OpenTelemetry Python

OpenTelemetry Authors

May 31, 2024

CORE PACKAGES

Welcome to the docs for the Python OpenTelemetry implementation.

For an introduction to OpenTelemetry, see the OpenTelemetry website docs.

To learn how to instrument your Python code, see Getting Started. For project status, information about releases, installation instructions and more, see Python.

CHAPTER

ONE

GETTING STARTED

- Getting Started
- Frequently Asked Questions and Cookbook

1.1 OpenTelemetry Python API

1.1.1 opentelemetry._logs package

Submodules

opentelemetry._logs.severity

Module contents

1.1.2 opentelemetry.baggage package

Subpackages

opentelemetry.baggage.propagation package

Module contents

Module contents

1.1.3 opentelemetry.context package

Submodules

opentelemetry.context.base_context module

Module contents

1.1.4 opentelemetry.propagate package

Module contents

1.1.5 opentelemetry.propagators package

Subpackages

opentelemetry.propagators.textmap

Module contents

opentelemetry.propagators.composite

Module contents

1.1.6 opentelemetry.trace package

Submodules

opentelemetry.trace.status

opentelemetry.trace.span

Module contents

1.1.7 opentelemetry.metrics package

Module contents

1.1.8 opentelemetry.environment_variables package

Module contents

1.2 OpenTelemetry Python SDK

1.2.1 opentelemetry.sdk._logs package

1.2.2 opentelemetry.sdk.resources package

1.2.3 opentelemetry.sdk.trace package

Submodules

opentelemetry.sdk.trace.export

opentelemetry.sdk.trace.id_generator

opentelemetry.sdk.trace.sampling

opentelemetry.sdk.util.instrumentation

1.2.4 opentelemetry.sdk.metrics package

Submodules

opentelemetry.sdk.metrics.export

opentelemetry.sdk.metrics.view

1.2.5 opentelemetry.sdk.error_handler package

1.2.6 opentelemetry.sdk.environment_variables

1.3 Exporters

1.3.1 OpenCensus Exporter

The **OpenCensus Exporter** allows to export traces using OpenCensus.

1.3.2 OpenTelemetry OTLP Exporters

This library allows to export tracing data to an OTLP collector.

Usage

The **OTLP Span Exporter** allows to export OpenTelemetry traces to the OTLP collector.

You can configure the exporter with the following environment variables:

- `OTEL_EXPORTER_OTLP_TRACES_TIMEOUT`
- `OTEL_EXPORTER_OTLP_TRACES_PROTOCOL`
- `OTEL_EXPORTER_OTLP_TRACES_HEADERS`
- `OTEL_EXPORTER_OTLP_TRACES_ENDPOINT`
- `OTEL_EXPORTER_OTLP_TRACES_COMPRESSION`
- `OTEL_EXPORTER_OTLP_TRACES_CERTIFICATE`
- `OTEL_EXPORTER_OTLP_TIMEOUT`
- `OTEL_EXPORTER_OTLP_PROTOCOL`
- `OTEL_EXPORTER_OTLP_HEADERS`
- `OTEL_EXPORTER_OTLP_ENDPOINT`
- `OTEL_EXPORTER_OTLP_COMPRESSION`
- `OTEL_EXPORTER_OTLP_CERTIFICATE`

```
from opentelemetry import trace
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

# Resource can be required for some backends, e.g. Jaeger
# If resource wouldn't be set - traces wouldn't appears in Jaeger
resource = Resource(attributes={
    "service.name": "service"
})

trace.set_tracer_provider(TracerProvider(resource=resource))
tracer = trace.get_tracer(__name__)

otlp_exporter = OTLPSpanExporter(endpoint="http://localhost:4317", insecure=True)

span_processor = BatchSpanProcessor(otlp_exporter)

trace.get_tracer_provider().add_span_processor(span_processor)

with tracer.start_as_current_span("foo"):
    print("Hello world!")
```

API

1.3.3 OpenTelemetry Zipkin Exporters

1.4 Shims

1.4.1 OpenCensus Shim for OpenTelemetry

1.4.2 OpenTracing Shim for OpenTelemetry

1.5 Examples

1.5.1 Auto-instrumentation

To learn about automatic instrumentation and how to run the example in this directory, see Automatic Instrumentation.

1.5.2 Basic Context

These examples show how context is propagated through Spans in OpenTelemetry. There are three different examples:

- implicit_context: Shows how starting a span implicitly creates context.
- child_context: Shows how context is propagated through child spans.
- async_context: Shows how context can be shared in another coroutine.

The source files of these examples are available here.

Installation

```
pip install opentelemetry-api
pip install opentelemetry-sdk
```

Run the Example

```
python <example_name>.py
```

The output will be shown in the console.

Useful links

- OpenTelemetry
- *opentelemetry.trace package*

1.5.3 Basic Trace

These examples show how to use OpenTelemetry to create and export Spans. There are two different examples:

- basic_trace: Shows how to configure a SpanProcessor and Exporter, and how to create a tracer and span.
- resources: Shows how to add resource information to a Provider.

The source files of these examples are available here.

Installation

```
pip install opentelemetry-api
pip install opentelemetry-sdk
```

Run the Example

```
python <example_name>.py
```

The output will be shown in the console.

Useful links

- OpenTelemetry
- *opentelemetry.trace package*

1.5.4 Django Instrumentation

This shows how to use `opentelemetry-instrumentation-django` to automatically instrument a Django app.

For more user convenience, a Django app is already provided in this directory.

Preparation

This example will be executed in a separate virtual environment:

```
$ mkdir django_auto_instrumentation
$ virtualenv django_auto_instrumentation
$ source django_auto_instrumentation/bin/activate
```

Installation

```
$ pip install opentelemetry-sdk
$ pip install opentelemetry-instrumentation-django
$ pip install requests
```

Execution

Execution of the Django app

This example uses Django features intended for development environment. The `runserver` option should not be used for production environments.

Set these environment variables first:

1. `export DJANGO_SETTINGS_MODULE=instrumentation_example.settings`

The way to achieve OpenTelemetry instrumentation for your Django app is to use an `opentelemetry.instrumentation.django.DjangoInstrumentor` to instrument the app.

Clone the `opentelemetry-python` repository and go to `opentelemetry-python/docs/examples/django`.

Once there, open the `manage.py` file. The call to `DjangoInstrumentor().instrument()` in `main` is all that is needed to make the app be instrumented.

Run the Django app with `python manage.py runserver --noreload`. The `--noreload` flag is needed to avoid Django from running `main` twice.

Execution of the client

Open up a new console and activate the previous virtual environment there too:

`source django_auto_instrumentation/bin/activate`

Go to `opentelemetry-python/docs/examples/django`, once there run the client with:

`python client.py hello`

Go to the previous console, where the Django app is running. You should see output similar to this one:


```
{
    "name": "home_page_view",
    "context": {
        "trace_id": "0xed88755c56d95d05a506f5f70e7849b9",
        "span_id": "0x0a94c7a60e0650d5",
        "trace_state": "{}"
    },
    "kind": "SpanKind.SERVER",
    "parent_id": "0x3096ef92e621c22d",
    "start_time": "2020-04-26T01:49:57.205833Z",
    "end_time": "2020-04-26T01:49:57.206214Z",
    "status": {
        "status_code": "OK"
    },
    "attributes": {
        "http.request.method": "GET",
        "server.address": "localhost",
        "url.scheme": "http",
        "server.port": 8000,
        "url.full": "http://localhost:8000/?param=hello",
        "server.socket.address": "127.0.0.1",
        "network.protocol.version": "1.1",
        "http.response.status_code": 200
    },
    "events": [],
    "links": []
}
```


The last output shows spans automatically generated by the OpenTelemetry Django Instrumentation package.

Disabling Django Instrumentation

Django's instrumentation can be disabled by setting the following environment variable:

```
export OTEL_PYTHON_DJANGO_INSTRUMENT=False
```

Auto Instrumentation

This same example can be run using auto instrumentation. Comment out the call to `DjangoInstrumentor().instrument()` in `main`, then Run the django app with `opentelemetry-instrument python manage.py runserver --noreload`. Repeat the steps with the client, the result should be the same.

Usage with Auto Instrumentation and uWSGI

uWSGI and Django can be used together with auto instrumentation. To do so, first install uWSGI in the previous virtual environment:

```
pip install uwsgi
```

Once that is done, run the server with `uwsgi` from the directory that contains `instrumentation_example`:

```
opentelemetry-instrument uwsgi --http :8000 --module instrumentation_example.wsgi
```

This should start one uWSGI worker in your console. Open up a browser and point it to `localhost:8000`. This request should display a span exported in the server console.

References

- Django
- OpenTelemetry Project
- OpenTelemetry Django extension

1.5.5 Global Error Handler

Overview

This example shows how to use the global error handler.

Preparation

This example will be executed in a separate virtual environment:

```
$ mkdir global_error_handler
$ virtualenv global_error_handler
$ source global_error_handler/bin/activate
```

Installation

Here we install first `opentelemetry-sdk`, the only dependency. Afterwards, 2 error handlers are installed: `error_handler_0` will handle `ZeroDivisionError` exceptions, `error_handler_1` will handle `IndexError` and `KeyError` exceptions.

```
$ pip install opentelemetry-sdk
$ git clone https://github.com/open-telemetry/opentelemetry-python.git
$ pip install -e opentelemetry-python/docs/examples/error_handler/error_handler_0
$ pip install -e opentelemetry-python/docs/examples/error_handler/error_handler_1
```

Execution

An example is provided in the `opentelemetry-python/docs/examples/error_handler/example.py`.

You can just run it, you should get output similar to this one:

```
ErrorHandler0 handling a ZeroDivisionError
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    1 / 0
ZeroDivisionError: division by zero

ErrorHandler1 handling an IndexError
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    [1][2]
IndexError: list index out of range

ErrorHandler1 handling a KeyError
Traceback (most recent call last):
  File "test.py", line 17, in <module>
    {1: 2}[2]
KeyError: 2

Error handled by default error handler:
Traceback (most recent call last):
  File "test.py", line 23, in <module>
    assert False
AssertionError

No error raised
```

The `opentelemetry-sdk.error_handler` module includes documentation that explains how this works. We recommend you read it also, here is just a small summary.

In `example.py` we use `GlobalErrorHandler` as a context manager in several places, for example:

```
with GlobalErrorHandler():
    {1: 2}[2]
```

Running that code will raise a `KeyError` exception. `GlobalErrorHandler` will "capture" that exception and pass it down to the registered error handlers. If there is one that handles `KeyError` exceptions then it will handle it. That can be seen in the result of the execution of `example.py`:

```
ErrorHandler1 handling a KeyError
Traceback (most recent call last):
  File "test.py", line 17, in <module>
    {1: 2}[2]
KeyError: 2
```

There is no registered error handler that can handle `AssertionError` exceptions so this kind of errors are handled by the default error handler which just logs the exception to standard logging, as seen here:

```
Error handled by default error handler:
Traceback (most recent call last):
  File "test.py", line 23, in <module>
    assert False
AssertionError
```

When no exception is raised, the code inside the scope of `GlobalErrorHandler` is executed normally:

```
No error raised
```

Users can create Python packages that provide their own custom error handlers and install them in their virtual environments before running their code which instantiates `GlobalErrorHandler` context managers. `error_handler_0` and `error_handler_1` can be used as examples to create these custom error handlers.

In order for the error handlers to be registered, they need to create a class that inherits from `opentelemetry.sdk.error_handler.ErrorHandler` and at least one `Exception`-type class. For example, this is an error handler that handles `ZeroDivisionError` exceptions:

```
from opentelemetry.sdk.error_handler import ErrorHandler
from logging import getLogger

logger = getLogger(__name__)


class ErrorHandler0(ErrorHandler, ZeroDivisionError):

    def handle(self, error: Exception, *args, **kwargs):

        logger.exception("ErrorHandler0 handling a ZeroDivisionError")
```

To register this error handler, use the `opentelemetry_error_handler` entry point in the setup of the error handler package:

```
[options.entry_points]
opentelemetry_error_handler =
    error_handler_0 = error_handler_0:ErrorHandler0
```

This entry point should point to the error handler class, `ErrorHandler0` in this case.

1.5.6 Working With Fork Process Models

The `BatchSpanProcessor` is not fork-safe and doesn't work well with application servers (Gunicorn, uWSGI) which are based on the pre-fork web server model. The `BatchSpanProcessor` spawns a thread to run in the background to export spans to the telemetry backend. During the fork, the child process inherits the lock which is held by the parent process and deadlock occurs. We can use fork hooks to get around this limitation of the span processor.

Please see http://bugs.python.org/issue6721 for the problems about Python locks in (multi)threaded context with fork.

Gunicorn post_fork hook

```
from opentelemetry import trace
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor


def post_fork(server, worker):
    server.log.info("Worker spawned (pid: %s)", worker.pid)

    resource = Resource.create(attributes={
        "service.name": "api-service"
    })

    trace.set_tracer_provider(TracerProvider(resource=resource))
    span_processor = BatchSpanProcessor(
        OTLPSpanExporter(endpoint="http://localhost:4317")
    )
    trace.get_tracer_provider().add_span_processor(span_processor)
```

uWSGI postfork decorator

```
from uwsgidecorators import postfork

from opentelemetry import trace
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor


@postfork
def init_tracing():
    resource = Resource.create(attributes={
        "service.name": "api-service"
    })

    trace.set_tracer_provider(TracerProvider(resource=resource))
    span_processor = BatchSpanProcessor(
        OTLPSpanExporter(endpoint="http://localhost:4317")
```

(continues on next page)

(continued from previous page)

```
    )
    trace.get_tracer_provider().add_span_processor(span_processor)
```

The source code for the examples with Flask app are available here.

1.5.7 OpenTelemetry Logs SDK

Warning: OpenTelemetry Python logs are in an experimental state. The APIs within `opentelemetry.sdk._logs` are subject to change in minor/patch releases and make no backward compatibility guarantees at this time.

Start the Collector locally to see data being exported. Write the following file:

```
# otel-collector-config.yaml
receivers:
  otlp:
    protocols:
      grpc:

processors:
  batch:

exporters:
  logging:
    verbosity: detailed

service:
    pipelines:
        logs:
            receivers: [otlp]
            processors: [batch]
            exporters: [logging]
```

Then start the Docker container:

```
docker run \
    -p 4317:4317 \
    -v $(pwd)/otel-collector-config.yaml:/etc/otelcol-contrib/config.yaml \
    otel/opentelemetry-collector-contrib:latest
```

```
$ python example.py
```

The resulting logs will appear in the output from the collector and look similar to this:

```
Resource SchemaURL:
Resource labels:
    -> telemetry.sdk.language: STRING(python)
    -> telemetry.sdk.name: STRING(opentelemetry)
    -> telemetry.sdk.version: STRING(1.8.0)
    -> service.name: STRING(shoppingcart)
    -> service.instance.id: STRING(instance-12)
```

(continues on next page)

(continued from previous page)

```
InstrumentationLibraryLogs #0
InstrumentationLibraryMetrics SchemaURL:
InstrumentationLibrary __main__ 0.1
LogRecord #0
Timestamp: 2022-01-13 20:37:03.998733056 +0000 UTC
Severity: WARNING
ShortName:
Body: Jail zesty vixen who grabbed pay from quack.
Trace ID:
Span ID:
Flags: 0
LogRecord #1
Timestamp: 2022-01-13 20:37:04.082757888 +0000 UTC
Severity: ERROR
ShortName:
Body: The five boxing wizards jump quickly.
Trace ID:
Span ID:
Flags: 0
LogRecord #2
Timestamp: 2022-01-13 20:37:04.082979072 +0000 UTC
Severity: ERROR
ShortName:
Body: Hyderabad, we have a major problem.
Trace ID: 63491217958f126f727622e41d4460f3
Span ID: d90c57d6e1ca4f6c
Flags: 1
```

1.5.8 OpenTelemetry Metrics SDK

Start the Collector locally to see data being exported. Write the following file:

```
# otel-collector-config.yaml
receivers:
    otlp:
        protocols:
            grpc:

exporters:
    logging:

processors:
    batch:

service:
    pipelines:
        metrics:
            receivers: [otlp]
            exporters: [logging]
```

Then start the Docker container:

```
docker run \
    -p 4317:4317 \
    -v $(pwd)/otel-collector-config.yaml:/etc/otel/config.yaml \
    otel/opentelemetry-collector-contrib:latest
```

```
$ python example.py
```

The resulting metrics will appear in the output from the collector and look similar to this:

```

```

TODO

1.5.9 MetricReader configuration scenarios

These examples show how to customize the metrics that are output by the SDK using configuration on metric readers. There are multiple examples:

- preferred_aggregation.py: Shows how to configure the preferred aggregation for metric instrument types.
- preferred_temporality.py: Shows how to configure the preferred temporality for metric instrument types.

The source files of these examples are available here.

Installation

```
pip install -r requirements.txt
```

Run the Example

```
python <example_name>.py
```

The output will be shown in the console.

Useful links

- OpenTelemetry
- *opentelemetry.metrics package*

1.5.10 View common scenarios

These examples show how to customize the metrics that are output by the SDK using Views. There are multiple examples:

- change_aggregation.py: Shows how to configure to change the default aggregation for an instrument.
- change_name.py: Shows how to change the name of a metric.
- limit_num_of_attrs.py: Shows how to limit the number of attributes that are output for a metric.
- drop_metrics_from_instrument.py: Shows how to drop measurements from an instrument.

The source files of these examples are available here.

Installation

```
pip install -r requirements.txt
```

Run the Example

```
python <example_name>.py
```

The output will be shown in the console.

Useful links

- OpenTelemetry
- *opentelemetry.metrics package*

1.5.11 OpenCensus Exporter

This example shows how to use the OpenCensus Exporter to export traces to the OpenTelemetry collector.

The source files of this example are available here.

Installation

```
pip install opentelemetry-api
pip install opentelemetry-sdk
pip install opentelemetry-exporter-opencensus
```

Run the Example

Before running the example, it's necessary to run the OpenTelemetry collector and Jaeger. The docker folder contains a `docker-compose` template with the configuration of those services.

```
pip install docker-compose
cd docker
docker-compose up
```

Now, the example can be executed:

```
python collector.py
```

The traces are available in the Jaeger UI at http://localhost:16686/.

Useful links

- OpenTelemetry
- OpenTelemetry Collector
- *opentelemetry.trace package*
- *OpenCensus Exporter*

1.5.12 OpenCensus Shim

This example shows how to use the *opentelemetry-opencensus-shim package* to interact with libraries instrumented with opencensus-python.

The source files required to run this example are available here.

Installation

Jaeger

Start Jaeger

```
docker run --rm \
    -p 6831:6831/udp \
    -p 6832:6832/udp \
    -p 16686:16686 \
    jaegertracing/all-in-one:1.13 \
    --log-level=debug
```

Python Dependencies

Install the Python dependencies in requirements.txt

```
pip install -r requirements.txt
```

Alternatively, you can install the Python dependencies separately:

```
pip install \
    opentelemetry-api \
    opentelemetry-sdk \
    opentelemetry-exporter-jaeger \
    opentelemetry-opencensus-shim \
    opentelemetry-instrumentation-sqlite3 \
    opencensus \
    opencensus-ext-flask \
    Flask
```

Run the Application

Start the application in a terminal.

```
flask --app app run -h 0.0.0.0
```

Point your browser to the address printed out (probably http://127.0.0.1:5000). Alternatively, just use curl to trigger a request:

```
curl http://127.0.0.1:5000
```

Jaeger UI

Open the Jaeger UI in your browser at http://localhost:16686 and view traces for the "opencensus-shim-example-flask" service. Click on a span named "span" in the scatter plot. You will see a span tree with the following structure:

- **span**
  - **query movies from db**
    - SELECT
  - build response html

The root span comes from OpenCensus Flask instrumentation. The children `query movies from db` and `build response html` come from the manual instrumentation using OpenTelemetry's `opentelemetry.trace.Tracer.start_as_current_span()`. Finally, the `SELECT` span is created by OpenTelemetry's SQLite3 instrumentation. Everything is exported to Jaeger using the OpenTelemetry exporter.

Useful links

- OpenTelemetry
- *OpenCensus Shim for OpenTelemetry*

1.5.13 OpenTracing Shim

This example shows how to use the *opentelemetry-opentracing-shim package* to interact with libraries instrumented with opentracing-python.

The included `rediscache` library creates spans via the OpenTracing Redis integration, redis_opentracing. Spans are exported via the Jaeger exporter, which is attached to the OpenTelemetry tracer.

The source files required to run this example are available here.

Installation

Jaeger

Start Jaeger

```
docker run --rm \
    -p 6831:6831/udp \
    -p 6832:6832/udp \
    -p 16686:16686 \
    jaegertracing/all-in-one:1.13 \
    --log-level=debug
```

Redis

Install Redis following the instructions.

Make sure that the Redis server is running by executing this:

```
redis-server
```

Python Dependencies

Install the Python dependencies in requirements.txt

```
pip install -r requirements.txt
```

Alternatively, you can install the Python dependencies separately:

```
pip install \
    opentelemetry-api \
    opentelemetry-sdk \
    opentelemetry-exporter-jaeger \
    opentelemetry-opentracing-shim \
    redis \
    redis_opentracing
```

Run the Application

The example script calculates a few Fibonacci numbers and stores the results in Redis. The script, the `rediscache` library, and the OpenTracing Redis integration all contribute spans to the trace.

To run the script:

```
python main.py
```

After running, you can view the generated trace in the Jaeger UI.

Jaeger UI

Open the Jaeger UI in your browser at http://localhost:16686 and view traces for the "OpenTracing Shim Example" service.

Each `main.py` run should generate a trace, and each trace should include multiple spans that represent calls to Redis.

Note that tags and logs (OpenTracing) and attributes and events (OpenTelemetry) from both tracing systems appear in the exported trace.

Useful links

- OpenTelemetry
- *OpenTracing Shim for OpenTelemetry*
- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX