prometheus-async

Release UNRELEASED

unknown

Apr 11, 2024

CONTENTS

Release **UNRELEASED** (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 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 -Im pip install -U pip
$ python -Im pip install prometheus-async
```

If you want to expose metrics using *aiohttp*:

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

If you want to instrument a Twisted application:

```
$ python -Im pip install -U pip
$ python -Im 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.

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

`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")
```

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

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.

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

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`:

async 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

Important: 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**.

As of 2023, the uWSGI project declared to only do emergency maintenance, therefore it's a good idea to migrate away from it anyway.

---

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?

async 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

CREDITS

prometheus-async is written and maintained by Hynek Schlawack.

The development is kindly supported by my employer Variomedia AG, *prometheus-async*'s Tidelift subscribers, and all my amazing GitHub Sponsors.

CHAPTER

THREE

PROMETHEUS-ASYNC FOR ENTERPRISE

Available as part of the Tidelift Subscription.

The maintainers of *prometheus-async* and thousands of other packages are working with Tidelift to deliver commercial support and maintenance for the open source packages you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact packages you use. Learn more.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search

INDEX