OpenTelemetry Python

OpenTelemetry Authors

Sep 30, 2021

GETTING STARTED

The Python OpenTelemetry client.

This documentation describes the *opentelemetry-api*, *opentelemetry-sdk*, and several *integration packages*.

Please note that this library is currently in alpha, and shouldn't be used in production environments.

CHAPTER

ONE

INSTALLATION

The API and SDK packages are available on PyPI, and can installed via pip:

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

In addition, there are several extension packages which can be installed separately as:

```
pip install opentelemetry-ext-{integration}
```

The extension packages can be found in ext/ directory of the repository.

CHAPTER

TWO

EXTENSIONS

Visit OpenTelemetry Registry to find related projects like exporters, instrumentation libraries, tracer implementations, etc.

2.1 Installing Cutting Edge Packages

While the project is pre-1.0, there may be significant functionality that has not yet been released to PyPI. In that situation, you may want to install the packages directly from the repo. This can be done by cloning the repository and doing an editable install:

```
git clone https://github.com/open-telemetry/opentelemetry-python.git
cd opentelemetry-python
pip install -e ./opentelemetry-api
pip install -e ./opentelemetry-sdk
pip install -e ./ext/opentelemetry-ext-{integration}
```

2.1.1 Getting Started with OpenTelemetry Python

This guide will walk you through instrumenting a Python application with `opentelemetry-python`.

For more elaborate examples, see *Examples*.

Hello world: emitting a trace to your console

To get started, install both the opentelemetry API and SDK:

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

The API package provides the interfaces required by the application owner, as well as some helper logic to load implementations.

The SDK provides an implementation of those interfaces, designed to be generic and extensible enough that in many situations, the SDK will be sufficient.

Once installed, we can now utilize the packages to emit spans from your application. A span represents an action within your application that you want to instrument, such as an HTTP request or a database call. Once instrumented, the application owner can extract helpful information such as how long the action took, or add arbitrary attributes to the span that may provide more insight for debugging.

Here's an example of a script that emits a trace containing three named spans: "foo", "bar", and "baz":

```
# tracing.py
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    ConsoleSpanExporter,
    SimpleExportSpanProcessor,
)

trace.set_tracer_provider(TracerProvider())
trace.get_tracer_provider().add_span_processor(
    SimpleExportSpanProcessor(ConsoleSpanExporter())
)

tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("foo"):
    with tracer.start_as_current_span("bar"):
        with tracer.start_as_current_span("baz"):
            print("Hello world from OpenTelemetry Python!")
```

We can run it, and see the traces print to your console:

```
$ python tracing_example.py
{
    "name": "baz",
    "context": {
        "trace_id": "0xb51058883c02f880111c959f3aa786a2",
        "span_id": "0xb2fa4c39f5f35e13",
        "trace_state": "{}"
    },
    "kind": "SpanKind.INTERNAL",
    "parent_id": "0x77e577e6a8813bf4",
    "start_time": "2020-05-07T14:39:52.906272Z",
    "end_time": "2020-05-07T14:39:52.906343Z",
    "status": {
        "canonical_code": "OK"
    },
    "attributes": {},
    "events": [],
    "links": []
}
{
    "name": "bar",
    "context": {
        "trace_id": "0xb51058883c02f880111c959f3aa786a2",
        "span_id": "0x77e577e6a8813bf4",
        "trace_state": "{}"
    },
    "kind": "SpanKind.INTERNAL",
    "parent_id": "0x3791d950cc5140c5",
    "start_time": "2020-05-07T14:39:52.906230Z",
    "end_time": "2020-05-07T14:39:52.906601Z",
    "status": {
        "canonical_code": "OK"
    },
    "attributes": {},
    "events": [],
```

(continues on next page)

(continued from previous page)

```
    "links": []
}
{
    "name": "foo",
    "context": {
        "trace_id": "0xb51058883c02f880111c959f3aa786a2",
        "span_id": "0x3791d950cc5140c5",
        "trace_state": "{}"
    },
    "kind": "SpanKind.INTERNAL",
    "parent_id": null,
    "start_time": "2020-05-07T14:39:52.906157Z",
    "end_time": "2020-05-07T14:39:52.906743Z",
    "status": {
        "canonical_code": "OK"
    },
    "attributes": {},
    "events": [],
    "links": []
}
```

Each span typically represents a single operation or unit of work. Spans can be nested, and have a parent-child relationship with other spans. While a given span is active, newly-created spans will inherit the active span's trace ID, options, and other attributes of its context. A span without a parent is called the "root span", and a trace is comprised of one root span and its descendants.

In the example above, the OpenTelemetry Python library creates one trace containing three spans and prints it to STDOUT.

Configure exporters to emit spans elsewhere

The example above does emit information about all spans, but the output is a bit hard to read. In common cases, you would instead *export* this data to an application performance monitoring backend, to be visualized and queried. It is also common to aggregate span and trace information from multiple services into a single database, so that actions that require multiple services can still all be visualized together.

This concept is known as distributed tracing. One such distributed tracing backend is known as Jaeger.

The Jaeger project provides an all-in-one docker container that provides a UI, database, and consumer. Let's bring it up now:

```
docker run -p 16686:16686 -p 6831:6831/udp jaegertracing/all-in-one
```

This will start Jaeger on port 16686 locally, and expose Jaeger thrift agent on port 6831. You can visit it at http://localhost:16686.

With this backend up, your application will now need to export traces to this system. `opentelemetry-sdk` does not provide an exporter for Jaeger, but you can install that as a separate package:

```
pip install opentelemetry-ext-jaeger
```

Once installed, update your code to import the Jaeger exporter, and use that instead:

```
# jaeger_example.py
from opentelemetry import trace
from opentelemetry.ext import jaeger
```

(continues on next page)

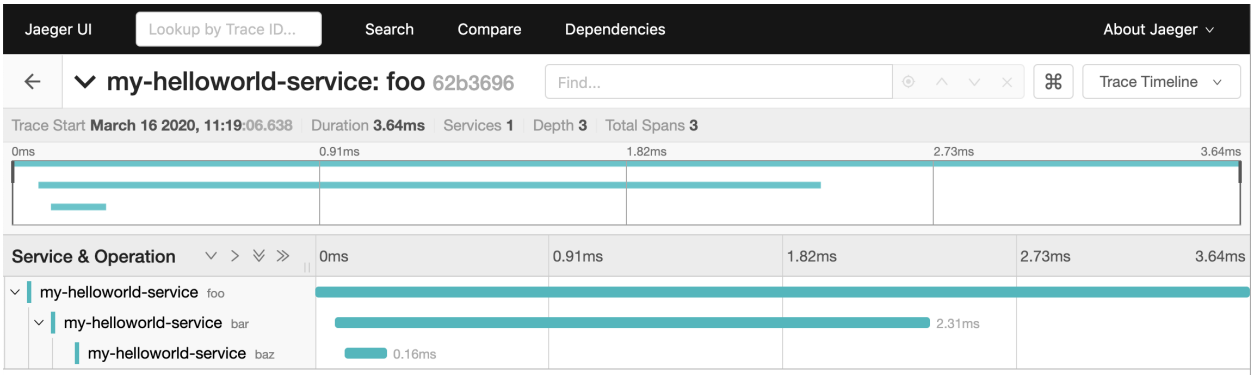
(continued from previous page)

```
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchExportSpanProcessor

trace.set_tracer_provider(TracerProvider())

jaeger_exporter = jaeger.JaegerSpanExporter(
    service_name="my-helloworld-service",
    agent_host_name="localhost",
    agent_port=6831,
)

trace.get_tracer_provider().add_span_processor(
    BatchExportSpanProcessor(jaeger_exporter)
)

tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("foo"):
    with tracer.start_as_current_span("bar"):
        with tracer.start_as_current_span("baz"):
            print("Hello world from OpenTelemetry Python!")
```

Run the script:

```
python jaeger_example.py
```

You can then visit the jaeger UI, see you service under “services”, and find your traces!



Integrations example with Flask

The above is a great example, but it’s very manual. Within the telemetry space, there are common actions that one wants to instrument:

- HTTP responses from web services
- HTTP requests from clients
- Database calls

To help instrument common scenarios, opentelemetry also has the concept of “instrumentations”: packages that are designed to interface with a specific framework or library, such as Flask and psycopg2. A list of the currently curated extension packages can be found here.

We will now instrument a basic Flask application that uses the requests library to send HTTP requests. First, install the instrumentation packages themselves:

```
pip install opentelemetry-ext-flask
pip install opentelemetry-ext-requests
```

And let's write a small Flask application that sends an HTTP request, activating each instrumentation during the initialization:

```
# flask_example.py
import flask
import requests

from opentelemetry import trace
from opentelemetry.ext.flask import FlaskInstrumentor
from opentelemetry.ext.requests import RequestsInstrumentor
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    ConsoleSpanExporter,
    SimpleExportSpanProcessor,
)

trace.set_tracer_provider(TracerProvider())
trace.get_tracer_provider().add_span_processor(
    SimpleExportSpanProcessor(ConsoleSpanExporter())
)

app = flask.Flask(__name__)
FlaskInstrumentor().instrument_app(app)
RequestsInstrumentor().instrument()


@app.route("/")
def hello():
    tracer = trace.get_tracer(__name__)
    with tracer.start_as_current_span("example-request"):
        requests.get("http://www.example.com")
    return "hello"


app.run(debug=True, port=5000)
```

Now run the above script, hit the root url (http://localhost:5000/) a few times, and watch your spans be emitted!

```
python flask_example.py
```

Adding Metrics

Spans are a great way to get detailed information about what your application is doing, but what about a more aggregated perspective? OpenTelemetry provides supports for metrics, a time series of numbers that might express things such as CPU utilization, request count for an HTTP server, or a business metric such as transactions.

All metrics can be annotated with labels: additional qualifiers that help describe what subdivision of the measurements the metric represents.

The following is an example of emitting metrics to console, in a similar fashion to the trace example:

```
# metrics.py
import time

from opentelemetry import metrics
from opentelemetry.sdk.metrics import Counter, MeterProvider
from opentelemetry.sdk.metrics.export import ConsoleMetricsExporter
from opentelemetry.sdk.metrics.export.controller import PushController

metrics.set_meter_provider(MeterProvider())
meter = metrics.get_meter(__name__, True)
exporter = ConsoleMetricsExporter()
controller = PushController(meter, exporter, 5)

staging_labels = {"environment": "staging"}

requests_counter = meter.create_metric(
    name="requests",
    description="number of requests",
    unit="1",
    value_type=int,
    metric_type=Counter,
    label_keys=("environment",),
)

requests_counter.add(25, staging_labels)
time.sleep(5)

requests_counter.add(20, staging_labels)
time.sleep(5)
```

The sleeps will cause the script to take a while, but running it should yield:

```
$ python metrics_example.py
ConsoleMetricsExporter(data="Counter(name="requests", description="number of requests
↪")", labels="(('environment', 'staging'),)", value=25)
ConsoleMetricsExporter(data="Counter(name="requests", description="number of requests
↪")", labels="(('environment', 'staging'),)", value=45)
```

Using Prometheus

Similar to traces, it is really valuable for metrics to have its own data store to help visualize and query the data. A common solution for this is Prometheus.

Let's start by bringing up a Prometheus instance ourselves, to scrape our application. Write the following configuration:

```
# /tmp/prometheus.yml
scrape_configs:
- job_name: 'my-app'
  scrape_interval: 5s
  static_configs:
  - targets: ['localhost:8000']
```

And start a docker container for it:

```
# --net=host will not work properly outside of Linux.
docker run --net=host -v /tmp/prometheus.yml:/etc/prometheus/prometheus.yml prom/
↪prometheus \
    --log.level=debug --config.file=/etc/prometheus/prometheus.yml
```

For our Python application, we will need to install an exporter specific to Prometheus:

```
pip install opentelemetry-ext-prometheus
```

And use that instead of the *ConsoleMetricsExporter*:

```
# prometheus.py
import sys
import time

from prometheus_client import start_http_server

from opentelemetry import metrics
from opentelemetry.ext.prometheus import PrometheusMetricsExporter
from opentelemetry.sdk.metrics import Counter, MeterProvider
from opentelemetry.sdk.metrics.export import ConsoleMetricsExporter
from opentelemetry.sdk.metrics.export.controller import PushController

# Start Prometheus client
start_http_server(port=8000, addr="localhost")

batcher_mode = "stateful"
metrics.set_meter_provider(MeterProvider())
meter = metrics.get_meter(__name__, batcher_mode == "stateful")
exporter = PrometheusMetricsExporter("MyAppPrefix")
controller = PushController(meter, exporter, 5)

staging_labels = {"environment": "staging"}

requests_counter = meter.create_metric(
    name="requests",
    description="number of requests",
    unit="1",
    value_type=int,
    metric_type=Counter,
    label_keys=("environment",),
)

requests_counter.add(25, staging_labels)
time.sleep(5)

requests_counter.add(20, staging_labels)
time.sleep(5)

# This line is added to keep the HTTP server up long enough to scrape.
input("Press any key to exit...")
```

The Prometheus server will run locally on port 8000, and the instrumented code will make metrics available to Prometheus via the *PrometheusMetricsExporter*. Visit the Prometheus UI (http://localhost:9090) to view your metrics.

Using the OpenTelemetry Collector for traces and metrics

Although it's possible to directly export your telemetry data to specific backends, you may more complex use cases, including:

- having a single telemetry sink shared by multiple services, to reduce overhead of switching exporters
- aggregating metrics or traces across multiple services, running on multiple hosts

To enable a broad range of aggregation strategies, OpenTelemetry provides the opentelemetry-collector. The Collector is a flexible application that can consume trace and metric data and export to multiple other backends, including to another instance of the Collector.

To see how this works in practice, let's start the Collector locally. Write the following file:

```
# /tmp/otel-collector-config.yaml
receivers:
    opencensus:
        endpoint: 0.0.0.0:55678
exporters:
    logging:
        loglevel: debug
processors:
    batch:
    queued_retry:
service:
    pipelines:
        traces:
            receivers: [opencensus]
            exporters: [logging]
            processors: [batch, queued_retry]
        metrics:
            receivers: [opencensus]
            exporters: [logging]
```

Start the docker container:

```
docker run -p 55678:55678 \
    -v /tmp/otel-collector-config.yaml:/etc/otel-collector-config.yaml \
    omnition/opentelemetry-collector-contrib:latest \
    --config=/etc/otel-collector-config.yaml
```

Install the OpenTelemetry Collector exporter:

```
pip install opentelemetry-ext-otcollector
```

And execute the following script:

```
# otcollector.py
import time

from opentelemetry import metrics, trace
from opentelemetry.ext.otcollector.metrics_exporter import (
    CollectorMetricsExporter,
)
from opentelemetry.ext.otcollector.trace_exporter import CollectorSpanExporter
from opentelemetry.sdk.metrics import Counter, MeterProvider
from opentelemetry.sdk.metrics.export.controller import PushController
from opentelemetry.sdk.trace import TracerProvider
```

(continues on next page)

(continued from previous page)

```
from opentelemetry.sdk.trace.export import BatchExportSpanProcessor

# create a CollectorSpanExporter
span_exporter = CollectorSpanExporter(
    # optional:
    # endpoint="myCollectorUrl:55678",
    # service_name="test_service",
    # host_name="machine/container name",
)
tracer_provider = TracerProvider()
trace.set_tracer_provider(tracer_provider)
span_processor = BatchExportSpanProcessor(span_exporter)
tracer_provider.add_span_processor(span_processor)

# create a CollectorMetricsExporter
metric_exporter = CollectorMetricsExporter(
    # optional:
    # endpoint="myCollectorUrl:55678",
    # service_name="test_service",
    # host_name="machine/container name",
)

# Meter is responsible for creating and recording metrics
metrics.set_meter_provider(MeterProvider())
meter = metrics.get_meter(__name__)
# controller collects metrics created from meter and exports it via the
# exporter every interval
controller = PushController(meter, metric_exporter, 5)

# Configure the tracer to use the collector exporter
tracer = trace.get_tracer_provider().get_tracer(__name__)

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

requests_counter = meter.create_metric(
    name="requests",
    description="number of requests",
    unit="1",
    value_type=int,
    metric_type=Counter,
    label_keys=("environment",),
)
# Labels are used to identify key-values that are associated with a specific
# metric that you want to record. These are useful for pre-aggregation and can
# be used to store custom dimensions pertaining to a metric
labels = {"environment": "staging"}
requests_counter.add(25, labels)
time.sleep(10)  # give push_controller time to push metrics
```

2.1.2 OpenTelemetry Python API

opentelemetry.configuration module

Module contents

Simple configuration manager

This is a configuration manager for OpenTelemetry. It reads configuration values from environment variables prefixed with `OPENTELEMETRY_PYTHON_` whose characters are only alphanumeric characters and unserscores, except for the first character after `OPENTELEMETRY_PYTHON_` which must not be a number.

For example, these environment variables will be read:

1. `OPENTELEMETRY_PYTHON_SOMETHING`
2. `OPENTELEMETRY_PYTHON_SOMETHING_ELSE_`
3. `OPENTELEMETRY_PYTHON_SOMETHING_ELSE_AND__ELSE`
4. `OPENTELEMETRY_PYTHON_SOMETHING_ELSE_AND_else`
5. `OPENTELEMETRY_PYTHON_SOMETHING_ELSE_AND_else2`

These won't:

1. `OPENTELEMETRY_PYTH_SOMETHING`
2. `OPENTELEMETRY_PYTHON_2_SOMETHING_AND__ELSE`
3. `OPENTELEMETRY_PYTHON_SOMETHING_%_ELSE`

The values stored in the environment variables can be found in an instance of `opentelemetry.configuration.Configuration`. This class can be instantiated freely because instantiating it returns always the same object.

For example, if the environment variable `OPENTELEMETRY_PYTHON_METER_PROVIDER` value is `my_meter_provider`, then `Configuration().meter_provider == "my_meter_provider"` would be `True`.

Non defined attributes will always return `None`. This is intended to make it easier to use the `Configuration` object in actual code, because it won't be necessary to check for the attribute to be defined first.

Environment variables used by OpenTelemetry

1. OPENTELEMETRY_PYTHON_METER_PROVIDER
2. OPENTELEMETRY_PYTHON_TRACER_PROVIDER

The value of these environment variables should be the name of the entry point that points to the class that implements either provider. This OpenTelemetry API package provides one entry point for each, which can be found in the setup.py file:

```
entry_points={
    ...
    "opentelemetry_meter_provider": [
        "default_meter_provider = "
        "opentelemetry.metrics:DefaultMeterProvider"
    ],
    "opentelemetry_tracer_provider": [
```

(continues on next page)

(continued from previous page)

```
        "default_tracer_provider = "
        "opentelemetry.trace:DefaultTracerProvider"
    ],
}
```

To use the meter provider above, then the `OPENTELEMETRY_PYTHON_METER_PROVIDER` should be set to `"default_meter_provider"` (this is not actually necessary since the OpenTelemetry API provided providers are the default ones used if no configuration is found in the environment variables).

Configuration values that are exactly `"True"` or `"False"` will be converted to its boolean values of `True` and `False` respectively.

Configuration values that can be casted to integers or floats will be casted.

This object can be used by any OpenTelemetry component, native or external. For that reason, the `Configuration` object is designed to be immutable. If a component would change the value of one of the `Configuration` object attributes then another component that relied on that value may break, leading to bugs that are very hard to debug. To avoid this situation, the preferred approach for components that need a different value than the one provided by the `Configuration` object is to implement a mechanism that allows the user to override this value instead of changing it.

class opentelemetry.configuration.**Configuration**
Bases: object

opentelemetry.context package

Submodules

opentelemetry.context.base_context module

class opentelemetry.context.context.**Context**
Bases: Dict[str, object]

class opentelemetry.context.context.**RuntimeContext**
Bases: abc.ABC

The RuntimeContext interface provides a wrapper for the different mechanisms that are used to propagate context in Python. Implementations can be made available via entry_points and selected through environment variables.

abstract attach(*context*)
Sets the current *Context* object. Returns a token that can be used to reset to the previous *Context*.

Parameters **context** (*Context*) – The Context to set.

Return type object

abstract get_current()
Returns the current *Context* object.

Return type *Context*

abstract detach(*token*)
Resets Context to a previous value

Parameters **token** (object) – A reference to a previous Context.

Return type None

Module contents

opentelemetry.context.**get_value**(*key*, *context=None*)

To access the local state of a concern, the RuntimeContext API provides a function which takes a context and a key as input, and returns a value.

Parameters

- **key** (str) – The key of the value to retrieve.
- **context** (Optional[*Context*]) – The context from which to retrieve the value, if None, the current context is used.

Return type object

Returns The value associated with the key.

opentelemetry.context.**set_value**(*key*, *value*, *context=None*)

To record the local state of a cross-cutting concern, the RuntimeContext API provides a function which takes a context, a key, and a value as input, and returns an updated context which contains the new value.

Parameters

- **key** (str) – The key of the entry to set.
- **value** (object) – The value of the entry to set.
- **context** (Optional[*Context*]) – The context to copy, if None, the current context is used.

Return type *Context*

Returns A new *Context* containing the value set.

opentelemetry.context.**get_current**()

To access the context associated with program execution, the Context API provides a function which takes no arguments and returns a Context.

Return type *Context*

Returns The current *Context* object.

opentelemetry.context.**attach**(*context*)

Associates a Context with the caller's current execution unit. Returns a token that can be used to restore the previous Context.

Parameters **context** (*Context*) – The Context to set as current.

Return type object

Returns A token that can be used with *detach* to reset the context.

opentelemetry.context.**detach**(*token*)

Resets the Context associated with the caller's current execution unit to the value it had before attaching a specified Context.

Parameters **token** (object) – The Token that was returned by a previous call to attach a Context.

Return type None

opentelemetry.correlationcontext package

Subpackages

opentelemetry.correlationcontext.propagation package

Module contents

class opentelemetry.correlationcontext.propagation.**CorrelationContextPropagator**
Bases: opentelemetry.trace.propagation.httptextformat.HTTPTextFormat

MAX_HEADER_LENGTH = 8192

MAX_PAIR_LENGTH = 4096

MAX_PAIRS = 180

extract(*get_from_carrier*, *carrier*, *context=None*)
Extract CorrelationContext from the carrier.

See opentelemetry.trace.propagation.httptextformat.HTTPTextFormat.extract

Return type *Context*

inject(*set_in_carrier*, *carrier*, *context=None*)
Injects CorrelationContext into the carrier.

See opentelemetry.trace.propagation.httptextformat.HTTPTextFormat.inject

Return type None

Module contents

opentelemetry.correlationcontext.**get_correlations**(*context=None*)
Returns the name/value pairs in the CorrelationContext

Parameters **context** (Optional[*Context*]) – The Context to use. If not set, uses current Context

Return type Dict[str, object]

Returns Name/value pairs in the CorrelationContext

opentelemetry.correlationcontext.**get_correlation**(*name*, *context=None*)
Provides access to the value for a name/value pair in the CorrelationContext

Parameters

- **name** (str) – The name of the value to retrieve
- **context** (Optional[*Context*]) – The Context to use. If not set, uses current Context

Return type Optional[object]

Returns The value associated with the given name, or null if the given name is not present.

opentelemetry.correlationcontext.**set_correlation**(*name*, *value*, *context=None*)
Sets a value in the CorrelationContext

Parameters

- **name** (str) – The name of the value to set
- **value** (object) – The value to set
- **context** (Optional[*Context*]) – The Context to use. If not set, uses current Context

Return type *Context*

Returns A Context with the value updated

opentelemetry.correlationcontext.**remove_correlation**(*name*, *context=None*)
Removes a value from the CorrelationContext

Parameters

- **name** (str) – The name of the value to remove
- **context** (Optional[*Context*]) – The Context to use. If not set, uses current Context

Return type *Context*

Returns A Context with the name/value removed

opentelemetry.correlationcontext.**clear_correlations**(*context=None*)
Removes all values from the CorrelationContext

Parameters **context** (Optional[*Context*]) – The Context to use. If not set, uses current Context

Return type *Context*

Returns A Context with all correlations removed

opentelemetry.metrics package

Module contents

The OpenTelemetry metrics API describes the classes used to report raw measurements, as well as metrics with known aggregation and labels.

The *Meter* class is used to construct *Metric* s to record raw statistics as well as metrics with predefined aggregation.

See the metrics api spec for terminology and context clarification.

New in version 0.1.0.

Changed in version 0.5.0: meter_provider was replaced by *get_meter_provider*, set_preferred_meter_provider_implementation was replaced by *set_meter_provider*.

class opentelemetry.metrics.**DefaultBoundInstrument**
Bases: object

The default bound metric instrument.

Used when no bound instrument implementation is available.

add(*value*)
No-op implementation of *BoundCounter* add.

Parameters **value** (*~ValueT*) – The value to add to the bound metric instrument.

Return type None

record(*value*)
No-op implementation of *BoundMeasure* record.

Parameters **value** (*~ValueT*) – The value to record to the bound metric instrument.

Return type None

release()
No-op implementation of release.

Return type None

class opentelemetry.metrics.**BoundCounter**
Bases: object

add(*value*)
Increases the value of the bound counter by value.

Parameters **value** (*~ValueT*) – The value to add to the bound counter.

Return type None

class opentelemetry.metrics.**BoundMeasure**
Bases: object

record(*value*)
Records the given value to this bound measure.

Parameters **value** (*~ValueT*) – The value to record to the bound measure.

Return type None

class opentelemetry.metrics.**Metric**
Bases: abc.ABC

Base class for various types of metrics.

Metric class that inherit from this class are specialized with the type of bound metric instrument that the metric holds.

abstract bind(*labels*)
Gets a bound metric instrument.

Bound metric instruments are useful to reduce the cost of repeatedly recording a metric with a pre-defined set of label values. All metric kinds (counter, measure) support declaring a set of required label keys. The values corresponding to these keys should be specified in every bound metric instrument. "Unspecified" label values, in cases where a bound metric instrument is requested but a value was not provided are permitted.

Parameters **labels** (Dict[str, str]) – Labels to associate with the bound instrument.

Return type object

class opentelemetry.metrics.**DefaultMetric**
Bases: *opentelemetry.metrics.Metric*

The default Metric used when no Metric implementation is available.

bind(*labels*)
Gets a *DefaultBoundInstrument*.

Parameters **labels** (Dict[str, str]) – Labels to associate with the bound instrument.

Return type *DefaultBoundInstrument*

add(*value*, *labels*)
No-op implementation of *Counter* add.

Parameters

- **value** (*~ValueT*) – The value to add to the counter metric.
- **labels** (Dict[str, str]) – Labels to associate with the bound instrument.

Return type None

record(*value*, *labels*)
No-op implementation of *Measure* record.

Parameters

- **value** (*~ValueT*) – The value to record to this measure metric.
- **labels** (Dict[str, str]) – Labels to associate with the bound instrument.

Return type None

class opentelemetry.metrics.**Counter**
Bases: *opentelemetry.metrics.Metric*

A counter type metric that expresses the computation of a sum.

bind(*labels*)
Gets a *BoundCounter*.

Return type *BoundCounter*

add(*value*, *labels*)
Increases the value of the counter by value.

Parameters

- **value** (*~ValueT*) – The value to add to the counter metric.
- **labels** (Dict[str, str]) – Labels to associate with the bound instrument.

Return type None

class opentelemetry.metrics.**Measure**
Bases: *opentelemetry.metrics.Metric*

A measure type metric that represent raw stats that are recorded.

Measure metrics represent raw statistics that are recorded.

bind(*labels*)
Gets a *BoundMeasure*.

Return type *BoundMeasure*

record(*value*, *labels*)
Records the value to the measure.

Parameters

- **value** (*~ValueT*) – The value to record to this measure metric.
- **labels** (Dict[str, str]) – Labels to associate with the bound instrument.

Return type None

class opentelemetry.metrics.**Observer**
Bases: abc.ABC

An observer type metric instrument used to capture a current set of values.

Observer instruments are asynchronous, a callback is invoked with the observer instrument as argument allowing the user to capture multiple values per collection interval.

abstract observe(*value*, *labels*)
Captures value to the observer.

Parameters

- **value** (*~ValueT*) – The value to capture to this observer metric.
- **labels** (Dict[str, str]) – Labels associated to value.

Return type None

class opentelemetry.metrics.**DefaultObserver**
Bases: *opentelemetry.metrics.Observer*

No-op implementation of Observer.

observe(*value*, *labels*)
Captures value to the observer.

Parameters

- **value** (*~ValueT*) – The value to capture to this observer metric.
- **labels** (Dict[str, str]) – Labels associated to value.

Return type None

class opentelemetry.metrics.**MeterProvider**
Bases: abc.ABC

abstract get_meter(*instrumenting_module_name*, *stateful=True*, *instrumenting_library_version=''*)
Returns a *Meter* for use by the given instrumentation library.

This function may return different *Meter* types (e.g. a no-op meter vs. a functional meter).

Parameters

- **instrumenting_module_name** (str) – The name of the instrumenting module (usually just __name__).

  This should *not* be the name of the module that is instrumented but the name of the module doing the instrumentation. E.g., instead of "requests", use "opentelemetry.ext.requests".

- **stateful** (bool) – True/False to indicate whether the meter will be stateful. True indicates the meter computes checkpoints from over the process lifetime. False indicates the meter computes checkpoints which describe the updates of a single collection period (deltas).

- **instrumenting_library_version** (str) – Optional. The version string of the instrumenting library. Usually this should be the same as pkg_resources.get_distribution(instrumenting_library_name).version.

Return type *Meter*

class opentelemetry.metrics.**DefaultMeterProvider**
Bases: *opentelemetry.metrics.MeterProvider*

The default MeterProvider, used when no implementation is available.

All operations are no-op.

get_meter(*instrumenting_module_name*, *stateful=True*, *instrumenting_library_version=''*)
Returns a *Meter* for use by the given instrumentation library.

This function may return different *Meter* types (e.g. a no-op meter vs. a functional meter).

Parameters

- **instrumenting_module_name** (str) – The name of the instrumenting module (usually just __name__).

  This should *not* be the name of the module that is instrumented but the name of the module doing the instrumentation. E.g., instead of "requests", use "opentelemetry.ext.requests".

- **stateful** (bool) – True/False to indicate whether the meter will be stateful. True indicates the meter computes checkpoints from over the process lifetime. False indicates the meter computes checkpoints which describe the updates of a single collection period (deltas).

- **instrumenting_library_version** (str) – Optional. The version string of the instrumenting library. Usually this should be the same as pkg_resources.get_distribution(instrumenting_library_name).version.

Return type *Meter*

class opentelemetry.metrics.**Meter**
Bases: abc.ABC

An interface to allow the recording of metrics.

Metric s are used for recording pre-defined aggregation (counter), or raw values (measure) in which the aggregation and labels for the exported metric are deferred.

abstract record_batch(*labels*, *record_tuples*)
Atomically records a batch of *Metric* and value pairs.

Allows the functionality of acting upon multiple metrics with a single API call. Implementations should find bound metric instruments that match the key-value pairs in the labels.

Parameters

- **labels** (Dict[str, str]) – Labels associated with all measurements in the batch.
- **record_tuples** (Sequence[Tuple[*Metric*, ~ValueT]]) – A sequence of pairs of *Metric* s and the corresponding value to record for that metric.

Return type None

abstract create_metric(*name*, *description*, *unit*, *value_type*, *metric_type*, *label_keys=()*, *enabled=True*)
Creates a metric_kind metric with type value_type.

Parameters

- **name** (str) – The name of the metric.
- **description** (str) – Human-readable description of the metric.

- **unit** (str) – Unit of the metric values following the UCUM convention (https://unitsofmeasure.org/ucum.html).
- **value_type** (Type[~ValueT]) – The type of values being recorded by the metric.
- **metric_type** (Type[~MetricT]) – The type of metric being created.
- **label_keys** (Sequence[str]) – The keys for the labels with dynamic values.
- **enabled** (bool) – Whether to report the metric by default.

Returns: A new `metric_type` metric with values of `value_type`.

Return type *Metric*

abstract register_observer(*callback*, *name*, *description*, *unit*, *value_type*, *label_keys=()*, *enabled=True*)

Registers an `Observer` metric instrument.

Parameters

- **callback** (Callable[[*Observer*], None]) – Callback invoked each collection interval with the observer as argument.
- **name** (str) – The name of the metric.
- **description** (str) – Human-readable description of the metric.
- **unit** (str) – Unit of the metric values following the UCUM convention (https://unitsofmeasure.org/ucum.html).
- **value_type** (Type[~ValueT]) – The type of values being recorded by the metric.
- **label_keys** (Sequence[str]) – The keys for the labels with dynamic values.
- **enabled** (bool) – Whether to report the metric by default.

Returns: A new `Observer` metric instrument.

Return type *Observer*

abstract unregister_observer(*observer*)

Unregisters an `Observer` metric instrument.

Parameters **observer** (*Observer*) – The observer to unregister.

Return type None

class opentelemetry.metrics.**DefaultMeter**

Bases: *opentelemetry.metrics.Meter*

The default Meter used when no Meter implementation is available.

record_batch(*labels*, *record_tuples*)

Atomically records a batch of *Metric* and value pairs.

Allows the functionality of acting upon multiple metrics with a single API call. Implementations should find bound metric instruments that match the key-value pairs in the labels.

Parameters

- **labels** (Dict[str, str]) – Labels associated with all measurements in the batch.
- **record_tuples** (Sequence[Tuple[*Metric*, ~ValueT]]) – A sequence of pairs of *Metric* s and the corresponding value to record for that metric.

Return type None

create_metric(*name*, *description*, *unit*, *value_type*, *metric_type*, *label_keys=()*, *enabled=True*)
Creates a `metric_kind` metric with type `value_type`.

Parameters

- **name** (`str`) – The name of the metric.
- **description** (`str`) – Human-readable description of the metric.
- **unit** (`str`) – Unit of the metric values following the UCUM convention (https://unitsofmeasure.org/ucum.html).
- **value_type** (`Type`[~ValueT]) – The type of values being recorded by the metric.
- **metric_type** (`Type`[~MetricT]) – The type of metric being created.
- **label_keys** (`Sequence`[`str`]) – The keys for the labels with dynamic values.
- **enabled** (`bool`) – Whether to report the metric by default.

Returns: A new `metric_type` metric with values of `value_type`.

Return type *Metric*

register_observer(*callback*, *name*, *description*, *unit*, *value_type*, *label_keys=()*, *enabled=True*)
Registers an `Observer` metric instrument.

Parameters

- **callback** (`Callable`[[*Observer*], `None`]) – Callback invoked each collection interval with the observer as argument.
- **name** (`str`) – The name of the metric.
- **description** (`str`) – Human-readable description of the metric.
- **unit** (`str`) – Unit of the metric values following the UCUM convention (https://unitsofmeasure.org/ucum.html).
- **value_type** (`Type`[~ValueT]) – The type of values being recorded by the metric.
- **label_keys** (`Sequence`[`str`]) – The keys for the labels with dynamic values.
- **enabled** (`bool`) – Whether to report the metric by default.

Returns: A new `Observer` metric instrument.

Return type *Observer*

unregister_observer(*observer*)
Unregisters an `Observer` metric instrument.

Parameters **observer** (*Observer*) – The observer to unregister.

Return type `None`

opentelemetry.metrics.**get_meter**(*instrumenting_module_name*, *stateful=True*, *instrumenting_library_version=''*)
Returns a *Meter* for use by the given instrumentation library. This function is a convenience wrapper for opentelemetry.metrics.get_meter_provider().get_meter

Return type *Meter*

opentelemetry.metrics.**set_meter_provider**(*meter_provider*)
Sets the current global *MeterProvider* object.

Return type `None`

opentelemetry.metrics.**get_meter_provider**()
Gets the current global *MeterProvider* object.

Return type *MeterProvider*

opentelemetry.trace package

Submodules

opentelemetry.trace.sampling

class opentelemetry.trace.sampling.**Decision**(*sampled=False*, *attributes=None*)
Bases: object

A sampling decision as applied to a newly-created Span.

Parameters

- **sampled** (bool) – Whether the *opentelemetry.trace.Span* should be sampled.
- **attributes** (Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]) – Attributes to add to the *opentelemetry.trace.Span*.

class opentelemetry.trace.sampling.**Sampler**
Bases: abc.ABC

abstract should_sample(*parent_context*, *trace_id*, *span_id*, *name*, *attributes=None*, *links=()*)

Return type *Decision*

class opentelemetry.trace.sampling.**StaticSampler**(*decision*)
Bases: *opentelemetry.trace.sampling.Sampler*

Sampler that always returns the same decision.

should_sample(*parent_context*, *trace_id*, *span_id*, *name*, *attributes=None*, *links=()*)

Return type *Decision*

class opentelemetry.trace.sampling.**ProbabilitySampler**(*rate*)
Bases: *opentelemetry.trace.sampling.Sampler*

TRACE_ID_LIMIT = 18446744073709551615

classmethod get_bound_for_rate(*rate*)

Return type int

property rate

Return type float

property bound

Return type int

should_sample(*parent_context*, *trace_id*, *span_id*, *name*, *attributes=None*, *links=()*)

Return type *Decision*

opentelemetry.trace.status

class opentelemetry.trace.status.**StatusCanonicalCode**(*value*)

Bases: enum.Enum

Represents the canonical set of status codes of a finished Span.

OK = 0

Not an error, returned on success.

CANCELLED = 1

The operation was cancelled, typically by the caller.

UNKNOWN = 2

Unknown error.

For example, this error may be returned when a Status value received from another address space belongs to an error space that is not known in this address space. Also errors raised by APIs that do not return enough error information may be converted to this error.

INVALID_ARGUMENT = 3

The client specified an invalid argument.

Note that this differs from FAILED_PRECONDITION. INVALID_ARGUMENT indicates arguments that are problematic regardless of the state of the system (e.g., a malformed file name).

DEADLINE_EXCEEDED = 4

The deadline expired before the operation could complete.

For operations that change the state of the system, this error may be returned even if the operation has completed successfully. For example, a successful response from a server could have been delayed long

NOT_FOUND = 5

Some requested entity (e.g., file or directory) was not found.

Note to server developers: if a request is denied for an entire class of users, such as gradual feature rollout or undocumented whitelist, NOT_FOUND may be used. If a request is denied for some users within a class of users, such as user-based access control, PERMISSION_DENIED must be used.

ALREADY_EXISTS = 6

The entity that a client attempted to create (e.g., file or directory) already exists.

PERMISSION_DENIED = 7

The caller does not have permission to execute the specified operation.

PERMISSION_DENIED must not be used for rejections caused by exhausting some resource (use RESOURCE_EXHAUSTED instead for those errors). PERMISSION_DENIED must not be used if the caller can not be identified (use UNAUTHENTICATED instead for those errors). This error code does not imply the request is valid or the requested entity exists or satisfies other pre-conditions.

RESOURCE_EXHAUSTED = 8

Some resource has been exhausted, perhaps a per-user quota, or perhaps the entire file system is out of space.

FAILED_PRECONDITION = 9

The operation was rejected because the system is not in a state required for the operation's execution.

For example, the directory to be deleted is non-empty, an rmdir operation is applied to a non-directory, etc. Service implementors can use the following guidelines to decide between FAILED_PRECONDITION, ABORTED, and UNAVAILABLE:

(a) Use UNAVAILABLE if the client can retry just the failing call.

(b) Use ABORTED if the client should retry at a higher level (e.g., when a client-specified test-and-set fails, indicating the client should restart a read-modify-write sequence).

(c) Use FAILED_PRECONDITION if the client should not retry until the system state has been explicitly fixed.

E.g., if an "rmdir" fails because the directory is non-empty, FAILED_PRECONDITION should be returned since the client should not retry unless the files are deleted from the directory.

ABORTED = 10

The operation was aborted, typically due to a concurrency issue such as a sequencer check failure or transaction abort.

See the guidelines above for deciding between FAILED_PRECONDITION, ABORTED, and UNAVAILABLE.

OUT_OF_RANGE = 11

The operation was attempted past the valid range.

E.g., seeking or reading past end-of-file. Unlike INVALID_ARGUMENT, this error indicates a problem that may be fixed if the system state changes. For example, a 32-bit file system will generate INVALID_ARGUMENT if asked to read at an offset that is not in the range [0,2^32-1],but it will generate OUT_OF_RANGE if asked to read from an offset past the current file size. There is a fair bit of overlap between FAILED_PRECONDITION and OUT_OF_RANGE. We recommend using OUT_OF_RANGE (the more specific error) when it applies so that callers who are iterating through a space can easily look for an OUT_OF_RANGE error to detect when they are done.

UNIMPLEMENTED = 12

The operation is not implemented or is not supported/enabled in this service.

INTERNAL = 13

Internal errors.

This means that some invariants expected by the underlying system have been broken. This error code is reserved for serious errors.

UNAVAILABLE = 14

The service is currently unavailable.

This is most likely a transient condition, which can be corrected by retrying with a backoff. Note that it is not always safe to retry non-idempotent operations.

DATA_LOSS = 15

Unrecoverable data loss or corruption.

UNAUTHENTICATED = 16

The request does not have valid authentication credentials for the operation.

class opentelemetry.trace.status.**Status**(*canonical_code=<StatusCanonicalCode.OK: 0>*, *description=None*)

Bases: object

Represents the status of a finished Span.

Parameters

- **canonical_code** (*StatusCanonicalCode*) – The canonical status code that describes the result status of the operation.
- **description** (Optional[str]) – An optional description of the status.

property canonical_code

Represents the canonical status code of a finished Span.

Return type *StatusCanonicalCode*

property description
Status description

Return type Optional[str]

property is_ok
Returns false if this represents an error, true otherwise.

Return type bool

Module contents

The OpenTelemetry tracing API describes the classes used to generate distributed traces.

The *Tracer* class controls access to the execution context, and manages span creation. Each operation in a trace is represented by a *Span*, which records the start, end time, and metadata associated with the operation.

This module provides abstract (i.e. unimplemented) classes required for tracing, and a concrete no-op *DefaultSpan* that allows applications to use the API package alone without a supporting implementation.

To get a tracer, you need to provide the package name from which you are calling the tracer APIs to OpenTelemetry by calling *TracerProvider.get_tracer* with the calling module name and the version of your package.

The tracer supports creating spans that are "attached" or "detached" from the context. New spans are "attached" to the context in that they are created as children of the currently active span, and the newly-created span can optionally become the new active span:

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

# Create a new root span, set it as the current span in context
with tracer.start_as_current_span("parent"):
    # Attach a new child and update the current span
    with tracer.start_as_current_span("child"):
        do_work():
    # Close child span, set parent as current
# Close parent span, set default span as current
```

When creating a span that's "detached" from the context the active span doesn't change, and the caller is responsible for managing the span's lifetime:

```
# Explicit parent span assignment
child = tracer.start_span("child", parent=parent)

try:
    do_work(span=child)
finally:
    child.end()
```

Applications should generally use a single global TracerProvider, and use either implicit or explicit context propagation consistently throughout.

New in version 0.1.0.

Changed in version 0.3.0: *TracerProvider* was introduced and the global `tracer` getter was replaced by `tracer_provider`.

Changed in version 0.5.0: `tracer_provider` was replaced by *get_tracer_provider*, `set_preferred_tracer_provider_implementation` was replaced by *set_tracer_provider*.

class opentelemetry.trace.**LinkBase**(*context*)
Bases: abc.ABC

property context

Return type *SpanContext*

abstract property attributes

Return type Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]

class opentelemetry.trace.**Link**(*context*, *attributes=None*)
Bases: *opentelemetry.trace.LinkBase*

A link to a *Span*.

Parameters

- **context** (*SpanContext*) – *SpanContext* of the *Span* to link to.
- **attributes** (Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]) – Link's attributes.

property attributes

Return type Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]

class opentelemetry.trace.**LazyLink**(*context*, *link_formatter*)
Bases: *opentelemetry.trace.LinkBase*

A lazy link to a *Span*.

Parameters

- **context** (*SpanContext*) – *SpanContext* of the *Span* to link to.
- **link_formatter** (Callable[[], Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]]) – Callable object that returns the attributes of the Link.

property attributes

Return type Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]

class opentelemetry.trace.**SpanKind**(*value*)
Bases: enum.Enum

Specifies additional details on how this span relates to its parent span.

Note that this enumeration is experimental and likely to change. See https://github.com/open-telemetry/opentelemetry-specification/pull/226.

INTERNAL = 0

SERVER = 1

CLIENT = 2
Indicates that the span describes a request to some remote service.

PRODUCER = 3
Indicates that the span describes a producer sending a message to a broker. Unlike client and server, there is usually no direct critical path latency relationship between producer and consumer spans.

CONSUMER = 4
Indicates that the span describes a consumer receiving a message from a broker. Unlike client and server, there is usually no direct critical path latency relationship between producer and consumer spans.

class opentelemetry.trace.**Span**
Bases: abc.ABC

A span represents a single operation within a trace.

abstract end(*end_time=None*)
Sets the current time as the span's end time.

The span's end time is the wall time at which the operation finished.

Only the first call to *end* should modify the span, and implementations are free to ignore or raise on further calls.

Return type None

abstract get_context()
Gets the span's SpanContext.

Get an immutable, serializable identifier for this span that can be used to create new child spans.

Return type *SpanContext*

Returns A *SpanContext* with a copy of this span's immutable state.

abstract set_attribute(*key*, *value*)
Sets an Attribute.

Sets a single Attribute with the key and value passed as arguments.

Return type None

abstract add_event(*name*, *attributes=None*, *timestamp=None*)
Adds an *Event*.

Adds a single *Event* with the name and, optionally, a timestamp and attributes passed as arguments. Implementations should generate a timestamp if the *timestamp* argument is omitted.

Return type None

abstract add_lazy_event(*name*, *event_formatter*, *timestamp=None*)
Adds an *Event*.

Adds a single *Event* with the name, an event formatter that calculates the attributes lazily and, optionally, a timestamp. Implementations should generate a timestamp if the *timestamp* argument is omitted.

Return type None

abstract update_name(*name*)
Updates the *Span* name.

This will override the name provided via *Tracer.start_span()*.

Upon this update, any sampling behavior based on Span name will depend on the implementation.

Return type None

abstract is_recording_events()
Returns whether this span will be recorded.

Returns true if this Span is active and recording information like events with the add_event operation and attributes using set_attribute.

Return type bool

abstract set_status(*status*)
Sets the Status of the Span. If used, this will override the default Span status, which is OK.

Return type None

class opentelemetry.trace.**TraceFlags**
Bases: int

A bitmask that represents options specific to the trace.

The only supported option is the "sampled" flag (0x01). If set, this flag indicates that the trace may have been sampled upstream.

See the W3C Trace Context - Traceparent spec for details.

DEFAULT = 0

SAMPLED = 1

classmethod get_default()

Return type *TraceFlags*

property sampled

Return type bool

class opentelemetry.trace.**TraceState**
Bases: Dict[str, str]

A list of key-value pairs representing vendor-specific trace info.

Keys and values are strings of up to 256 printable US-ASCII characters. Implementations should conform to the W3C Trace Context - Tracestate spec, which describes additional restrictions on valid field values.

classmethod get_default()

Return type *TraceState*

opentelemetry.trace.**format_trace_id**(*trace_id*)

Return type str

opentelemetry.trace.**format_span_id**(*span_id*)

Return type str

class opentelemetry.trace.**SpanContext**(*trace_id*, *span_id*, *is_remote*, *trace_flags=0*, *trace_state={}*)
Bases: object

The state of a Span to propagate between processes.

This class includes the immutable attributes of a *Span* that must be propagated to a span's children and across process boundaries.

Parameters

- **trace_id** (int) – The ID of the trace that this span belongs to.
- **span_id** (int) – This span's ID.

- **trace_flags** (*TraceFlags*) – Trace options to propagate.
- **trace_state** (*TraceState*) – Tracing-system-specific info to propagate.
- **is_remote** (bool) – True if propagated from a remote parent.

is_valid()
Get whether this *SpanContext* is valid.

A *SpanContext* is said to be invalid if its trace ID or span ID is invalid (i.e. 0).

Return type bool

Returns True if the *SpanContext* is valid, false otherwise.

class opentelemetry.trace.**DefaultSpan**(*context*)
Bases: *opentelemetry.trace.Span*

The default Span that is used when no Span implementation is available.

All operations are no-op except context propagation.

get_context()
Gets the span's SpanContext.

Get an immutable, serializable identifier for this span that can be used to create new child spans.

Return type *SpanContext*

Returns A *SpanContext* with a copy of this span's immutable state.

is_recording_events()
Returns whether this span will be recorded.

Returns true if this Span is active and recording information like events with the add_event operation and attributes using set_attribute.

Return type bool

end(*end_time=None*)
Sets the current time as the span's end time.

The span's end time is the wall time at which the operation finished.

Only the first call to *end* should modify the span, and implementations are free to ignore or raise on further calls.

Return type None

set_attribute(*key*, *value*)
Sets an Attribute.

Sets a single Attribute with the key and value passed as arguments.

Return type None

add_event(*name*, *attributes=None*, *timestamp=None*)
Adds an *Event*.

Adds a single *Event* with the name and, optionally, a timestamp and attributes passed as arguments. Implementations should generate a timestamp if the *timestamp* argument is omitted.

Return type None

add_lazy_event(*name*, *event_formatter*, *timestamp=None*)
Adds an *Event*.

Adds a single *Event* with the name, an event formatter that calculates the attributes lazily and, optionally, a timestamp. Implementations should generate a timestamp if the *timestamp* argument is omitted.

Return type None

update_name(*name*)
Updates the *Span* name.

This will override the name provided via *Tracer.start_span()*.

Upon this update, any sampling behavior based on Span name will depend on the implementation.

Return type None

set_status(*status*)
Sets the Status of the Span. If used, this will override the default Span status, which is OK.

Return type None

class opentelemetry.trace.**TracerProvider**
Bases: abc.ABC

abstract get_tracer(*instrumenting_module_name*, *instrumenting_library_version=''*)
Returns a *Tracer* for use by the given instrumentation library.

For any two calls it is undefined whether the same or different *Tracer* instances are returned, even for different library names.

This function may return different *Tracer* types (e.g. a no-op tracer vs. a functional tracer).

Parameters

- **instrumenting_module_name** (str) – The name of the instrumenting module (usually just __name__).

  This should *not* be the name of the module that is instrumented but the name of the module doing the instrumentation. E.g., instead of "requests", use "opentelemetry.ext.requests".

- **instrumenting_library_version** (str) – Optional. The version string of the instrumenting library. Usually this should be the same as pkg_resources.get_distribution(instrumenting_library_name).version.

Return type *Tracer*

class opentelemetry.trace.**DefaultTracerProvider**
Bases: *opentelemetry.trace.TracerProvider*

The default TracerProvider, used when no implementation is available.

All operations are no-op.

get_tracer(*instrumenting_module_name*, *instrumenting_library_version=''*)
Returns a *Tracer* for use by the given instrumentation library.

For any two calls it is undefined whether the same or different *Tracer* instances are returned, even for different library names.

This function may return different *Tracer* types (e.g. a no-op tracer vs. a functional tracer).

Parameters

- **instrumenting_module_name** (str) – The name of the instrumenting module (usually just __name__).

This should *not* be the name of the module that is instrumented but the name of the module doing the instrumentation. E.g., instead of `"requests"`, use `"opentelemetry.ext.requests"`.

- **instrumenting_library_version** (`str`) – Optional. The version string of the instrumenting library. Usually this should be the same as `pkg_resources.get_distribution(instrumenting_library_name).version`.

Return type *Tracer*

class opentelemetry.trace.**Tracer**

Bases: `abc.ABC`

Handles span creation and in-process context propagation.

This class provides methods for manipulating the context, creating spans, and controlling spans' lifecycles.

CURRENT_SPAN = <opentelemetry.trace.DefaultSpan object>

abstract get_current_span()

Gets the currently active span from the context.

If there is no current span, return a placeholder span with an invalid context.

Return type *Span*

Returns The currently active *Span*, or a placeholder span with an invalid *SpanContext*.

abstract start_span(*name*, *parent=<opentelemetry.trace.DefaultSpan object>*, *kind=<SpanKind.INTERNAL: 0>*, *attributes=None*, *links=()*, *start_time=None*, *set_status_on_exception=True*)

Starts a span.

Create a new span. Start the span without setting it as the current span in this tracer's context.

By default the current span will be used as parent, but an explicit parent can also be specified, either a *Span* or a *SpanContext*. If the specified value is `None`, the created span will be a root span.

The span can be used as context manager. On exiting, the span will be ended.

Example:

```
# tracer.get_current_span() will be used as the implicit parent.
# If none is found, the created span will be a root instance.
with tracer.start_span("one") as child:
    child.add_event("child's event")
```

Applications that need to set the newly created span as the current instance should use *start_as_current_span()* instead.

Parameters

- **name** (`str`) – The name of the span to be created.
- **parent** (`Union[Span, SpanContext, None]`) – The span's parent. Defaults to the current span.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.
- **attributes** (`Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]`) – The span's attributes.
- **links** (`Sequence[Link]`) – Links span to other spans

- **start_time** (Optional[int]) – Sets the start time of a span
- **set_status_on_exception** (bool) – Only relevant if the returned span is used in a with/context manager. Defines wether the span status will be automatically set to UNKNOWN when an uncaught exception is raised in the span with block. The span status won't be set by this mechanism if it was previousy set manually.

Return type *Span*

Returns The newly-created span.

abstract start_as_current_span(*name*, *parent=<opentelemetry.trace.DefaultSpan object>*, *kind=<SpanKind.INTERNAL: 0>*, *attributes=None*, *links=()*)

Context manager for creating a new span and set it as the current span in this tracer's context.

On exiting the context manager stops the span and set its parent as the current span.

Example:

```
with tracer.start_as_current_span("one") as parent:
    parent.add_event("parent's event")
    with tracer.start_as_current_span("two") as child:
        child.add_event("child's event")
        tracer.get_current_span()  # returns child
    tracer.get_current_span()      # returns parent
tracer.get_current_span()          # returns previously active span
```

This is a convenience method for creating spans attached to the tracer's context. Applications that need more control over the span lifetime should use *start_span()* instead. For example:

```
with tracer.start_as_current_span(name) as span:
    do_work()
```

is equivalent to:

```
span = tracer.start_span(name)
with tracer.use_span(span, end_on_exit=True):
    do_work()
```

Parameters

- **name** (str) – The name of the span to be created.
- **parent** (Union[*Span*, *SpanContext*, None]) – The span's parent. Defaults to the current span.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.
- **attributes** (Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]) – The span's attributes.
- **links** (Sequence[*Link*]) – Links span to other spans

Yields The newly-created span.

Return type Iterator[*Span*]

abstract use_span(*span*, *end_on_exit=False*)
Context manager for controlling a span's lifetime.

Set the given span as the current span in this tracer's context.

On exiting the context manager set the span that was previously active as the current span (this is usually but not necessarily the parent of the given span). If `end_on_exit` is `True`, then the span is also ended when exiting the context manager.

Parameters

- **span** (*Span*) – The span to start and make current.
- **end_on_exit** (`bool`) – Whether to end the span automatically when leaving the context manager.

Return type `Iterator[None]`

class opentelemetry.trace.**DefaultTracer**
Bases: *opentelemetry.trace.Tracer*

The default Tracer, used when no Tracer implementation is available.

All operations are no-op.

get_current_span()
Gets the currently active span from the context.

If there is no current span, return a placeholder span with an invalid context.

Return type *Span*

Returns The currently active *Span*, or a placeholder span with an invalid *SpanContext*.

start_span(*name*, *parent=<opentelemetry.trace.DefaultSpan object>*, *kind=<SpanKind.INTERNAL: 0>*, *attributes=None*, *links=()*, *start_time=None*, *set_status_on_exception=True*)
Starts a span.

Create a new span. Start the span without setting it as the current span in this tracer's context.

By default the current span will be used as parent, but an explicit parent can also be specified, either a *Span* or a *SpanContext*. If the specified value is `None`, the created span will be a root span.

The span can be used as context manager. On exiting, the span will be ended.

Example:

```
# tracer.get_current_span() will be used as the implicit parent.
# If none is found, the created span will be a root instance.
with tracer.start_span("one") as child:
    child.add_event("child's event")
```

Applications that need to set the newly created span as the current instance should use *start_as_current_span()* instead.

Parameters

- **name** (`str`) – The name of the span to be created.
- **parent** (`Union[`*Span*, *SpanContext*, `None]`) – The span's parent. Defaults to the current span.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.

- **attributes** (Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]) – The span's attributes.
- **links** (Sequence[*Link*]) – Links span to other spans
- **start_time** (Optional[int]) – Sets the start time of a span
- **set_status_on_exception** (bool) – Only relevant if the returned span is used in a with/context manager. Defines wether the span status will be automatically set to UNKNOWN when an uncaught exception is raised in the span with block. The span status won't be set by this mechanism if it was previousy set manually.

Return type *Span*

Returns The newly-created span.

start_as_current_span(*name*, *parent=<opentelemetry.trace.DefaultSpan object>*, *kind=<SpanKind.INTERNAL: 0>*, *attributes=None*, *links=()*)

Context manager for creating a new span and set it as the current span in this tracer's context.

On exiting the context manager stops the span and set its parent as the current span.

Example:

```
with tracer.start_as_current_span("one") as parent:
    parent.add_event("parent's event")
    with tracer.start_as_current_span("two") as child:
        child.add_event("child's event")
        tracer.get_current_span()  # returns child
    tracer.get_current_span()      # returns parent
tracer.get_current_span()          # returns previously active span
```

This is a convenience method for creating spans attached to the tracer's context. Applications that need more control over the span lifetime should use *start_span()* instead. For example:

```
with tracer.start_as_current_span(name) as span:
    do_work()
```

is equivalent to:

```
span = tracer.start_span(name)
with tracer.use_span(span, end_on_exit=True):
    do_work()
```

Parameters

- **name** (str) – The name of the span to be created.
- **parent** (Union[*Span*, *SpanContext*, None]) – The span's parent. Defaults to the current span.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.
- **attributes** (Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]) – The span's attributes.
- **links** (Sequence[*Link*]) – Links span to other spans

Yields The newly-created span.

Return type Iterator[*Span*]

use_span(*span*, *end_on_exit=False*)
Context manager for controlling a span's lifetime.

Set the given span as the current span in this tracer's context.

On exiting the context manager set the span that was previously active as the current span (this is usually but not necessarily the parent of the given span). If end_on_exit is True, then the span is also ended when exiting the context manager.

Parameters

- **span** (*Span*) – The span to start and make current.
- **end_on_exit** (bool) – Whether to end the span automatically when leaving the context manager.

Return type Iterator[None]

opentelemetry.trace.**get_tracer**(*instrumenting_module_name*, *instrumenting_library_version=''*, *tracer_provider=None*)
Returns a *Tracer* for use by the given instrumentation library.

This function is a convenience wrapper for opentelemetry.trace.TracerProvider.get_tracer.

If tracer_provider is ommited the current configured one is used.

Return type *Tracer*

opentelemetry.trace.**set_tracer_provider**(*tracer_provider*)
Sets the current global *TracerProvider* object.

Return type None

opentelemetry.trace.**get_tracer_provider**()
Gets the current global *TracerProvider* object.

Return type *TracerProvider*

2.1.3 OpenTelemetry Python SDK

opentelemetry.sdk.metrics package

Submodules

opentelemetry.sdk.metrics.export.aggregate

class opentelemetry.sdk.metrics.export.aggregate.**Aggregator**
Bases: abc.ABC

Base class for aggregators.

Aggregators are responsible for holding aggregated values and taking a snapshot of these values upon export (checkpoint).

abstract update(*value*)
Updates the current with the new value.

abstract take_checkpoint()
Stores a snapshot of the current value.

abstract merge(*other*)
Combines two aggregator values.

class opentelemetry.sdk.metrics.export.aggregate.**CounterAggregator**
Bases: *opentelemetry.sdk.metrics.export.aggregate.Aggregator*

Aggregator for Counter metrics.

update(*value*)
Updates the current with the new value.

take_checkpoint()
Stores a snapshot of the current value.

merge(*other*)
Combines two aggregator values.

class opentelemetry.sdk.metrics.export.aggregate.**MinMaxSumCountAggregator**
Bases: *opentelemetry.sdk.metrics.export.aggregate.Aggregator*

Agregator for Measure metrics that keeps min, max, sum and count.

update(*value*)
Updates the current with the new value.

take_checkpoint()
Stores a snapshot of the current value.

merge(*other*)
Combines two aggregator values.

class opentelemetry.sdk.metrics.export.aggregate.**ObserverAggregator**
Bases: *opentelemetry.sdk.metrics.export.aggregate.Aggregator*

Same as MinMaxSumCount but also with last value.

update(*value*)
Updates the current with the new value.

take_checkpoint()
Stores a snapshot of the current value.

merge(*other*)
Combines two aggregator values.

opentelemetry.sdk.metrics.export.aggregate.**get_latest_timestamp**(*time_stamp*, *other_timestamp*)

opentelemetry.sdk.metrics.export.batcher

opentelemetry.sdk.metrics.export

class opentelemetry.sdk.metrics.export.**MetricsExportResult**(*value*)
Bases: enum.Enum

An enumeration.

SUCCESS = 0

FAILURE = 1

class opentelemetry.sdk.metrics.export.**MetricRecord**(*aggregator*, *labels*, *metric*)
Bases: object

class opentelemetry.sdk.metrics.export.**MetricsExporter**
Bases: object

Interface for exporting metrics.

Interface to be implemented by services that want to export recorded metrics in its own format.

export(*metric_records*)
Exports a batch of telemetry data.

Parameters **metric_records** (Sequence[*MetricRecord*]) – A sequence of *MetricRecord* s. A *MetricRecord* contains the metric to be exported, the labels associated with that metric, as well as the aggregator used to export the current checkpointed value.

Return type *MetricsExportResult*

Returns The result of the export

shutdown()
Shuts down the exporter.

Called when the SDK is shut down.

Return type None

class opentelemetry.sdk.metrics.export.**ConsoleMetricsExporter**
Bases: *opentelemetry.sdk.metrics.export.MetricsExporter*

Implementation of *MetricsExporter* that prints metrics to the console.

This class can be used for diagnostic purposes. It prints the exported metrics to the console STDOUT.

export(*metric_records*)
Exports a batch of telemetry data.

Parameters **metric_records** (Sequence[*MetricRecord*]) – A sequence of *MetricRecord* s. A *MetricRecord* contains the metric to be exported, the labels associated with that metric, as well as the aggregator used to export the current checkpointed value.

Return type *MetricsExportResult*

Returns The result of the export

class opentelemetry.sdk.metrics.export.batcher.**Batcher**(*stateful*)
Bases: abc.ABC

Base class for all batcher types.

The batcher is responsible for storing the aggregators and aggregated values received from updates from metrics in the meter. The stored values will be sent to an exporter for exporting.

aggregator_for(*metric_type*)
Returns an aggregator based on metric type.

Aggregators keep track of and updates values when metrics get updated.

Return type *Aggregator*

checkpoint_set()
Returns a list of MetricRecords used for exporting.

The list of MetricRecords is a snapshot created from the current data in all of the aggregators in this batcher.

Return type Sequence[*MetricRecord*]

finished_collection()
Performs certain post-export logic.

For batchers that are stateless, resets the batch map.

abstract process(*record*)
Stores record information to be ready for exporting.

Depending on type of batcher, performs pre-export logic, such as filtering records based off of keys.

Return type None

class opentelemetry.sdk.metrics.export.batcher.**UngroupedBatcher**(*stateful*)
Bases: *opentelemetry.sdk.metrics.export.batcher.Batcher*

Accepts all records and passes them for exporting

process(*record*)
Stores record information to be ready for exporting.

Depending on type of batcher, performs pre-export logic, such as filtering records based off of keys.

opentelemetry.sdk.util.instrumentation

class opentelemetry.sdk.util.instrumentation.**InstrumentationInfo**(*name*, *version*)
Bases: object

Immutable information about an instrumentation library module.

See *opentelemetry.trace.TracerProvider.get_tracer* or *opentelemetry.metrics.MeterProvider.get_meter* for the meaning of these properties.

property version

Return type str

property name

Return type str

opentelemetry.sdk.metrics.**get_labels_as_key**(*labels*)
Gets a list of labels that can be used as a key in a dictionary.

Return type Tuple[Tuple[str, str]]

class opentelemetry.sdk.metrics.**BaseBoundInstrument**(*value_type*, *enabled*, *aggregator*)
Bases: object

Class containing common behavior for all bound metric instruments.

Bound metric instruments are responsible for operating on data for metric instruments for a specific set of labels.

Parameters

- **value_type** (Type[~ValueT]) – The type of values for this bound instrument (int, float).
- **enabled** (bool) – True if the originating instrument is enabled.
- **aggregator** (*Aggregator*) – The aggregator for this bound metric instrument. Will handle aggregation upon updates and checkpointing of values for exporting.

update(*value*)

release()

decrease_ref_count()

increase_ref_count()

ref_count()

class opentelemetry.sdk.metrics.**BoundCounter**(*value_type*, *enabled*, *aggregator*)

Bases: *opentelemetry.metrics.BoundCounter*, *opentelemetry.sdk.metrics.BaseBoundInstrument*

add(*value*)

See *opentelemetry.metrics.BoundCounter.add*.

Return type None

class opentelemetry.sdk.metrics.**BoundMeasure**(*value_type*, *enabled*, *aggregator*)

Bases: *opentelemetry.metrics.BoundMeasure*, *opentelemetry.sdk.metrics.BaseBoundInstrument*

record(*value*)

See *opentelemetry.metrics.BoundMeasure.record*.

Return type None

class opentelemetry.sdk.metrics.**Metric**(*name*, *description*, *unit*, *value_type*, *meter*, *label_keys=()*, *enabled=True*)

Bases: *opentelemetry.metrics.Metric*

Base class for all metric types.

Also known as metric instrument. This is the class that is used to represent a metric that is to be continuously recorded and tracked. Each metric has a set of bound metrics that are created from the metric. See *BaseBoundInstrument* for information on bound metric instruments.

BOUND_INSTR_TYPE

alias of *opentelemetry.sdk.metrics.BaseBoundInstrument*

bind(*labels*)

See *opentelemetry.metrics.Metric.bind*.

Return type *BaseBoundInstrument*

UPDATE_FUNCTION(*y*)

class opentelemetry.sdk.metrics.**Counter**(*name*, *description*, *unit*, *value_type*, *meter*, *label_keys=()*, *enabled=True*)

Bases: *opentelemetry.sdk.metrics.Metric*, *opentelemetry.metrics.Counter*

See *opentelemetry.metrics.Counter*.

BOUND_INSTR_TYPE

alias of *opentelemetry.sdk.metrics.BoundCounter*

add(*value*, *labels*)

See *opentelemetry.metrics.Counter.add*.

Return type None

UPDATE_FUNCTION(*value*, *labels*)

See *opentelemetry.metrics.Counter.add*.

Return type None

class opentelemetry.sdk.metrics.**Measure**(*name*, *description*, *unit*, *value_type*, *meter*, *label_keys=()*, *enabled=True*)

Bases: *opentelemetry.sdk.metrics.Metric*, *opentelemetry.metrics.Measure*

See *opentelemetry.metrics.Measure*.

BOUND_INSTR_TYPE

alias of *opentelemetry.sdk.metrics.BoundMeasure*

record(*value*, *labels*)

See *opentelemetry.metrics.Measure.record*.

Return type None

UPDATE_FUNCTION(*value*, *labels*)

See *opentelemetry.metrics.Measure.record*.

Return type None

class opentelemetry.sdk.metrics.**Observer**(*callback*, *name*, *description*, *unit*, *value_type*, *meter*, *label_keys=()*, *enabled=True*)

Bases: *opentelemetry.metrics.Observer*

See *opentelemetry.metrics.Observer*.

observe(*value*, *labels*)

Captures `value` to the observer.

Parameters

- **value** (*~ValueT*) – The value to capture to this observer metric.
- **labels** (`Dict[str, str]`) – Labels associated to `value`.

Return type None

run()

Return type `bool`

class opentelemetry.sdk.metrics.**Record**(*metric*, *labels*, *aggregator*)

Bases: `object`

Container class used for processing in the *Batcher*

class opentelemetry.sdk.metrics.**Meter**(*instrumentation_info*, *stateful*, *resource=<opentelemetry.sdk.resources.Resource object>*)

Bases: *opentelemetry.metrics.Meter*

See *opentelemetry.metrics.Meter*.

Parameters

- **instrumentation_info** (*InstrumentationInfo*) – The *InstrumentationInfo* for this meter.
- **stateful** (`bool`) – Indicates whether the meter is stateful.

collect()

Collects all the metrics created with this *Meter* for export.

Utilizes the batcher to create checkpoints of the current values in each aggregator belonging to the metrics that were created with this meter instance.

Return type None

record_batch(*labels*, *record_tuples*)
See *opentelemetry.metrics.Meter.record_batch*.

Return type None

create_metric(*name*, *description*, *unit*, *value_type*, *metric_type*, *label_keys=()*, *enabled=True*)
See *opentelemetry.metrics.Meter.create_metric*.

Return type ~MetricT

register_observer(*callback*, *name*, *description*, *unit*, *value_type*, *label_keys=()*, *enabled=True*)
Registers an Observer metric instrument.

Parameters

- **callback** (Callable[[*Observer*], None]) – Callback invoked each collection interval with the observer as argument.
- **name** (str) – The name of the metric.
- **description** (str) – Human-readable description of the metric.
- **unit** (str) – Unit of the metric values following the UCUM convention (https://unitsofmeasure.org/ucum.html).
- **value_type** (Type[~ValueT]) – The type of values being recorded by the metric.
- **label_keys** (Sequence[str]) – The keys for the labels with dynamic values.
- **enabled** (bool) – Whether to report the metric by default.

Returns: A new Observer metric instrument.

Return type *Observer*

unregister_observer(*observer*)
Unregisters an Observer metric instrument.

Parameters **observer** (*Observer*) – The observer to unregister.

Return type None

class opentelemetry.sdk.metrics.**MeterProvider**(*resource=<opentelemetry.sdk.resources.Resource object>*)
Bases: *opentelemetry.metrics.MeterProvider*

get_meter(*instrumenting_module_name*, *stateful=True*, *instrumenting_library_version=''*)
Returns a *Meter* for use by the given instrumentation library.

This function may return different *Meter* types (e.g. a no-op meter vs. a functional meter).

Parameters

- **instrumenting_module_name** (str) – The name of the instrumenting module (usually just __name__).

  This should *not* be the name of the module that is instrumented but the name of the module doing the instrumentation. E.g., instead of "requests", use "opentelemetry.ext.requests".
- **stateful** – True/False to indicate whether the meter will be stateful. True indicates the meter computes checkpoints from over the process lifetime. False indicates the meter computes checkpoints which describe the updates of a single collection period (deltas).
- **instrumenting_library_version** (str) – Optional. The version string of the instrumenting library. Usually this should be the same as pkg_resources.get_distribution(instrumenting_library_name).version.

Return type *Meter*

opentelemetry.sdk.resources package

class opentelemetry.sdk.resources.**Resource**(*labels*)
Bases: object

static create(*labels*)

Return type *Resource*

static create_empty()

Return type *Resource*

property labels

Return type Dict[str, Union[str, bool, int, float]]

merge(*other*)

Return type *Resource*

opentelemetry.sdk.trace package

Submodules

opentelemetry.sdk.trace.export

class opentelemetry.sdk.trace.export.**SpanExportResult**(*value*)
Bases: enum.Enum

An enumeration.

SUCCESS = 0

FAILURE = 1

class opentelemetry.sdk.trace.export.**SpanExporter**
Bases: object

Interface for exporting spans.

Interface to be implemented by services that want to export recorded in its own format.

To export data this MUST be registered to the :class`opentelemetry.sdk.trace.Tracer` using a *SimpleExportSpanProcessor* or a *BatchExportSpanProcessor*.

export(*spans*)
Exports a batch of telemetry data.

Parameters **spans** (Sequence[*Span*]) – The list of *opentelemetry.trace.Span* objects to be exported

Return type *SpanExportResult*

Returns The result of the export

shutdown()
Shuts down the exporter.

Called when the SDK is shut down.

Return type None

class opentelemetry.sdk.trace.export.**SimpleExportSpanProcessor**(*span_exporter*)

Bases: *opentelemetry.sdk.trace.SpanProcessor*

Simple SpanProcessor implementation.

SimpleExportSpanProcessor is an implementation of *SpanProcessor* that passes ended spans directly to the configured *SpanExporter*.

on_start(*span*)

Called when a *opentelemetry.trace.Span* is started.

This method is called synchronously on the thread that starts the span, therefore it should not block or throw an exception.

Parameters **span** (*Span*) – The *opentelemetry.trace.Span* that just started.

Return type None

on_end(*span*)

Called when a *opentelemetry.trace.Span* is ended.

This method is called synchronously on the thread that ends the span, therefore it should not block or throw an exception.

Parameters **span** (*Span*) – The *opentelemetry.trace.Span* that just ended.

Return type None

shutdown()

Called when a *opentelemetry.sdk.trace.Tracer* is shutdown.

Return type None

force_flush(*timeout_millis=30000*)

Export all ended spans to the configured Exporter that have not yet been exported.

Parameters **timeout_millis** (int) – The maximum amount of time to wait for spans to be exported.

Return type bool

Returns False if the timeout is exceeded, True otherwise.

class opentelemetry.sdk.trace.export.**BatchExportSpanProcessor**(*span_exporter*, *max_queue_size=2048*, *scheduled_delay_millis=5000*, *max_export_batch_size=512*)

Bases: *opentelemetry.sdk.trace.SpanProcessor*

Batch span processor implementation.

BatchExportSpanProcessor is an implementation of *SpanProcessor* that batches ended spans and pushes them to the configured *SpanExporter*.

on_start(*span*)

Called when a *opentelemetry.trace.Span* is started.

This method is called synchronously on the thread that starts the span, therefore it should not block or throw an exception.

Parameters **span** (*Span*) – The *opentelemetry.trace.Span* that just started.

Return type None

on_end(*span*)
Called when a *opentelemetry.trace.Span* is ended.

This method is called synchronously on the thread that ends the span, therefore it should not block or throw an exception.

Parameters **span** (*Span*) – The *opentelemetry.trace.Span* that just ended.

Return type None

worker()

export()
Exports at most max_export_batch_size spans.

Return type None

force_flush(*timeout_millis=30000*)
Export all ended spans to the configured Exporter that have not yet been exported.

Parameters **timeout_millis** (int) – The maximum amount of time to wait for spans to be exported.

Return type bool

Returns False if the timeout is exceeded, True otherwise.

shutdown()
Called when a *opentelemetry.sdk.trace.Tracer* is shutdown.

Return type None

class opentelemetry.sdk.trace.export.**ConsoleSpanExporter**(*out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *formatter=<function ConsoleSpanExporter.<lambda>>*)
Bases: *opentelemetry.sdk.trace.export.SpanExporter*

Implementation of *SpanExporter* that prints spans to the console.

This class can be used for diagnostic purposes. It prints the exported spans to the console STDOUT.

export(*spans*)
Exports a batch of telemetry data.

Parameters **spans** (Sequence[*Span*]) – The list of *opentelemetry.trace.Span* objects to be exported

Return type *SpanExportResult*

Returns The result of the export

class opentelemetry.sdk.trace.**SpanProcessor**
Bases: object

Interface which allows hooks for SDK's *Span* start and end method invocations.

Span processors can be registered directly using *TracerProvider.add_span_processor()* and they are invoked in the same order as they were registered.

on_start(*span*)
Called when a *opentelemetry.trace.Span* is started.

This method is called synchronously on the thread that starts the span, therefore it should not block or throw an exception.

Parameters **span** (*Span*) – The *opentelemetry.trace.Span* that just started.

Return type None

on_end(*span*)
Called when a *opentelemetry.trace.Span* is ended.

This method is called synchronously on the thread that ends the span, therefore it should not block or throw an exception.

Parameters **span** (*Span*) – The *opentelemetry.trace.Span* that just ended.

Return type None

shutdown()
Called when a *opentelemetry.sdk.trace.Tracer* is shutdown.

Return type None

force_flush(*timeout_millis=30000*)
Export all ended spans to the configured Exporter that have not yet been exported.

Parameters **timeout_millis** (int) – The maximum amount of time to wait for spans to be exported.

Return type bool

Returns False if the timeout is exceeded, True otherwise.

class opentelemetry.sdk.trace.**MultiSpanProcessor**
Bases: *opentelemetry.sdk.trace.SpanProcessor*

Implementation of *SpanProcessor* that forwards all received events to a list of *SpanProcessor*.

add_span_processor(*span_processor*)
Adds a SpanProcessor to the list handled by this instance.

Return type None

on_start(*span*)
Called when a *opentelemetry.trace.Span* is started.

This method is called synchronously on the thread that starts the span, therefore it should not block or throw an exception.

Parameters **span** (*Span*) – The *opentelemetry.trace.Span* that just started.

Return type None

on_end(*span*)
Called when a *opentelemetry.trace.Span* is ended.

This method is called synchronously on the thread that ends the span, therefore it should not block or throw an exception.

Parameters **span** (*Span*) – The *opentelemetry.trace.Span* that just ended.

Return type None

shutdown()
Called when a *opentelemetry.sdk.trace.Tracer* is shutdown.

Return type None

class opentelemetry.sdk.trace.**EventBase**(*name*, *timestamp=None*)
Bases: abc.ABC

property name

Return type str

property timestamp

Return type int

abstract property attributes

Return type Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]

class opentelemetry.sdk.trace.**Event**(*name*, *attributes=None*, *timestamp=None*)
Bases: *opentelemetry.sdk.trace.EventBase*

A text annotation with a set of attributes.

Parameters

- **name** (str) – Name of the event.
- **attributes** (Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]) – Attributes of the event.
- **timestamp** (Optional[int]) – Timestamp of the event. If None it will filled automatically.

property attributes

Return type Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]

class opentelemetry.sdk.trace.**LazyEvent**(*name*, *event_formatter*, *timestamp=None*)
Bases: *opentelemetry.sdk.trace.EventBase*

A text annotation with a set of attributes.

Parameters

- **name** (str) – Name of the event.
- **event_formatter** (Callable[[], Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]]) – Callable object that returns the attributes of the event.
- **timestamp** (Optional[int]) – Timestamp of the event. If None it will filled automatically.

property attributes

Return type Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]

class opentelemetry.sdk.trace.**Span**(*name*, *context*, *parent=None*, *sampler=None*, *trace_config=None*, *resource=<opentelemetry.sdk.resources.Resource object>*, *attributes=None*, *events=None*, *links=()*, *kind=<SpanKind.INTERNAL: 0>*, *span_processor=<opentelemetry.sdk.trace.SpanProcessor object>*, *instrumentation_info=None*, *set_status_on_exception=True*)

Bases: *opentelemetry.trace.Span*

See *opentelemetry.trace.Span*.

Users should create *Span* objects via the *Tracer* instead of this constructor.

Parameters

- **name** (str) – The name of the operation this span represents
- **context** (*SpanContext*) – The immutable span context
- **parent** (Optional[*SpanContext*]) – This span's parent's *SpanContext*, or null if this is a root span
- **sampler** (Optional[*Sampler*]) – The sampler used to create this span
- **trace_config** (None) – TODO
- **resource** (*Resource*) – Entity producing telemetry
- **attributes** (Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]) – The span's attributes to be exported
- **events** (Optional[Sequence[*Event*]]) – Timestamped events to be exported
- **links** (Sequence[*Link*]) – Links to other spans to be exported
- **span_processor** (*SpanProcessor*) – *SpanProcessor* to invoke when starting and ending this *Span*.

property start_time

property end_time

to_json(*indent=4*)

get_context()
Gets the span's SpanContext.

Get an immutable, serializable identifier for this span that can be used to create new child spans.

Returns A *SpanContext* with a copy of this span's immutable state.

set_attribute(*key*, *value*)
Sets an Attribute.

Sets a single Attribute with the key and value passed as arguments.

Return type None

add_event(*name*, *attributes=None*, *timestamp=None*)
Adds an *Event*.

Adds a single *Event* with the name and, optionally, a timestamp and attributes passed as arguments. Implementations should generate a timestamp if the *timestamp* argument is omitted.

Return type None

add_lazy_event(*name*, *event_formatter*, *timestamp=None*)
Adds an *Event*.

Adds a single *Event* with the name, an event formatter that calculates the attributes lazily and, optionally, a timestamp. Implementations should generate a timestamp if the *timestamp* argument is omitted.

Return type None

start(*start_time=None*)

Return type None

end(*end_time=None*)
Sets the current time as the span's end time.

The span's end time is the wall time at which the operation finished.

Only the first call to *end* should modify the span, and implementations are free to ignore or raise on further calls.

Return type None

update_name(*name*)
Updates the *Span* name.

This will override the name provided via *Tracer.start_span()*.

Upon this update, any sampling behavior based on Span name will depend on the implementation.

Return type None

is_recording_events()
Returns whether this span will be recorded.

Returns true if this Span is active and recording information like events with the add_event operation and attributes using set_attribute.

Return type bool

set_status(*status*)
Sets the Status of the Span. If used, this will override the default Span status, which is OK.

Return type None

opentelemetry.sdk.trace.**generate_span_id**()
Get a new random span ID.

Return type int

Returns A random 64-bit int for use as a span ID

opentelemetry.sdk.trace.**generate_trace_id**()
Get a new random trace ID.

Return type int

Returns A random 128-bit int for use as a trace ID

class opentelemetry.sdk.trace.**Tracer**(*source*, *instrumentation_info*)
Bases: *opentelemetry.trace.Tracer*

See *opentelemetry.trace.Tracer*.

Parameters

- **name** – The name of the tracer.
- **shutdown_on_exit** – Register an atexit hook to shut down the tracer when the application exits.

get_current_span()
Gets the currently active span from the context.

If there is no current span, return a placeholder span with an invalid context.

Returns The currently active *Span*, or a placeholder span with an invalid *SpanContext*.

start_as_current_span(*name*, *parent=<opentelemetry.trace.DefaultSpan object>*, *kind=<SpanKind.INTERNAL: 0>*, *attributes=None*, *links=()*)

Context manager for creating a new span and set it as the current span in this tracer's context.

On exiting the context manager stops the span and set its parent as the current span.

Example:

```
with tracer.start_as_current_span("one") as parent:
    parent.add_event("parent's event")
    with tracer.start_as_current_span("two") as child:
        child.add_event("child's event")
        tracer.get_current_span()  # returns child
    tracer.get_current_span()      # returns parent
tracer.get_current_span()          # returns previously active span
```

This is a convenience method for creating spans attached to the tracer's context. Applications that need more control over the span lifetime should use *start_span()* instead. For example:

```
with tracer.start_as_current_span(name) as span:
    do_work()
```

is equivalent to:

```
span = tracer.start_span(name)
with tracer.use_span(span, end_on_exit=True):
    do_work()
```

Parameters

- **name** (str) – The name of the span to be created.
- **parent** (Union[*Span*, *SpanContext*, None]) – The span's parent. Defaults to the current span.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.
- **attributes** (Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]) – The span's attributes.
- **links** (Sequence[*Link*]) – Links span to other spans

Yields The newly-created span.

Return type Iterator[*Span*]

start_span(*name*, *parent=<opentelemetry.trace.DefaultSpan object>*, *kind=<SpanKind.INTERNAL: 0>*, *attributes=None*, *links=()*, *start_time=None*, *set_status_on_exception=True*)

Starts a span.

Create a new span. Start the span without setting it as the current span in this tracer's context.

By default the current span will be used as parent, but an explicit parent can also be specified, either a *Span* or a *SpanContext*. If the specified value is None, the created span will be a root span.

The span can be used as context manager. On exiting, the span will be ended.

Example:

```
# tracer.get_current_span() will be used as the implicit parent.
# If none is found, the created span will be a root instance.
with tracer.start_span("one") as child:
    child.add_event("child's event")
```

Applications that need to set the newly created span as the current instance should use *start_as_current_span()* instead.

Parameters

- **name** (str) – The name of the span to be created.
- **parent** (Union[*Span*, *SpanContext*, None]) – The span's parent. Defaults to the current span.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.
- **attributes** (Optional[Dict[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]) – The span's attributes.
- **links** (Sequence[*Link*]) – Links span to other spans
- **start_time** (Optional[int]) – Sets the start time of a span
- **set_status_on_exception** (bool) – Only relevant if the returned span is used in a with/context manager. Defines wether the span status will be automatically set to UNKNOWN when an uncaught exception is raised in the span with block. The span status won't be set by this mechanism if it was previousy set manually.

Return type *Span*

Returns The newly-created span.

use_span(*span*, *end_on_exit=False*)

Context manager for controlling a span's lifetime.

Set the given span as the current span in this tracer's context.

On exiting the context manager set the span that was previously active as the current span (this is usually but not necessarily the parent of the given span). If end_on_exit is True, then the span is also ended when exiting the context manager.

Parameters

- **span** (*Span*) – The span to start and make current.
- **end_on_exit** (bool) – Whether to end the span automatically when leaving the context manager.

Return type Iterator[*Span*]

class opentelemetry.sdk.trace.**TracerProvider**(*sampler=<opentelemetry.trace.sampling.StaticSampler object>*, *resource=<opentelemetry.sdk.resources.Resource object>*, *shutdown_on_exit=True*)

Bases: *opentelemetry.trace.TracerProvider*

get_tracer(*instrumenting_module_name*, *instrumenting_library_version=''*)

Returns a *Tracer* for use by the given instrumentation library.

For any two calls it is undefined whether the same or different *Tracer* instances are returned, even for different library names.

This function may return different *Tracer* types (e.g. a no-op tracer vs. a functional tracer).

Parameters

- **instrumenting_module_name** (str) – The name of the instrumenting module (usually just __name__).

  This should *not* be the name of the module that is instrumented but the name of the module doing the instrumentation. E.g., instead of "requests", use "opentelemetry.ext.requests".

- **instrumenting_library_version** (str) – Optional. The version string of the instrumenting library. Usually this should be the same as pkg_resources.get_distribution(instrumenting_library_name).version.

Return type *Tracer*

static get_current_span()

Return type *Span*

add_span_processor(*span_processor*)
Registers a new *SpanProcessor* for this *TracerProvider*.

The span processors are invoked in the same order they are registered.

Return type None

shutdown()
Shut down the span processors added to the tracer.

2.1.4 OpenTelemetry Python Autoinstrumentation

This package provides a couple of commands that help automatically instruments a program:

opentelemetry-auto-instrumentation

```
opentelemetry-auto-instrumentation python program.py
```

The code in program.py needs to use one of the packages for which there is an OpenTelemetry integration. For a list of the available integrations please check *here*.

opentelemetry-bootstrap

```
opentelemetry-bootstrap --action=install|requirements
```

This commands inspects the active Python site-packages and figures out which instrumentation packages the user might want to install. By default it prints out a list of the suggested instrumentation packages which can be added to a requirements.txt file. It also supports installing the suggested packages when run with --action=install flag.

Submodules

opentelemetry.auto_instrumentation.instrumentor package

OpenTelemetry Base Instrumentor

class opentelemetry.auto_instrumentation.instrumentor.**BaseInstrumentor**
Bases: abc.ABC

An ABC for instrumentors

Child classes of this ABC should instrument specific third party libraries or frameworks either by using the opentelemetry-auto-instrumentation command or by calling their methods directly.

Since every third party library or framework is different and has different instrumentation needs, more methods can be added to the child classes as needed to provide practical instrumentation to the end user.

instrument(***kwargs*)
Instrument the library

This method will be called without any optional arguments by the opentelemetry-auto-instrumentation command. The configuration of the instrumentation when done in this way should be done by previously setting the configuration (using environment variables or any other mechanism) that will be used later by the code in the instrument implementation via the global Configuration object.

The instrument methods kwargs should default to values from the Configuration object.

This means that calling this method directly without passing any optional values should do the very same thing that the opentelemetry-auto-instrumentation command does. This approach is followed because the Configuration object is immutable.

uninstrument(***kwargs*)
Uninstrument the library

See BaseInstrumentor.instrument for more information regarding the usage of kwargs.

2.1.5 OpenTelemetry aiohttp client Integration

The opentelemetry-ext-aiohttp-client package allows tracing HTTP requests made by the aiohttp client library.

Usage

```
import aiohttp
from opentelemetry.ext.aiohttp_client import (
    create_trace_config,
    url_path_span_name
)
import yarl

def strip_query_params(url: yarl.URL) -> str:
    return str(url.with_query(None))

async with aiohttp.ClientSession(trace_configs=[create_trace_config(
        # Remove all query params from the URL attribute on the span.
        url_filter=strip_query_params,
        # Use the URL's path as the span name.
```

(continues on next page)

(continued from previous page)

```
        span_name=url_path_span_name
)]) as session:
    async with session.get(url) as response:
        await response.text()
```

opentelemetry.ext.aiohttp_client.**http_status_to_canonical_code**(*status*)

Return type *StatusCanonicalCode*

opentelemetry.ext.aiohttp_client.**url_path_span_name**(*params*)

Extract a span name from the request URL path.

A simple callable to extract the path portion of the requested URL for use as the span name.

Parameters **params** (*aiohttp.TraceRequestStartParams*) – Parameters describing the traced request.

Returns The URL path.

Return type str

opentelemetry.ext.aiohttp_client.**create_trace_config**(*url_filter=None*, *span_name=None*)

Create an aiohttp-compatible trace configuration.

One span is created for the entire HTTP request, including initial TCP/TLS setup if the connection doesn't exist.

By default the span name is set to the HTTP request method.

Example usage:

```
import aiohttp
from opentelemetry.ext.aiohttp_client import create_trace_config

async with aiohttp.ClientSession(trace_configs=[create_trace_config()]) as␣
↪session:
    async with session.get(url) as response:
        await response.text()
```

Parameters

- **url_filter** (Optional[Callable[[str], str]]) – A callback to process the requested URL prior to adding it as a span attribute. This can be useful to remove sensitive data such as API keys or user personal information.
- **span_name** (*str*) – Override the default span name.

Returns An object suitable for use with aiohttp.ClientSession.

Return type aiohttp.TraceConfig

2.1.6 opentelemetry.ext.asgi package

Module contents

The opentelemetry-ext-asgi package provides an ASGI middleware that can be used on any ASGI framework (such as Django-channels / Quart) to track requests timing through OpenTelemetry.

`opentelemetry.ext.asgi.`**`get_header_from_scope`**(*scope*, *header_name*)
Retrieve a HTTP header value from the ASGI scope.

Return type `List[str]`

Returns A list with a single string with the header value if it exists, else an empty list.

`opentelemetry.ext.asgi.`**`http_status_to_canonical_code`**(*code*, *allow_redirect=True*)

`opentelemetry.ext.asgi.`**`collect_request_attributes`**(*scope*)
Collects HTTP request attributes from the ASGI scope and returns a dictionary to be used as span creation attributes.

`opentelemetry.ext.asgi.`**`set_status_code`**(*span*, *status_code*)
Adds HTTP response attributes to span using the status_code argument.

`opentelemetry.ext.asgi.`**`get_default_span_name`**(*scope*)
Default implementation for name_callback

`class` `opentelemetry.ext.asgi.`**`OpenTelemetryMiddleware`**(*app*, *name_callback=None*)
Bases: `object`

The ASGI application middleware.

This class is an ASGI middleware that starts and annotates spans for any requests it is invoked with.

Parameters

- **`app`** – The ASGI application callable to forward requests to.
- **`name_callback`** – Callback which calculates a generic span name for an incoming HTTP request based on the ASGI scope. Optional: Defaults to get_default_span_name.

2.1.7 OpenTelemetry Datadog Exporter

2.1.8 OpenTelemetry Database API Integration

The trace integration with Database API supports libraries that follow the Python Database API Specification v2.0. https://www.python.org/dev/peps/pep-0249/

Usage

```
import mysql.connector
import pyodbc

from opentelemetry import trace
from opentelemetry.ext.dbapi import trace_integration
from opentelemetry.trace import TracerProvider

trace.set_tracer_provider(TracerProvider())
```

(continues on next page)

(continued from previous page)

```
# Ex: mysql.connector
trace_integration(mysql.connector, "connect", "mysql", "sql")
# Ex: pyodbc
trace_integration(pyodbc, "Connection", "odbc", "sql")
```

API

opentelemetry.ext.dbapi.**trace_integration**(*connect_module*, *connect_method_name*, *database_component*, *database_type=''*, *connection_attributes=None*, *tracer_provider=None*)

Integrate with DB API library. https://www.python.org/dev/peps/pep-0249/

Parameters

- **connect_module** (Callable[..., Any]) – Module name where connect method is available.
- **connect_method_name** (str) – The connect method name.
- **database_component** (str) – Database driver name or database name "JDBI", "jdbc", "odbc", "postgreSQL".
- **database_type** (str) – The Database type. For any SQL database, "sql".
- **connection_attributes** (Optional[Dict]) – Attribute names for database, port, host and user in Connection object.
- **tracer_provider** (Optional[*TracerProvider*]) – The *opentelemetry.trace.TracerProvider* to use. If ommited the current configured one is used.

opentelemetry.ext.dbapi.**wrap_connect**(*tracer*, *connect_module*, *connect_method_name*, *database_component*, *database_type=''*, *connection_attributes=None*)

Integrate with DB API library. https://www.python.org/dev/peps/pep-0249/

Parameters

- **tracer** (*Tracer*) – The *opentelemetry.trace.Tracer* to use.
- **connect_module** (Callable[..., Any]) – Module name where connect method is available.
- **connect_method_name** (str) – The connect method name.
- **database_component** (str) – Database driver name or database name "JDBI", "jdbc", "odbc", "postgreSQL".
- **database_type** (str) – The Database type. For any SQL database, "sql".
- **connection_attributes** (Optional[Dict]) – Attribute names for database, port, host and user in Connection object.

opentelemetry.ext.dbapi.**unwrap_connect**(*connect_module*, *connect_method_name*)

Disable integration with DB API library. https://www.python.org/dev/peps/pep-0249/

Parameters

- **connect_module** (Callable[..., Any]) – Module name where the connect method is available.
- **connect_method_name** (str) – The connect method name.

opentelemetry.ext.dbapi.**instrument_connection**(*tracer*, *connection*, *database_component*, *database_type=''*, *connection_attributes=None*)

Enable instrumentation in a database connection.

Parameters

- **tracer** – The *opentelemetry.trace.Tracer* to use.
- **connection** – The connection to instrument.
- **database_component** (str) – Database driver name or database name "JDBI", "jdbc", "odbc", "postgreSQL".
- **database_type** (str) – The Database type. For any SQL database, "sql".
- **connection_attributes** (Optional[Dict]) – Attribute names for database, port, host and user in a connection object.

Returns An instrumented connection.

opentelemetry.ext.dbapi.**uninstrument_connection**(*connection*)

Disable instrumentation in a database connection.

Parameters **connection** – The connection to uninstrument.

Returns An uninstrumented connection.

class opentelemetry.ext.dbapi.**DatabaseApiIntegration**(*tracer*, *database_component*, *database_type='sql'*, *connection_attributes=None*)

Bases: object

wrapped_connection(*connect_method*, *args*, *kwargs*)

Add object proxy to connection object.

get_connection_attributes(*connection*)

opentelemetry.ext.dbapi.**get_traced_connection_proxy**(*connection*, *db_api_integration*, **args*, ***kwargs*)

class opentelemetry.ext.dbapi.**TracedCursor**(*db_api_integration*)

Bases: object

traced_execution(*query_method*, **args*, ***kwargs*)

opentelemetry.ext.dbapi.**get_traced_cursor_proxy**(*cursor*, *db_api_integration*, **args*, ***kwargs*)

2.1.9 OpenTelemetry Django Instrumentation

class opentelemetry.ext.django.**DjangoInstrumentor**

Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

An instrumentor for Django

See *BaseInstrumentor*

2.1.10 OpenTelemetry Flask Integration

This library builds on the OpenTelemetry WSGI middleware to track web requests in Flask applications. In addition to opentelemetry-ext-wsgi, it supports flask-specific features such as:

- The Flask endpoint name is used as the Span name.
- The `http.route` Span attribute is set so that one can see which URL rule matched a request.

Usage

```
from flask import Flask
from opentelemetry.ext.flask import FlaskInstrumentor

app = Flask(__name__)

FlaskInstrumentor().instrument_app(app)

@app.route("/")
def hello():
    return "Hello!"

if __name__ == "__main__":
    app.run(debug=True)
```

API

opentelemetry.ext.flask.**get_excluded_hosts**()

opentelemetry.ext.flask.**get_excluded_paths**()

class opentelemetry.ext.flask.**FlaskInstrumentor**

Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

An instrumentor for flask.Flask

See *BaseInstrumentor*

instrument_app(*app*)

uninstrument_app(*app*)

2.1.11 OpenTelemetry gRPC Integration

Module contents

opentelemetry.ext.grpc.**client_interceptor**(*tracer_provider=None*)

Create a gRPC client channel interceptor.

Parameters **tracer** – The tracer to use to create client-side spans.

Returns An invocation-side interceptor object.

opentelemetry.ext.grpc.**server_interceptor**(*tracer_provider=None*)

Create a gRPC server interceptor.

Parameters **tracer** – The tracer to use to create server-side spans.

Returns A service-side interceptor object.

2.1.12 Opentelemetry Jaeger Exporter

The **OpenTelemetry Jaeger Exporter** allows to export OpenTelemetry traces to Jaeger. This exporter always send traces to the configured agent using Thrift compact protocol over UDP. An optional collector can be configured, in this case Thrift binary protocol over HTTP is used. gRPC is still not supported by this implementation.

Usage

```
from opentelemetry import trace
from opentelemetry.ext import jaeger
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchExportSpanProcessor

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

# create a JaegerSpanExporter
jaeger_exporter = jaeger.JaegerSpanExporter(
    service_name='my-helloworld-service',
    # configure agent
    agent_host_name='localhost',
    agent_port=6831,
    # optional: configure also collector
    # collector_host_name='localhost',
    # collector_port=14268,
    # collector_endpoint='/api/traces?format=jaeger.thrift',
    # username=xxxx, # optional
    # password=xxxx, # optional
)

# Create a BatchExportSpanProcessor and add the exporter to it
span_processor = BatchExportSpanProcessor(jaeger_exporter)

# add to the tracer
trace.get_tracer_provider().add_span_processor(span_processor)

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

API

class opentelemetry.ext.jaeger.**JaegerSpanExporter**(*service_name*, *agent_host_name='localhost'*, *agent_port=6831*, *collector_host_name=None*, *collector_port=None*, *collector_endpoint='/api/traces?format=jaeger.thrift'*, *username=None*, *password=None*)

Bases: *opentelemetry.sdk.trace.export.SpanExporter*

Jaeger span exporter for OpenTelemetry.

Parameters

- **service_name** – Service that logged an annotation in a trace.Classifier when query for spans.
- **agent_host_name** – The host name of the Jaeger-Agent.
- **agent_port** – The port of the Jaeger-Agent.
- **collector_host_name** – The host name of the Jaeger-Collector HTTP Thrift.
- **collector_port** – The port of the Jaeger-Collector HTTP Thrift.
- **collector_endpoint** – The endpoint of the Jaeger-Collector HTTP Thrift.
- **username** – The user name of the Basic Auth if authentication is required.
- **password** – The password of the Basic Auth if authentication is required.

property agent_client

property collector

export(*spans*)

Exports a batch of telemetry data.

Parameters **spans** – The list of *opentelemetry.trace.Span* objects to be exported

Returns The result of the export

shutdown()

Shuts down the exporter.

Called when the SDK is shut down.

class opentelemetry.ext.jaeger.**AgentClientUDP**(*host_name*, *port*, *max_packet_size=65000*, *client=<class 'opentelemetry.ext.jaeger.gen.agent.Agent.Client'>*)

Bases: object

Implement a UDP client to agent.

Parameters

- **host_name** – The host name of the Jaeger server.
- **port** – The port of the Jaeger server.
- **max_packet_size** – Maximum size of UDP packet.
- **client** – Class for creating new client objects for agencies.

emit(*batch*)

Parameters **batch** (*Batch*) – Object to emit Jaeger spans.

class opentelemetry.ext.jaeger.**Collector**(*thrift_url=''*, *auth=None*)

Bases: object

Submits collected spans to Thrift HTTP server.

Parameters

- **thrift_url** – URL of the Jaeger HTTP Thrift.
- **auth** – Auth tuple that contains username and password for Basic Auth.

submit(*batch*)

Submits batches to Thrift HTTP Server through Binary Protocol.

Parameters **batch** (*Batch*) – Object to emit Jaeger spans.

Submodules

class opentelemetry.ext.jaeger.gen.jaeger.ttypes.**TagType**
Bases: object

STRING = 0

DOUBLE = 1

BOOL = 2

LONG = 3

BINARY = 4

class opentelemetry.ext.jaeger.gen.jaeger.ttypes.**SpanRefType**
Bases: object

CHILD_OF = 0

FOLLOWS_FROM = 1

class opentelemetry.ext.jaeger.gen.jaeger.ttypes.**Tag**(*key=None*, *vType=None*, *vStr=None*, *vDouble=None*, *vBool=None*, *vLong=None*, *vBinary=None*)
Bases: object

- key

- vType

- vStr

- vDouble

- vBool

- vLong

- vBinary

thrift_spec = (None, (1, 11, 'key', 'UTF8', None), (2, 8, 'vType', None, None), (3, 11

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.ext.jaeger.gen.jaeger.ttypes.**Log**(*timestamp=None*, *fields=None*)
Bases: object

- timestamp

- fields

thrift_spec = (None, (1, 10, 'timestamp', None, None), (2, 15, 'fields', (12, (<class

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.ext.jaeger.gen.jaeger.ttypes.**SpanRef**(*refType=None*, *traceIdLow=None*, *traceIdHigh=None*, *spanId=None*)

Bases: object

- refType

- traceIdLow

- traceIdHigh

- spanId

thrift_spec = (None, (1, 8, 'refType', None, None), (2, 10, 'traceIdLow', None, None),

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.ext.jaeger.gen.jaeger.ttypes.**Span**(*traceIdLow=None*, *traceIdHigh=None*, *spanId=None*, *parentSpanId=None*, *operationName=None*, *references=None*, *flags=None*, *startTime=None*, *duration=None*, *tags=None*, *logs=None*)

Bases: object

- traceIdLow

- traceIdHigh

- spanId

- parentSpanId

- operationName

- references

- flags

- startTime

- duration

- tags

- logs

thrift_spec = (None, (1, 10, 'traceIdLow', None, None), (2, 10, 'traceIdHigh', None, N

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.ext.jaeger.gen.jaeger.ttypes.**Process**(*serviceName=None*, *tags=None*)

Bases: object

- serviceName

- tags

thrift_spec = (None, (1, 11, 'serviceName', 'UTF8', None), (2, 15, 'tags', (12, (<clas

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.ext.jaeger.gen.jaeger.ttypes.**Batch**(*process=None*, *spans=None*)

Bases: object

- process

- spans

thrift_spec = (None, (1, 12, 'process', (<class 'opentelemetry.ext.jaeger.gen.jaeger.t

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.ext.jaeger.gen.jaeger.ttypes.**BatchSubmitResponse**(*ok=None*)

Bases: object

- ok

thrift_spec = (None, (1, 2, 'ok', None, None))

read(*iprot*)

write(*oprot*)

validate()

2.1.13 OpenTelemetry Jinja2 Instrumentation

Usage

The OpenTelemetry jinja2 integration traces templates loading, compilation and rendering.

Usage

```
from jinja2 import Environment, FileSystemLoader
from opentelemetry.ext.jinja2 import Jinja2Instrumentor
from opentelemetry import trace
from opentelemetry.trace import TracerProvider

trace.set_tracer_provider(TracerProvider())

Jinja2Instrumentor().instrument()

env = Environment(loader=FileSystemLoader("templates"))
template = env.get_template("mytemplate.html")
```

API

class opentelemetry.ext.jinja2.**Jinja2Instrumentor**
Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

An instrumentor for jinja2

See *BaseInstrumentor*

2.1.14 OpenTelemetry MySQL Integration

MySQL instrumentation supporting mysql-connector, it can be enabled by using `MySQLInstrumentor`.

Usage

```
import mysql.connector
from opentelemetry import trace
from opentelemetry.trace import TracerProvider
from opentelemetry.ext.mysql import MySQLInstrumentor

trace.set_tracer_provider(TracerProvider())

MySQLInstrumentor().instrument()

cnx = mysql.connector.connect(database="MySQL_Database")
cursor = cnx.cursor()
cursor.execute("INSERT INTO test (testField) VALUES (123)"
cursor.close()
cnx.close()
```

API

class opentelemetry.ext.mysql.**MySQLInstrumentor**
Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

instrument_connection(*connection*)
Enable instrumentation in a MySQL connection.

Parameters **connection** – The connection to instrument.

Returns An instrumented connection.

uninstrument_connection(*connection*)
Disable instrumentation in a MySQL connection.

Parameters **connection** – The connection to uninstrument.

Returns An uninstrumented connection.

2.1.15 OpenCensus Exporter

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

2.1.16 OpenTracing Shim for OpenTelemetry

2.1.17 OpenTelemetry Prometheus Exporter

This library allows export of metrics data to Prometheus.

Usage

The **OpenTelemetry Prometheus Exporter** allows export of OpenTelemetry metrics to Prometheus.

```
from opentelemetry import metrics
from opentelemetry.ext.prometheus import PrometheusMetricsExporter
from opentelemetry.sdk.metrics import Counter, Meter
from opentelemetry.sdk.metrics.export.controller import PushController
from prometheus_client import start_http_server

# Start Prometheus client
start_http_server(port=8000, addr="localhost")

# Meter is responsible for creating and recording metrics
metrics.set_meter_provider(MeterProvider())
meter = metrics.meter()
# exporter to export metrics to Prometheus
prefix = "MyAppPrefix"
exporter = PrometheusMetricsExporter(prefix)
# controller collects metrics created from meter and exports it via the
# exporter every interval
controller = PushController(meter, exporter, 5)

counter = meter.create_metric(
    "requests",
    "number of requests",
    "requests",
    int,
    Counter,
    ("environment",),
)

# Labels are used to identify key-values that are associated with a specific
# metric that you want to record. These are useful for pre-aggregation and can
# be used to store custom dimensions pertaining to a metric
labels = {"environment": "staging"}

counter.add(25, labels)
input("Press any key to exit...")
```

API

class opentelemetry.ext.prometheus.**PrometheusMetricsExporter**(*prefix=''*)
Bases: *opentelemetry.sdk.metrics.export.MetricsExporter*

Prometheus metric exporter for OpenTelemetry.

Parameters **prefix** (str) – single-word application prefix relevant to the domain the metric belongs to.

export(*metric_records*)
Exports a batch of telemetry data.

Parameters **metric_records** (Sequence[*MetricRecord*]) – A sequence of *MetricRecord* s. A *MetricRecord* contains the metric to be exported, the labels associated with that metric, as well as the aggregator used to export the current checkpointed value.

Return type *MetricsExportResult*

Returns The result of the export

shutdown()
Shuts down the exporter.

Called when the SDK is shut down.

Return type None

class opentelemetry.ext.prometheus.**CustomCollector**(*prefix=''*)
Bases: object

CustomCollector represents the Prometheus Collector object https://github.com/prometheus/client_python#custom-collectors

add_metrics_data(*metric_records*)

collect()
Collect fetches the metrics from OpenTelemetry and delivers them as Prometheus Metrics. Collect is invoked every time a prometheus.Gatherer is run for example when the HTTP endpoint is invoked by Prometheus.

2.1.18 OpenTelemetry Psycopg Integration

The integration with PostgreSQL supports the Psycopg library, it can be enabled by using Psycopg2Instrumentor.

Usage

```
import psycopg2
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.ext.psycopg2 import Psycopg2Instrumentor

trace.set_tracer_provider(TracerProvider())

Psycopg2Instrumentor().instrument()
```

(continues on next page)

(continued from previous page)

```
cnx = psycopg2.connect(database='Database')
cursor = cnx.cursor()
cursor.execute("INSERT INTO test (testField) VALUES (123)")
cursor.close()
cnx.close()
```

API

class opentelemetry.ext.psycopg2.**Psycopg2Instrumentor**

Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

instrument_connection(*connection*)

Enable instrumentation in a Psycopg2 connection.

Parameters **connection** – The connection to instrument.

Returns An instrumented connection.

uninstrument_connection(*connection*)

Disable instrumentation in a Psycopg2 connection.

Parameters **connection** – The connection to uninstrument.

Returns An uninstrumented connection.

2.1.19 OpenTelemetry pymongo Integration

The integration with MongoDB supports the pymongo library, it can be enabled using the `PymongoInstrumentor`.

Usage

```
from pymongo import MongoClient
from opentelemetry import trace
from opentelemetry.trace import TracerProvider
from opentelemetry.ext.pymongo import PymongoInstrumentor

trace.set_tracer_provider(TracerProvider())

PymongoInstrumentor().instrument()
client = MongoClient()
db = client["MongoDB_Database"]
collection = db["MongoDB_Collection"]
collection.find_one()
```

API

class opentelemetry.ext.pymongo.**CommandTracer**(*tracer*)
Bases: pymongo.monitoring.CommandListener

started(*event*)
Method to handle a pymongo CommandStartedEvent

succeeded(*event*)
Method to handle a pymongo CommandSucceededEvent

failed(*event*)
Method to handle a pymongo CommandFailedEvent

class opentelemetry.ext.pymongo.**PymongoInstrumentor**
Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

2.1.20 OpenTelemetry PyMySQL Integration

The integration with PyMySQL supports the PyMySQL library and can be enabled by using PyMySQLInstrumentor.

Usage

```
import pymysql
from opentelemetry import trace
from opentelemetry.ext.pymysql import PyMySQLInstrumentor
from opentelemetry.sdk.trace import TracerProvider

trace.set_tracer_provider(TracerProvider())

PyMySQLInstrumentor().instrument()

cnx = pymysql.connect(database="MySQL_Database")
cursor = cnx.cursor()
cursor.execute("INSERT INTO test (testField) VALUES (123)"
cnx.commit()
cursor.close()
cnx.close()
```

API

class opentelemetry.ext.pymysql.**PyMySQLInstrumentor**
Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

instrument_connection(*connection*)
Enable instrumentation in a PyMySQL connection.

Parameters **connection** – The connection to instrument.

Returns An instrumented connection.

uninstrument_connection(*connection*)
Disable instrumentation in a PyMySQL connection.

Parameters **connection** – The connection to uninstrument.

Returns An uninstrumented connection.

2.1.21 OpenTelemetry Redis Instrumentation

Instrument redis to report Redis queries.

There are two options for instrumenting code. The first option is to use the `opentelemetry-auto-instrumentation` executable which will automatically instrument your Redis client. The second is to programmatically enable instrumentation via the following code:

Usage

```
from opentelemetry import trace
from opentelemetry.ext.redis import RedisInstrumentor
from opentelemetry.sdk.trace import TracerProvider
import redis

trace.set_tracer_provider(TracerProvider())

# Instrument redis
RedisInstrumentor().instrument(tracer_provider=trace.get_tracer_provider())

# This will report a span with the default settings
client = redis.StrictRedis(host="localhost", port=6379)
client.get("my-key")
```

API

class opentelemetry.ext.redis.**RedisInstrumentor**

Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

An instrumentor for Redis See *BaseInstrumentor*

2.1.22 OpenTelemetry requests Integration

This library allows tracing HTTP requests made by the requests library.

Usage

```
import requests
import opentelemetry.ext.requests

# You can optionally pass a custom TracerProvider to RequestInstrumentor.instrument()
opentelemetry.ext.requests.RequestsInstrumentor().instrument()
response = requests.get(url="https://www.example.org/")
```

Limitations

Note that calls that do not use the higher-level APIs but use `requests.sessions.Session.send` (or an alias thereof) directly, are currently not traced. If you find any other way to trigger an untraced HTTP request, please report it via a GitHub issue with `[requests: untraced API]` in the title.

API

class opentelemetry.ext.requests.**RequestsInstrumentor**

Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

An instrumentor for requests See *BaseInstrumentor*

static uninstrument_session(*session*)

Disables instrumentation on the session object.

2.1.23 OpenTelemetry SQLAlchemy Instrumentation

Instrument sqlalchemy to report SQL queries.

There are two options for instrumenting code. The first option is to use the `opentelemetry-auto-instrumentation` executable which will automatically instrument your SQLAlchemy engine. The second is to programmatically enable instrumentation via the following code:

Usage

```
from sqlalchemy import create_engine

from opentelemetry import trace
from opentelemetry.ext.sqlalchemy import SQLAlchemyInstrumentor
from opentelemetry.sdk.trace import TracerProvider
import sqlalchemy

trace.set_tracer_provider(TracerProvider())
engine = create_engine("sqlite:///:memory:")
SQLAlchemyInstrumentor().instrument(
    engine=engine,
    service="service-A",
)
```

API

class opentelemetry.ext.sqlalchemy.**SQLAlchemyInstrumentor**

Bases: *opentelemetry.auto_instrumentation.instrumentor.BaseInstrumentor*

An instrumentor for SQLAlchemy See *BaseInstrumentor*

2.1.24 OpenTelemetry WSGI Middleware

This library provides a WSGI middleware that can be used on any WSGI framework (such as Django / Flask) to track requests timing through OpenTelemetry.

Usage (Flask)

```
from flask import Flask
from opentelemetry.ext.wsgi import OpenTelemetryMiddleware

app = Flask(__name__)
app.wsgi_app = OpenTelemetryMiddleware(app.wsgi_app)

@app.route("/")
def hello():
    return "Hello!"

if __name__ == "__main__":
    app.run(debug=True)
```

Usage (Django)

Modify the application's `wsgi.py` file as shown below.

```
import os
from opentelemetry.ext.wsgi import OpenTelemetryMiddleware
from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'application.settings')

application = get_wsgi_application()
application = OpenTelemetryMiddleware(application)
```

API

opentelemetry.ext.wsgi.**get_header_from_environ**(*environ*, *header_name*)
Retrieve a HTTP header value from the PEP3333-conforming WSGI environ.

Return type List[str]

Returns A list with a single string with the header value if it exists, else an empty list.

opentelemetry.ext.wsgi.**setifnotnone**(*dic*, *key*, *value*)

opentelemetry.ext.wsgi.**http_status_to_canonical_code**(*code*, *allow_redirect=True*)

opentelemetry.ext.wsgi.**collect_request_attributes**(*environ*)
Collects HTTP request attributes from the PEP3333-conforming WSGI environ and returns a dictionary to be used as span creation attributes.

opentelemetry.ext.wsgi.**add_response_attributes**(*span*, *start_response_status*, *response_headers*)
Adds HTTP response attributes to span using the arguments passed to a PEP3333-conforming start_response callable.

opentelemetry.ext.wsgi.**get_default_span_name**(*environ*)
Calculates a (generic) span name for an incoming HTTP request based on the PEP3333 conforming WSGI environ.

class opentelemetry.ext.wsgi.**OpenTelemetryMiddleware**(*wsgi*)
Bases: object

The WSGI application middleware.

This class is a PEP 3333 conforming WSGI middleware that starts and annotates spans for any requests it is invoked with.

Parameters **wsgi** – The WSGI application callable to forward requests to.

2.1.25 Opentelemetry Zipkin Exporter

This library allows to export tracing data to Zipkin.

Usage

The **OpenTelemetry Zipkin Exporter** allows to export OpenTelemetry traces to Zipkin. This exporter always send traces to the configured Zipkin collector using HTTP.

```
from opentelemetry import trace
from opentelemetry.ext import zipkin
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchExportSpanProcessor

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

# create a ZipkinSpanExporter
zipkin_exporter = zipkin.ZipkinSpanExporter(
    service_name="my-helloworld-service",
    # optional:
    # host_name="localhost",
    # port=9411,
    # endpoint="/api/v2/spans",
    # protocol="http",
    # ipv4="",
    # ipv6="",
    # retry=False,
)

# Create a BatchExportSpanProcessor and add the exporter to it
span_processor = BatchExportSpanProcessor(zipkin_exporter)

# add to the tracer
trace.get_tracer_provider().add_span_processor(span_processor)

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

API

class opentelemetry.ext.zipkin.**ZipkinSpanExporter**(*service_name*, *host_name='localhost'*, *port=9411*, *endpoint='/api/v2/spans'*, *protocol='http'*, *ipv4=None*, *ipv6=None*, *retry=False*)

Bases: *opentelemetry.sdk.trace.export.SpanExporter*

Zipkin span exporter for OpenTelemetry.

Parameters

- **service_name** (str) – Service that logged an annotation in a trace.Classifier when query for spans.
- **host_name** (str) – The host name of the Zipkin server
- **port** (int) – The port of the Zipkin server
- **endpoint** (str) – The endpoint of the Zipkin server
- **protocol** (str) – The protocol used for the request.
- **ipv4** (Optional[str]) – Primary IPv4 address associated with this connection.
- **ipv6** (Optional[str]) – Primary IPv6 address associated with this connection.
- **retry** (Optional[str]) – Set to True to configure the exporter to retry on failure.

export(*spans*)

Exports a batch of telemetry data.

Parameters **spans** (Sequence[*Span*]) – The list of *opentelemetry.trace.Span* objects to be exported

Return type *SpanExportResult*

Returns The result of the export

shutdown()

Shuts down the exporter.

Called when the SDK is shut down.

Return type None

2.1.26 Autoinstrumentation

Overview

This example shows how to use auto-instrumentation in OpenTelemetry. This example is also based on a previous example for OpenTracing that can be found here.

The source files of these examples are available here.

This example uses 2 scripts whose main difference is they being instrumented manually or not:

1. server_instrumented.py which has been instrumented manually
2. server_uninstrumented.py which has not been instrumented manually

The former will be run without the automatic instrumentation agent and the latter with the automatic instrumentation agent. They should produce the same result, showing that the automatic instrumentation agent does the equivalent of what manual instrumentation does.

In order to understand this better, here is the relevant part of both scripts:

Manually instrumented server

server_instrumented.py

```
@app.route("/server_request")
def server_request():
    with tracer.start_as_current_span(
        "server_request",
        parent=propagators.extract(
            lambda dict_, key: dict_.get(key, []), request.headers
        )["current-span"],
    ):
        print(request.args.get("param"))
        return "served"
```

Publisher not instrumented manually

server_uninstrumented.py

```
@app.route("/server_request")
def server_request():
    print(request.args.get("param"))
    return "served"
```

Preparation

This example will be executed in a separate virtual environment:

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

Installation

```
$ pip install opentelemetry-sdk
$ pip install opentelemetry-auto-instrumentation
$ pip install opentelemetry-ext-flask
$ pip install requests
```

Execution

Execution of the manually instrumented server

This is done in 2 separate consoles, one to run each of the scripts that make up this example:

```
$ source auto_instrumentation/bin/activate
$ python server_instrumented.py
```

```
$ source auto_instrumentation/bin/activate
$ python client.py testing
```

The execution of `server_instrumented.py` should return an output similar to:

```
{
    "name": "server_request",
    "context": {
        "trace_id": "0xfa002aad260b5f7110db674a9ddfcd23",
        "span_id": "0x8b8bbaf3ca9c5131",
        "trace_state": "{}"
    },
    "kind": "SpanKind.SERVER",
    "parent_id": null,
    "start_time": "2020-04-30T17:28:57.886397Z",
    "end_time": "2020-04-30T17:28:57.886490Z",
    "status": {
        "canonical_code": "OK"
    },
    "attributes": {
        "component": "http",
        "http.method": "GET",
        "http.server_name": "127.0.0.1",
        "http.scheme": "http",
        "host.port": 8082,
        "http.host": "localhost:8082",
        "http.target": "/server_request?param=testing",
        "net.peer.ip": "127.0.0.1",
        "net.peer.port": 52872,
        "http.flavor": "1.1"
    },
    "events": [],
    "links": []
}
```

Execution of an automatically instrumented server

Now, kill the execution of `server_instrumented.py` with `ctrl + c` and run this instead:

```
$ opentelemetry-auto-instrumentation python server_uninstrumented.py
```

In the console where you previously executed `client.py`, run again this again:

```
$ python client.py testing
```

The execution of `server_uninstrumented.py` should return an output similar to:

```
{
    "name": "server_request",
    "context": {
        "trace_id": "0x9f528e0b76189f539d9c21b1a7a2fc24",
        "span_id": "0xd79760685cd4c269",
        "trace_state": "{}"
    },
    "kind": "SpanKind.SERVER",
    "parent_id": "0xb4fb7eee22ef78e4",
    "start_time": "2020-04-30T17:10:02.400604Z",
    "end_time": "2020-04-30T17:10:02.401858Z",
    "status": {
        "canonical_code": "OK"
    },
    "attributes": {
        "component": "http",
        "http.method": "GET",
        "http.server_name": "127.0.0.1",
        "http.scheme": "http",
        "host.port": 8082,
        "http.host": "localhost:8082",
        "http.target": "/server_request?param=testing",
        "net.peer.ip": "127.0.0.1",
        "net.peer.port": 48240,
        "http.flavor": "1.1",
        "http.route": "/server_request",
        "http.status_text": "OK",
        "http.status_code": 200
    },
    "events": [],
    "links": []
}
```


Both outputs are equivalent since the automatic instrumentation does what the manual instrumentation does too.

2.1.27 Basic Meter

These examples show how to use OpenTelemetry to capture and report metrics.

There are three different examples:

- basic_metrics: Shows to how create a metric instrument, how to configure an exporter and a controller and also how to capture data by using the direct calling convention.
- calling_conventions: Shows how to use the direct, bound and batch calling conventions.
- observer: Shows how to use the observer instrument.

The source files of these examples are available here.

Installation

```
pip install opentelemetry-api
pip install opentelemetry-sdk
pip install psutil # needed to get ram and cpu usage in the observer example
```

Run the Example

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

The output will be shown in the console after few seconds.

Useful links

- OpenTelemetry
- *opentelemetry.metrics package*

2.1.28 Datadog Exporter Example

These examples show how to use OpenTelemetry to send tracing data to Datadog.

Basic Example

- Installation

```
pip install opentelemetry-api
pip install opentelemetry-sdk
pip install opentelemetry-ext-datadog
```

- Start Datadog Agent

```
docker run --rm \
    -v /var/run/docker.sock:/var/run/docker.sock:ro \
    -v /proc/:/host/proc/:ro \
    -v /sys/fs/cgroup/:/host/sys/fs/cgroup:ro \
    -p 127.0.0.1:8126:8126/tcp \
    -e DD_API_KEY="<DATADOG_API_KEY>" \
    -e DD_APM_ENABLED=true \
    datadog/agent:latest
```

- Run example

```
python datadog_exporter.py
```

Auto-Instrumention Example

- Installation

```
pip install opentelemetry-api
pip install opentelemetry-sdk
pip install opentelemetry-ext-datadog
pip install opentelemetry-auto-instrumentation
pip install opentelemetry-ext-flask
pip install flask
pip install requests
```

- Start Datadog Agent

```
docker run --rm \
    -v /var/run/docker.sock:/var/run/docker.sock:ro \
    -v /proc/:/host/proc/:ro \
    -v /sys/fs/cgroup/:/host/sys/fs/cgroup:ro \
    -p 127.0.0.1:8126:8126/tcp \
    -e DD_API_KEY="<DATADOG_API_KEY>" \
    -e DD_APM_ENABLED=true \
    datadog/agent:latest
```

- Start server

```
opentelemetry-auto-instrumentation python server.py
```

- Run client

```
opentelemetry-auto-instrumentation python client.py testing
```

- Run client with parameter to raise error

```
opentelemetry-auto-instrumentation python client.py error
```

2.1.29 OpenTelemetry Django Instrumentation Example

This shows how to use `opentelemetry-ext-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-ext-django
$ pip install requests
```

Execution

Execution of the Django app

Set these environment variables first:

1. `export OPENTELEMETRY_PYTHON_DJANGO_INSTRUMENT=True`
2. `export DJANGO_SETTINGS_MODULE=instrumentation_example.settings`

The way to achieve OpenTelemetry instrumentation for your Django app is to use an `opentelemetry.ext.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`.

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/ext/opentelemetry-ext-django/example`, 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": {
        "canonical_code": "OK"
    },
    "attributes": {
        "component": "http",
        "http.method": "GET",
```

(continues on next page)

(continued from previous page)

```
        "http.server_name": "localhost",
        "http.scheme": "http",
        "host.port": 8000,
        "http.host": "localhost:8000",
        "http.url": "http://localhost:8000/?param=hello",
        "net.peer.ip": "127.0.0.1",
        "http.flavor": "1.1",
        "http.status_text": "OK",
        "http.status_code": 200
    },
    "events": [],
    "links": []
}
```

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

References

- Django
- OpenTelemetry Project
- OpenTelemetry Django extension

2.1.30 OpenTelemetry Collector Metrics OpenCensus Exporter Example

This example shows how to use the OpenCensus Exporter to export metrics 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-ext-opencensusexporter
```

Run the Example

Before running the example, it's necessary to run the OpenTelemetry collector and Prometheus. The docker folder contains the 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 metrics are available in the Prometheus dashboard at http://localhost:9090/graph, look for the "requests" metric on the list.

Useful links

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

2.1.31 OpenTelemetry Collector Tracer OpenCensus Exporter Example

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-ext-opencensusexporter
```

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*

2.1.32 OpenTelemetry Example Application

This package is a complete example of an application instrumented with OpenTelemetry.

The purpose is to provide a reference for consumers that demonstrates how to use the OpenTelemetry API and SDK in a variety of use cases and how to set it up.

References

- OpenTelemetry Project

2.1.33 OpenTracing Shim Example

This example shows how to use the *opentelemetry-ext-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-ext-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*

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Z