, the underlying *Prometheus* client library, exposes a `twisted.web.resource.Resource` – namely `prometheus_client.twisted.MetricsResource` – that makes it extremely easy to expose your metrics.

```
from prometheus_client.twisted import MetricsResource
from twisted.web.server import Site
from twisted.web.resource import Resource
from twisted.internet import reactor

root = Resource()
root.putChild(b"metrics", MetricsResource())

factory = Site(root)
reactor.listenTCP(8000, factory)
reactor.run()
```

As a slightly more in-depth example, the following exposes the application's metrics under `/metrics` and sets up a `prometheus_client.Counter` for inbound HTTP requests. It also uses *Klein* to set up the routes instead of relying directly on `twisted.web` for routing.

```
from prometheus_client.twisted import MetricsResource
from twisted.web.server import Site
from twisted.internet import reactor

from klein import Klein

from prometheus_client import Counter


INBOUND_REQUESTS = Counter(
   "inbound_requests_total",
   "Counter (int) of inbound http requests",
   ["endpoint", "method"]
)

app = Klein()

@app.route("/metrics")
def metrics(request):
    INBOUND_REQUESTS.labels("/metrics", "GET").inc()
    return MetricsResource()


factory = Site(app.resource())
reactor.listenTCP(8000, factory)
reactor.run()
```

CHAPTER

TWO

PROJECT INFORMATION

2.1 Changelog

All notable changes to this project will be documented in this file.

The format is based on *Keep a Changelog* and this project adheres to *Calendar Versioning*.

The **first number** of the version is the year. The **second number** is incremented with each release, starting at 1 for each year. The **third number** is when we need to start branches for older releases (only for emergencies).

prometheus-async has a very strong backwards-compatibility policy. Generally speaking, you shouldn't ever be afraid of updating.

Whenever breaking changes are needed, they are:

1. …announced here in the changelog.
2. …the old behavior raises a `DeprecationWarning` for a year (if possible).
3. …are done with another announcement in the changelog.

2.1.1 22.2.0 - 2022-05-14

Deprecated

- The `prometheus_async.types.IncDecrementer` `Protocol` is deprecated and will be removed in a year. It was never a public API. #29

Changed

- Due to improvements of `prometheus_client`'s type hints, we don't block them from Mypy anymore.

Fixed

- The type hints for `prometheus_async.track_inprogress()` now accept `prometheus_client.Gauge`s. #29

2.1.2 22.1.0 - 2022-02-15

Removed

- Support for Python 2.7, 3.5, and 3.6 has been dropped.
- The *loop* argument has been removed from `prometheus_async.aio.start_http_server()`.

Added

- Added type hints for all APIs. #21
- Added support for OpenMetrics exposition in `prometheus_async.aio.web.server_stats()` and thus `prometheus_async.aio.web.start_http_server_in_thread()`. #23

2.1.3 19.2.0 - 2019-01-17

Fixed

- Revert the switch to decorator.py since it turned out to be a very breaking change. Please note that the now-current release of *wrapt* 1.11.0 has a memory leak so you should block it in your lockfile.

  Sorry for the inconvenience this has caused!

2.1.4 19.1.0 - 2019-01-15

Changed

- Dropped most dependencies and switched to *decorator.py* to avoid a C dependency (*wrapt*) that produces functions that can't be pickled.

2.1.5 18.4.0 - 2018-12-07

Removed

- *prometheus_client* 0.0.18 or newer is now required.

Fixed

- Restored compatibility with *prometheus_client* 0.5.

2.1.6 18.3.0 - 2018-06-21

Fixed

- The HTTP access log when using `prometheus_async.start_http_server()` is disabled now. It was activated accidentally when moving to *aiohttp*'s application runner APIs.

2.1.7 18.2.0 - 2018-05-29

Deprecated

- Passing a *loop* argument to `prometheus_async.aio.start_http_server()` is a no-op and raises a `DeprecationWarning` now.

Changed

- Port to *aiohttp*'s application runner APIs to avoid those pesky deprecation warnings. As a consequence, the *loop* argument has been removed from internal APIs and became a no-op in public APIs.

2.1.8 18.1.0 - 2018-02-15

Removed

- Python 3.4 is no longer supported.
- *aiohttp* 3.0 or later is now required for aio metrics exposure.

Changed

- *python-consul* is no longer required for asyncio Consul service discovery. A plain *aiohttp* is enough now.

2.1.9 17.5.0 - 2017-10-30

Removed

- `prometheus_async.aio.web` now requires *aiohttp* 2.0 or later.

Added

- The thread created by `prometheus_async.aio.start_http_server_in_thread()` has a human-readable name now.

Fixed

- Fixed compatibility with *aiohttp* 2.3.

2.1.10 17.4.0 - 2017-08-14

Fixed

- Set proper content type header for the root redirection page.

2.1.11 17.3.0 - 2017-06-01

Fixed

- `prometheus_async.aio.web.start_http_server()` now passes the *loop* argument to `aiohttp.web.Application.make_handler()` instead of `Application`'s initializer. This fixes a "loop argument is deprecated" warning.

2.1.12 17.2.0 - 2017-03-21

Deprecated

- Using *aiohttp* older than 0.21 is now deprecated.

Fixed

- `prometheus_async.aio.web` now supports *aiohttp* 2.0.

2.1.13 17.1.0 - 2017-01-14

Fixed

- Fix monotonic timer on Python 2. #7

2.1.14 16.2.0 - 2016-10-28

Changed

- When using the *aiohttp* metrics exporter, create the web application using an explicit loop argument. #6

2.1.15 16.1.0 - 2016-09-23

Changed

- Service discovery deregistration is optional now.

2.1.16 16.0.0 - 2016-05-19

Added

- Initial release.

2.2 License and Credits

prometheus-async is licensed under the *Apache License 2*. The full license text can be also found in the source code repository.

2.2.1 Credits

prometheus-async is written and maintained by Hynek Schlawack.

The development is kindly supported by Variomedia AG.

Other contributors can be found in GitHub's overview.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search

INDEX