OpenTelemetry Python

OpenTelemetry Authors

Sep 29, 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 _beta_, and shouldn't generally be used in production environments.

CHAPTER

ONE

REQUIREMENT

OpenTelemetry-Python supports Python 3.6 and higher.

CHAPTER

TWO

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-exporter-{exporter}
pip install opentelemetry-instrumentation-{instrumentation}
```

These are for exporter and instrumentation packages respectively. The Jaeger, Zipkin, OTLP and OpenCensus Exporters can be found in the exporter directory of the repository. Instrumentations and additional exporters can be found in the Contrib repo instrumentation and Contrib repo exporter directories.

CHAPTER

THREE

EXTENSIONS

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

3.1 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
```

3.1.1 Getting Started with OpenTelemetry Python

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

For more elaborate examples, see examples.

Hello world: emit 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. The implementation is designed to be generic and extensible enough that in many situations, the SDK is sufficient.

Once installed, you can use 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, you can extract helpful information such as how long the action took. You can also add arbitrary attributes to the span that provide more insight for debugging.

The following example script emits a trace containing three named spans: “foo”, “bar”, and “baz”:

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

provider = TracerProvider()
processor = BatchSpanProcessor(ConsoleSpanExporter())
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)


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!")
```

When you run the script you can see the traces printed 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": {
        "status_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",
```

(continues on next page)

(continued from previous page)

```
    "status": {
        "status_code": "OK"
    },
    "attributes": {},
    "events": [],
    "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": {
        "status_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 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 this example, the OpenTelemetry Python library creates one trace containing three spans and prints it to STDOUT.

Configure exporters to emit spans elsewhere

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

This concept of aggregating span and trace information is known as distributed tracing. One such distributed tracing backend is known as Jaeger. The Jaeger project provides an all-in-one Docker container with a UI, database, and consumer.

Run the following command to start Jaeger:

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

This command starts Jaeger locally on port 16686 and exposes the Jaeger thrift agent on port 6831. You can visit Jaeger at http://localhost:16686.

After you spin up the backend, your application needs to export traces to this system. Although `opentelemetry-sdk` doesn't provide an exporter for Jaeger, you can install it as a separate package with the following command:

```
pip install opentelemetry-exporter-jaeger
```

After you install the exporter, update your code to import the Jaeger exporter and use that instead:

```
# jaeger_example.py
from opentelemetry import trace
from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.sdk.resources import SERVICE_NAME, Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

trace.set_tracer_provider(
    TracerProvider(
        resource=Resource.create({SERVICE_NAME: "my-helloworld-service"})
    )
)

jaeger_exporter = JaegerExporter(
    agent_host_name="localhost",
    agent_port=6831,
)

trace.get_tracer_provider().add_span_processor(
    BatchSpanProcessor(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!")
```

Finally, run the Python script:

```
python jaeger_example.py
```

You can then visit the Jaeger UI, see your service under "services", and find your traces!

Instrumentation example with Flask

While the example in the previous section is great, it's very manual. The following are common actions you might want to track and include as part of your distributed tracing.

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

To track these common actions, OpenTelemetry has the concept of instrumentations. Instrumentations are packages designed to interface with a specific framework or library, such as Flask and psycopg2. You can find a list of the currently curated extension packages in the Contrib repository.

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

```
pip install opentelemetry-instrumentation-flask
pip install opentelemetry-instrumentation-requests
```

The following small Flask application sends an HTTP request and also activates each instrumentation during its initialization:

```
# flask_example.py
import flask
import requests

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

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

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

tracer = trace.get_tracer(__name__)


@app.route("/")
def hello():
    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 script, hit the root URL (http://localhost:5000/) a few times, and watch your spans be emitted!

```
python flask_example.py
```

Configure Your HTTP propagator (b3, Baggage)

A major feature of distributed tracing is the ability to correlate a trace across multiple services. However, those services need to propagate information about a trace from one service to the other.

To enable this propagation, OpenTelemetry has the concept of propagators, which provide a common method to encode and decode span information from a request and response, respectively.

By default, `opentelemetry-python` is configured to use the W3C Trace Context and W3C Baggage HTTP headers for HTTP requests, but you can configure it to leverage different propagators. Here's an example using Zipkin's b3 propagation:

```
pip install opentelemetry-propagator-b3
```

Following the installation of the package containing the b3 propagator, configure the propagator as follows:

```
from opentelemetry.propagate import set_global_textmap
from opentelemetry.propagators.b3 import B3Format

set_global_textmap(B3Format())
```

Use the OpenTelemetry Collector for traces

Although it's possible to directly export your telemetry data to specific backends, you might have more complex use cases such as the following:

- A single telemetry sink shared by multiple services, to reduce overhead of switching exporters.
- Aggregating 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 data and export to multiple other backends, including to another instance of the Collector.

Start the Collector locally to see how the Collector works in practice. Write the following file:

```
# /tmp/otel-collector-config.yaml
receivers:
    otlp:
        protocols:
            grpc:
            http:
exporters:
    logging:
        loglevel: debug
processors:
    batch:
service:
    pipelines:
        traces:
            receivers: [otlp]
            exporters: [logging]
            processors: [batch]
```

Then start the Docker container:

```
docker run -p 4317:4317 \
    -v /tmp/otel-collector-config.yaml:/etc/otel-collector-config.yaml \
    otel/opentelemetry-collector:latest \
    --config=/etc/otel-collector-config.yaml
```

Install the OpenTelemetry Collector exporter:

```
pip install opentelemetry-exporter-otlp
```

Finally, execute the following script:

```
# otcollector.py
import time

from opentelemetry import trace
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import (
    OTLPSpanExporter,
)
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

span_exporter = OTLPSpanExporter(
    # optional
    # endpoint="myCollectorURL:4317",
    # credentials=ChannelCredentials(credentials),
    # headers=(("metadata", "metadata")),
)
tracer_provider = TracerProvider()
trace.set_tracer_provider(tracer_provider)
span_processor = BatchSpanProcessor(span_exporter)
tracer_provider.add_span_processor(span_processor)

# 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!")
```

3.1.2 Frequently Asked Questions and Cookbook

This page answers frequently asked questions, and serves as a cookbook for common scenarios.

Create a new span

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)
with tracer.start_as_current_span("print") as span:
    print("foo")
    span.set_attribute("printed_string", "foo")
```

Getting and modifying a span

```
from opentelemetry import trace

current_span = trace.get_current_span()
current_span.set_attribute("hometown", "seattle")
```

Capturing baggage at different contexts

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)
with tracer.start_as_current_span(name="root span") as root_span:
    parent_ctx = baggage.set_baggage("context", "parent")
    with tracer.start_as_current_span(
        name="child span", context=parent_ctx
    ) as child_span:
        child_ctx = baggage.set_baggage("context", "child")

print(baggage.get_baggage("context", parent_ctx))
print(baggage.get_baggage("context", child_ctx))
```

Manually setting span context

```
from opentelemetry import trace
from opentelemetry.trace import NonRecordingSpan, SpanContext, TraceFlags
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import ConsoleSpanExporter, BatchSpanProcessor
from opentelemetry.trace.propagation.tracecontext import TraceContextTextMapPropagator

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

tracer = trace.get_tracer(__name__)

# Extracting from carrier header
carrier = {'traceparent': '00-a9c3b99a95cc045e573e163c3ac80a77-d99d251a8caecd06-01'}
ctx = TraceContextTextMapPropagator().extract(carrier=carrier)
```

(continues on next page)

(continued from previous page)

```
with tracer.start_as_current_span('child', context=ctx) as span:
    span.set_attribute('primes', [2, 3, 5, 7])

# Or if you have a SpanContext object already.
span_context = SpanContext(
    trace_id=260450463492234107677662326386898679 7,
    span_id=5213367945872657620,
    is_remote=True,
    trace_flags=TraceFlags(0x01)
)
ctx = trace.set_span_in_context(NonRecordingSpan(span_context))

with tracer.start_as_current_span("child", context=ctx) as span:
    span.set_attribute('evens', [2, 4, 6, 8])
```

Using multiple tracer providers with different Resource

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace.export import ConsoleSpanExporter, BatchSpanProcessor

# Global tracer provider which can be set only once
trace.set_tracer_provider(
    TracerProvider(resource=Resource.create({"service.name": "service1"}))
)
trace.get_tracer_provider().add_span_processor(BatchSpanProcessor(ConsoleSpanExporter()))

tracer = trace.get_tracer(__name__)
with tracer.start_as_current_span("some-name") as span:
    span.set_attribute("key", "value")



another_tracer_provider = TracerProvider(
    resource=Resource.create({"service.name": "service2"})
)
another_tracer_provider.add_span_processor(BatchSpanProcessor(ConsoleSpanExporter()))

another_tracer = trace.get_tracer(__name__, tracer_provider=another_tracer_provider)
with another_tracer.start_as_current_span("name-here") as span:
    span.set_attribute("another-key", "another-value")
```

3.1.3 OpenTelemetry Python API

opentelemetry.baggage package

Subpackages

opentelemetry.baggage.propagation package

Module contents

class opentelemetry.baggage.propagation.**W3CBaggagePropagator**
Bases: opentelemetry.propagators.textmap.TextMapPropagator

Extracts and injects Baggage which is used to annotate telemetry.

extract(*carrier*, *context=None*, *getter=<opentelemetry.propagators.textmap.DefaultGetter object>*)
Extract Baggage from the carrier.

See opentelemetry.propagators.textmap.TextMapPropagator.extract

Return type *Context*

inject(*carrier*, *context=None*, *setter=<opentelemetry.propagators.textmap.DefaultSetter object>*)
Injects Baggage into the carrier.

See opentelemetry.propagators.textmap.TextMapPropagator.inject

Return type None

property fields: Set[str]
Returns a set with the fields set in *inject*.

Return type Set[str]

Module contents

opentelemetry.baggage.**get_all**(*context=None*)
Returns the name/value pairs in the Baggage

Parameters **context** (Optional[*Context*, None]) – The Context to use. If not set, uses current Context

Return type Mapping[str, object]

Returns The name/value pairs in the Baggage

opentelemetry.baggage.**get_baggage**(*name*, *context=None*)
Provides access to the value for a name/value pair in the Baggage

Parameters

- **name** (str) – The name of the value to retrieve
- **context** (Optional[*Context*, None]) – The Context to use. If not set, uses current Context

Return type Optional[object, None]

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

opentelemetry.baggage.**set_baggage**(*name*, *value*, *context=None*)
Sets a value in the Baggage

Parameters

- **name** (str) – The name of the value to set
- **value** (object) – The value to set
- **context** (Optional[*Context*, None]) – The Context to use. If not set, uses current Context

Return type *Context*

Returns A Context with the value updated

opentelemetry.baggage.**remove_baggage**(*name*, *context=None*)
Removes a value from the Baggage

Parameters

- **name** (str) – The name of the value to remove
- **context** (Optional[*Context*, None]) – The Context to use. If not set, uses current Context

Return type *Context*

Returns A Context with the name/value removed

opentelemetry.baggage.**clear**(*context=None*)
Removes all values from the Baggage

Parameters **context** (Optional[*Context*, None]) – The Context to use. If not set, uses current Context

Return type *Context*

Returns A Context with all baggage entries removed

opentelemetry.context package

Submodules

opentelemetry.context.base_context module

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

Module contents

opentelemetry.context.**create_key**(*keyname*)
To allow cross-cutting concern to control access to their local state, the RuntimeContext API provides a function which takes a keyname as input, and returns a unique key. :type keyname: str :param keyname: The key name is for debugging purposes and is not required to be unique.

Return type str

Returns A unique string representing the newly created key.

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*, None]) – 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*, None]) – 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.trace package

Submodules

opentelemetry.trace.status

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

Bases: enum.Enum

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

UNSET = 0

The default status.

OK = 1
The operation has been validated by an Application developer or Operator to have completed successfully.

ERROR = 2
The operation contains an error.

class opentelemetry.trace.status.**Status**(*status_code=StatusCode.UNSET*, *description=None*)
Bases: object

Represents the status of a finished Span.

Parameters

- **status_code** (*StatusCode*) – The canonical status code that describes the result status of the operation.
- **description** (Optional[str, None]) – An optional description of the status.

property status_code: opentelemetry.trace.status.StatusCode
Represents the canonical status code of a finished Span.

Return type *StatusCode*

property description: Optional[str]
Status description

Return type Optional[str, None]

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

Return type bool

property is_unset: bool
Returns true if unset, false otherwise.

Return type bool

opentelemetry.trace.span

class opentelemetry.trace.span.**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_span_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 *opentelemetry.trace.SpanContext* with a copy of this span's immutable state.

abstract set_attributes(*attributes*)
Sets Attributes.

Sets Attributes with the key and value passed as arguments dict.

Note: The behavior of `None` value attributes is undefined, and hence strongly discouraged.

Return type None

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

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

Note: The behavior of `None` value attributes is undefined, and hence strongly discouraged.

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

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

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

Return type None

abstract is_recording()
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.

Return type None

abstract record_exception(*exception*, *attributes=None*, *timestamp=None*, *escaped=False*)
Records an exception as a span event.

Return type None

class opentelemetry.trace.span.**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: bool

Return type bool

class opentelemetry.trace.span.**TraceState**(*entries=None*)

Bases: Mapping[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.

add(*key*, *value*)

Adds a key-value pair to tracestate. The provided pair should adhere to w3c tracestate identifiers format.

Parameters

- **key** (str) – A valid tracestate key to add
- **value** (str) – A valid tracestate value to add

Return type *TraceState*

Returns

A new TraceState with the modifications applied.

If the provided key-value pair is invalid or results in tracestate that violates tracecontext specification, they are discarded and same tracestate will be returned.

update(*key*, *value*)

Updates a key-value pair in tracestate. The provided pair should adhere to w3c tracestate identifiers format.

Parameters

- **key** (str) – A valid tracestate key to update
- **value** (str) – A valid tracestate value to update for key

Return type *TraceState*

Returns

A new TraceState with the modifications applied.

If the provided key-value pair is invalid or results in tracestate that violates tracecontext specification, they are discarded and same tracestate will be returned.

delete(*key*)

Deletes a key-value from tracestate.

Parameters **key** (str) – A valid tracestate key to remove key-value pair from tracestate

Return type *TraceState*

Returns

A new TraceState with the modifications applied.

If the provided key-value pair is invalid or results in tracestate that violates tracecontext specification, they are discarded and same tracestate will be returned.

to_header()
Creates a w3c tracestate header from a TraceState.

Return type str

Returns A string that adheres to the w3c tracestate header format.

classmethod from_header(*header_list*)
Parses one or more w3c tracestate header into a TraceState.

Parameters **header_list** (List[str]) – one or more w3c tracestate headers.

Return type *TraceState*

Returns

A valid TraceState that contains values extracted from the tracestate header.

If the format of one headers is illegal, all values will be discarded and an empty tracestate will be returned.

If the number of keys is beyond the maximum, all values will be discarded and an empty tracestate will be returned.

classmethod get_default()

Return type *TraceState*

keys()

Return type KeysView[str]

items()

Return type ItemsView[str, str]

values()

Return type ValuesView[str]

class opentelemetry.trace.span.**SpanContext**(*trace_id: int*, *span_id: int*, *is_remote: bool*, *trace_flags: Optional[*opentelemetry.trace.span.TraceFlags*] = 0*, *trace_state: Optional[*opentelemetry.trace.span.TraceState*] = []*)
Bases: Tuple[int, int, bool, *TraceFlags*, *TraceState*, bool]

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** – The ID of the trace that this span belongs to.
- **span_id** – This span's ID.
- **is_remote** – True if propagated from a remote parent.
- **trace_flags** – Trace options to propagate.
- **trace_state** – Tracing-system-specific info to propagate.

property trace_id: int

Return type int

property span_id: int

Return type int

property is_remote: bool

Return type bool

property trace_flags: opentelemetry.trace.span.TraceFlags

Return type *TraceFlags*

property trace_state: opentelemetry.trace.span.TraceState

Return type *TraceState*

property is_valid: bool

Return type bool

class opentelemetry.trace.span.**NonRecordingSpan**(*context*)

Bases: *opentelemetry.trace.span.Span*

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

All operations are no-op except context propagation.

get_span_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 *opentelemetry.trace.SpanContext* with a copy of this span's immutable state.

is_recording()

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_attributes(*attributes*)

Sets Attributes.

Sets Attributes with the key and value passed as arguments dict.

Note: The behavior of None value attributes is undefined, and hence strongly discouraged.

Return type None

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

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

Note: The behavior of `None` value attributes is undefined, and hence strongly discouraged.

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

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

This will override the name provided via *opentelemetry.trace.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.

Return type None

record_exception(*exception*, *attributes=None*, *timestamp=None*, *escaped=False*)
Records an exception as a span event.

Return type None

opentelemetry.trace.span.**format_trace_id**(*trace_id*)
Convenience trace ID formatting method :type trace_id: `int` :param trace_id: Trace ID int

Return type str

Returns The trace ID as 32-byte hexadecimal string

opentelemetry.trace.span.**format_span_id**(*span_id*)
Convenience span ID formatting method :type span_id: `int` :param span_id: Span ID int

Return type str

Returns The span ID as 16-byte hexadecimal string

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 *NonRecordingSpan* 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 is done via the Context
from opentelemetry.trace import set_span_in_context

context = set_span_in_context(parent)
child = tracer.start_span("child", context=context)

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.**NonRecordingSpan**(*context*)

Bases: *opentelemetry.trace.span.Span*

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

All operations are no-op except context propagation.

get_span_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 *opentelemetry.trace.SpanContext* with a copy of this span's immutable state.

is_recording()

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_attributes(*attributes*)
Sets Attributes.

Sets Attributes with the key and value passed as arguments dict.

Note: The behavior of None value attributes is undefined, and hence strongly discouraged.

Return type None

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

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

Note: The behavior of None value attributes is undefined, and hence strongly discouraged.

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

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

This will override the name provided via *opentelemetry.trace.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.

Return type None

record_exception(*exception*, *attributes=None*, *timestamp=None*, *escaped=False*)
Records an exception as a span event.

Return type None

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

A link to a *Span*. The attributes of a Link are immutable.

Parameters

- **context** (*SpanContext*) – *SpanContext* of the *Span* to link to.

- **attributes** (Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]) – Link's attributes.

property attributes: Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]

Return type Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]

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_span_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 *opentelemetry.trace.SpanContext* with a copy of this span's immutable state.

abstract set_attributes(*attributes*)

Sets Attributes.

Sets Attributes with the key and value passed as arguments dict.

Note: The behavior of None value attributes is undefined, and hence strongly discouraged.

Return type None

abstract set_attribute(*key*, *value*)

Sets an Attribute.

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

Note: The behavior of None value attributes is undefined, and hence strongly discouraged.

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 update_name(*name*)

Updates the *Span* name.

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

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

Return type None

abstract is_recording()
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.

Return type None

abstract record_exception(*exception*, *attributes=None*, *timestamp=None*, *escaped=False*)
Records an exception as a span event.

Return type None

class opentelemetry.trace.**SpanContext**(*trace_id: int*, *span_id: int*, *is_remote: bool*, *trace_flags: Optional[*opentelemetry.trace.span.TraceFlags*] = 0*, *trace_state: Optional[*opentelemetry.trace.span.TraceState*] = []*)
Bases: Tuple[int, int, bool, *TraceFlags*, *TraceState*, bool]

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** – The ID of the trace that this span belongs to.
- **span_id** – This span's ID.
- **is_remote** – True if propagated from a remote parent.
- **trace_flags** – Trace options to propagate.
- **trace_state** – Tracing-system-specific info to propagate.

property trace_id: int

Return type int

property span_id: int

Return type int

property is_remote: bool

Return type bool

property trace_flags: opentelemetry.trace.span.TraceFlags

Return type *TraceFlags*

property trace_state: opentelemetry.trace.span.TraceState

Return type *TraceState*

property is_valid: bool

Return type bool

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.**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: bool

Return type bool

class opentelemetry.trace.**TraceState**(*entries=None*)
Bases: Mapping[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.

add(*key*, *value*)
Adds a key-value pair to tracestate. The provided pair should adhere to w3c tracestate identifiers format.

Parameters

- **key** (str) – A valid tracestate key to add
- **value** (str) – A valid tracestate value to add

Return type *TraceState*

Returns

A new TraceState with the modifications applied.

If the provided key-value pair is invalid or results in tracestate that violates tracecontext specification, they are discarded and same tracestate will be returned.

update(*key*, *value*)
Updates a key-value pair in tracestate. The provided pair should adhere to w3c tracestate identifiers format.

Parameters

- **key** (str) – A valid tracestate key to update
- **value** (str) – A valid tracestate value to update for key

Return type *TraceState*

Returns

A new TraceState with the modifications applied.

If the provided key-value pair is invalid or results in tracestate that violates tracecontext specification, they are discarded and same tracestate will be returned.

delete(*key*)
Deletes a key-value from tracestate.

Parameters **key** (str) – A valid tracestate key to remove key-value pair from tracestate

Return type *TraceState*

Returns

A new TraceState with the modifications applied.

If the provided key-value pair is invalid or results in tracestate that violates tracecontext specification, they are discarded and same tracestate will be returned.

to_header()
Creates a w3c tracestate header from a TraceState.

Return type str

Returns A string that adheres to the w3c tracestate header format.

classmethod from_header(*header_list*)
Parses one or more w3c tracestate header into a TraceState.

Parameters **header_list** (List[str]) – one or more w3c tracestate headers.

Return type *TraceState*

Returns

A valid TraceState that contains values extracted from the tracestate header.

If the format of one headers is illegal, all values will be discarded and an empty tracestate will be returned.

If the number of keys is beyond the maximum, all values will be discarded and an empty tracestate will be returned.

classmethod get_default()

Return type *TraceState*

keys()

Return type KeysView[str]

items()

Return type ItemsView[str, str]

values()

Return type ValuesView[str]

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.instrumentation.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.

abstract start_span(*name*, *context=None*, *kind=SpanKind.INTERNAL*, *attributes=None*, *links=None*, *start_time=None*, *record_exception=True*, *set_status_on_exception=True*)

Starts a span.

Create a new span. Start the span without setting it as the current span in the context. To start the span and use the context in a single method, see *start_as_current_span()*.

By default the current span in the context will be used as parent, but an explicit context can also be specified, by passing in a *Context* containing a current *Span*. If there is no current span in the global *Context* or in the specified context, the created span will be a root span.

The span can be used as a context manager. On exiting the context manager, the span's end() method will be called.

Example:

```
# trace.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")
```

Parameters

- **name** (str) – The name of the span to be created.

- **context** (Optional[*Context*, None]) – An optional Context containing the span's parent. Defaults to the global context.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.
- **attributes** (Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]) – The span's attributes.
- **links** (Optional[Sequence[*Link*], None]) – Links span to other spans
- **start_time** (Optional[int, None]) – Sets the start time of a span
- **record_exception** (bool) – Whether to record any exceptions raised within the context as error event on the 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 ERROR when an uncaught exception is raised in the span with block. The span status won't be set by this mechanism if it was previously set manually.

Return type *Span*

Returns The newly-created span.

abstract start_as_current_span(*name*, *context=None*, *kind=SpanKind.INTERNAL*, *attributes=None*, *links=None*, *start_time=None*, *record_exception=True*, *set_status_on_exception=True*, *end_on_exit=True*)

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

Exiting the context manager will call the span's end method, as well as return the current span to its previous value by returning to the previous context.

Example:

```
with tracer.start_as_current_span("one") as parent:
    parent.add_event("parent's event")
    with trace.start_as_current_span("two") as child:
        child.add_event("child's event")
        trace.get_current_span()  # returns child
    trace.get_current_span()      # returns parent
trace.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 opentelemetry.trace.use_span(span, end_on_exit=True):
    do_work()
```

Parameters

- **name** (str) – The name of the span to be created.

- **context** (Optional[*Context*, None]) – An optional Context containing the span's parent. Defaults to the global context.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.
- **attributes** (Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]) – The span's attributes.
- **links** (Optional[Sequence[*Link*], None]) – Links span to other spans
- **start_time** (Optional[int, None]) – Sets the start time of a span
- **record_exception** (bool) – Whether to record any exceptions raised within the context as error event on the 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 ERROR when an uncaught exception is raised in the span with block. The span status won't be set by this mechanism if it was previously set manually.
- **end_on_exit** (bool) – Whether to end the span automatically when leaving the context manager.

Yields The newly-created span.

Return type Iterator[*Span*]

opentelemetry.trace.**format_span_id**(*span_id*)
Convenience span ID formatting method :type span_id: int :param span_id: Span ID int

Return type str

Returns The span ID as 16-byte hexadecimal string

opentelemetry.trace.**format_trace_id**(*trace_id*)
Convenience trace ID formatting method :type trace_id: int :param trace_id: Trace ID int

Return type str

Returns The trace ID as 32-byte hexadecimal string

opentelemetry.trace.**get_current_span**(*context=None*)
Retrieve the current span.

Parameters **context** (Optional[*Context*, None]) – A Context object. If one is not passed, the default current context is used instead.

Return type *Span*

Returns The Span set in the context if it exists. INVALID_SPAN otherwise.

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 omitted the current configured one is used.

Return type *Tracer*

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

Return type *TracerProvider*

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

This can only be done once, a warning will be logged if any furter attempt is made.

Return type None

opentelemetry.trace.**set_span_in_context**(*span*, *context=None*)
Set the span in the given context.

Parameters

- **span** (*Span*) – The Span to set.
- **context** (Optional[*Context*, None]) – a Context object. if one is not passed, the default current context is used instead.

Return type *Context*

opentelemetry.trace.**use_span**(*span*, *end_on_exit=False*, *record_exception=True*, *set_status_on_exception=True*)
Takes a non-active span and activates it in the current context.

Parameters

- **span** (*Span*) – The span that should be activated in the current context.
- **end_on_exit** (bool) – Whether to end the span automatically when leaving the context manager scope.
- **record_exception** (bool) – Whether to record any exceptions raised within the context as error event on the 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 ERROR when an uncaught exception is raised in the span with block. The span status won't be set by this mechanism if it was previously set manually.

Return type Iterator[*Span*]

class opentelemetry.trace.**Status**(*status_code=StatusCode.UNSET*, *description=None*)
Bases: object

Represents the status of a finished Span.

Parameters

- **status_code** (*StatusCode*) – The canonical status code that describes the result status of the operation.
- **description** (Optional[str, None]) – An optional description of the status.

property status_code: opentelemetry.trace.status.StatusCode
Represents the canonical status code of a finished Span.

Return type *StatusCode*

property description: Optional[str]
Status description

Return type Optional[str, None]

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

Return type bool

property is_unset: bool
Returns true if unset, false otherwise.

Return type bool

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

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

UNSET = 0
The default status.

OK = 1
The operation has been validated by an Application developer or Operator to have completed successfully.

ERROR = 2
The operation contains an error.

opentelemetry.environment_variables package

Module contents

opentelemetry.environment_variables.OTEL_PROPAGATORS = 'OTEL_PROPAGATORS'

OTEL_PROPAGATORS

opentelemetry.environment_variables.OTEL_PYTHON_CONTEXT = 'OTEL_PYTHON_CONTEXT'

OTEL_PYTHON_CONTEXT

opentelemetry.environment_variables.OTEL_PYTHON_DISABLED_INSTRUMENTATIONS = 'OTEL_PYTHON_DISABLED_INSTRUMENTATIONS'

OTEL_PYTHON_DISABLED_INSTRUMENTATIONS

opentelemetry.environment_variables.OTEL_PYTHON_ID_GENERATOR = 'OTEL_PYTHON_ID_GENERATOR'

OTEL_PYTHON_ID_GENERATOR

opentelemetry.environment_variables.OTEL_TRACES_EXPORTER = 'OTEL_TRACES_EXPORTER'

OTEL_TRACES_EXPORTER

opentelemetry.environment_variables.OTEL_PYTHON_TRACER_PROVIDER = 'OTEL_PYTHON_TRACER_PROVIDER'

OTEL_PYTHON_TRACER_PROVIDER

3.1.4 OpenTelemetry Python SDK

opentelemetry.sdk.resources package

This package implements OpenTelemetry Resources:

> *A Resource is an immutable representation of the entity producing telemetry. For example, a process producing telemetry that is running in a container on Kubernetes has a Pod name, it is in a namespace and possibly is part of a Deployment which also has a name. All three of these attributes can be included in the Resource.*

Resource objects are created with *Resource.create*, which accepts attributes (key-values). Resources should NOT be created via constructor, and working with *Resource* objects should only be done via the Resource API methods. Resource attributes can also be passed at process invocation in the *OTEL_RESOURCE_ATTRIBUTES* environment variable. You should register your resource with the *opentelemetry.sdk.trace.TracerProvider* by passing them into their constructors. The *Resource* passed to a provider is available to the exporter, which can send on this information as it sees fit.

```
trace.set_tracer_provider(
    TracerProvider(
        resource=Resource.create({
            "service.name": "shoppingcart",
            "service.instance.id": "instance-12",
        }),
    ),
)
print(trace.get_tracer_provider().resource.attributes)

{'telemetry.sdk.language': 'python',
'telemetry.sdk.name': 'opentelemetry',
'telemetry.sdk.version': '0.13.dev0',
'service.name': 'shoppingcart',
'service.instance.id': 'instance-12'}
```

Note that the OpenTelemetry project documents certain "standard attributes" that have prescribed semantic meanings, for example `service.name` in the above example.

class opentelemetry.sdk.resources.**Resource**(*attributes*, *schema_url=None*)

Bases: `object`

A Resource is an immutable representation of the entity producing telemetry as Attributes.

static create(*attributes=None*, *schema_url=None*)

Creates a new *Resource* from attributes.

Parameters

- **attributes** (`Optional[Dict[str, Union[str, bool, int, float]], None]`) – Optional zero or more key-value pairs.
- **schema_url** (`Optional[str, None]`) – Optional URL pointing to the schema

Return type *Resource*

Returns The newly-created Resource.

static get_empty()

Return type *Resource*

property attributes: Dict[str, Union[str, bool, int, float]]

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

property schema_url: str

Return type str

merge(*other*)

Merges this resource and an updating resource into a new *Resource*.

If a key exists on both the old and updating resource, the value of the updating resource will override the old resource value.

The updating resource's *schema_url* will be used only if the old *schema_url* is empty. Attempting to merge two resources with different, non-empty values for *schema_url* will result in an error and return the old resource.

Parameters **other** (*Resource*) – The other resource to be merged.

Return type *Resource*

Returns The newly-created Resource.

class opentelemetry.sdk.resources.**ResourceDetector**(*raise_on_error=False*)

Bases: abc.ABC

abstract detect()

Return type *Resource*

class opentelemetry.sdk.resources.**OTELResourceDetector**(*raise_on_error=False*)

Bases: *opentelemetry.sdk.resources.ResourceDetector*

detect()

Return type *Resource*

opentelemetry.sdk.resources.**get_aggregated_resources**(*detectors*, *initial_resource=None*, *timeout=5*)

Retrieves resources from detectors in the order that they were passed

Parameters

- **detectors** (List[*ResourceDetector*]) – List of resources in order of priority
- **initial_resource** (Optional[*Resource*, None]) – Static resource. This has highest priority
- **timeout** – Number of seconds to wait for each detector to return

Return type *Resource*

Returns

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 spans recorded in their own format.

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

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

Parameters **spans** (Sequence[*ReadableSpan*]) – 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.**SimpleSpanProcessor**(*span_exporter*)
Bases: *opentelemetry.sdk.trace.SpanProcessor*

Simple SpanProcessor implementation.

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

on_start(*span*, *parent_context=None*)
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.
- **parent_context** (Optional[*Context*, None]) – The parent context of the 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** (*ReadableSpan*) – 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.**BatchSpanProcessor**(*span_exporter*, *max_queue_size=None*, *schedule_delay_millis=None*, *max_export_batch_size=None*, *export_timeout_millis=None*)

Bases: *opentelemetry.sdk.trace.SpanProcessor*

Batch span processor implementation.

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

BatchSpanProcessor is configurable with the following environment variables which correspond to constructor parameters:

- *OTEL_BSP_SCHEDULE_DELAY*
- *OTEL_BSP_MAX_QUEUE_SIZE*
- *OTEL_BSP_MAX_EXPORT_BATCH_SIZE*
- *OTEL_BSP_EXPORT_TIMEOUT*

on_start(*span*, *parent_context=None*)
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.
- **parent_context** (Optional[*Context*, None]) – The parent context of the 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** (*ReadableSpan*) – The *opentelemetry.trace.Span* that just ended.

Return type None

worker()

force_flush(*timeout_millis=None*)
Export all ended spans to the configured Exporter that have not yet been exported.

Parameters **timeout_millis** (Optional[int, None]) – 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**(*service_name=None*, *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[*ReadableSpan*]) – The list of *opentelemetry.trace.Span* objects to be exported

Return type *SpanExportResult*

Returns The result of the export

opentelemetry.sdk.trace.id_generator

class opentelemetry.sdk.trace.id_generator.**IdGenerator**

Bases: abc.ABC

abstract generate_span_id()
Get a new span ID.

Return type int

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

abstract generate_trace_id()
Get a new trace ID.

Implementations should at least make the 64 least significant bits uniformly random. Samplers like the *TraceIdRatioBased* sampler rely on this randomness to make sampling decisions.

See the specification on TraceIdRatioBased.

Return type int

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

class opentelemetry.sdk.trace.id_generator.**RandomIdGenerator**

Bases: *opentelemetry.sdk.trace.id_generator.IdGenerator*

The default ID generator for TracerProvider which randomly generates all bits when generating IDs.

generate_span_id()

Get a new span ID.

Return type int

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

generate_trace_id()

Get a new trace ID.

Implementations should at least make the 64 least significant bits uniformly random. Samplers like the *TraceIdRatioBased* sampler rely on this randomness to make sampling decisions.

See the specification on TraceIdRatioBased.

Return type int

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

opentelemetry.sdk.trace.sampling

For general information about sampling, see the specification.

OpenTelemetry provides two types of samplers:

- *StaticSampler*
- *TraceIdRatioBased*

A *StaticSampler* always returns the same sampling result regardless of the conditions. Both possible StaticSamplers are already created:

- Always sample spans: ALWAYS_ON
- Never sample spans: ALWAYS_OFF

A *TraceIdRatioBased* sampler makes a random sampling result based on the sampling probability given.

If the span being sampled has a parent, *ParentBased* will respect the parent delegate sampler. Otherwise, it returns the sampling result from the given root sampler.

Currently, sampling results are always made during the creation of the span. However, this might not always be the case in the future (see OTEP #115).

Custom samplers can be created by subclassing *Sampler* and implementing *Sampler.should_sample* as well as *Sampler.get_description*.

Samplers are able to modify the *opentelemetry.trace.span.TraceState* of the parent of the span being created. For custom samplers, it is suggested to implement *Sampler.should_sample* to utilize the parent span context's *opentelemetry.trace.span.TraceState* and pass into the *SamplingResult* instead of the explicit trace_state field passed into the parameter of *Sampler.should_sample*.

To use a sampler, pass it into the tracer provider constructor. For example:

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    ConsoleSpanExporter,
    SimpleSpanProcessor,
)
from opentelemetry.sdk.trace.sampling import TraceIdRatioBased

# sample 1 in every 1000 traces
sampler = TraceIdRatioBased(1/1000)

# set the sampler onto the global tracer provider
trace.set_tracer_provider(TracerProvider(sampler=sampler))

# set up an exporter for sampled spans
trace.get_tracer_provider().add_span_processor(
    SimpleSpanProcessor(ConsoleSpanExporter())
)

# created spans will now be sampled by the TraceIdRatioBased sampler
with trace.get_tracer(__name__).start_as_current_span("Test Span"):
    ...
```

The tracer sampler can also be configured via environment variables `OTEL_TRACES_SAMPLER` and `OTEL_TRACES_SAMPLER_ARG` (only if applicable). The list of known values for `OTEL_TRACES_SAMPLER` are:

- always_on - Sampler that always samples spans, regardless of the parent span's sampling decision.
- always_off - Sampler that never samples spans, regardless of the parent span's sampling decision.
- traceidratio - Sampler that samples probabalistically based on rate.
- parentbased_always_on - (default) Sampler that respects its parent span's sampling decision, but otherwise always samples.
- parentbased_always_off - Sampler that respects its parent span's sampling decision, but otherwise never samples.
- parentbased_traceidratio - Sampler that respects its parent span's sampling decision, but otherwise samples probabalistically based on rate.

Sampling probability can be set with `OTEL_TRACES_SAMPLER_ARG` if the sampler is traceidratio or parentbased_traceidratio, when not provided rate will be set to 1.0 (maximum rate possible).

Prev example but with environment vairables. Please make sure to set the env `OTEL_TRACES_SAMPLER=traceidratio` and `OTEL_TRACES_SAMPLER_ARG=0.001`.

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    ConsoleSpanExporter,
    SimpleSpanProcessor,
)

trace.set_tracer_provider(TracerProvider())

# set up an exporter for sampled spans
trace.get_tracer_provider().add_span_processor(
```

(continues on next page)

(continued from previous page)

```
    SimpleSpanProcessor(ConsoleSpanExporter())
)

# created spans will now be sampled by the TraceIdRatioBased sampler with rate 1/1000.
with trace.get_tracer(__name__).start_as_current_span("Test Span"):
    ...
```

class opentelemetry.sdk.trace.sampling.**Decision**(*value*)

Bases: enum.Enum

An enumeration.

DROP = 0

RECORD_ONLY = 1

RECORD_AND_SAMPLE = 2

is_recording()

is_sampled()

class opentelemetry.sdk.trace.sampling.**SamplingResult**(*decision*, *attributes=None*, *trace_state=None*)

Bases: object

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

Parameters

- **decision** (*Decision*) – A sampling decision based off of whether the span is recorded and the sampled flag in trace flags in the span context.
- **attributes** (Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]) – Attributes to add to the *opentelemetry.trace.Span*.
- **trace_state** (Optional[*TraceState*, None]) – The tracestate used for the *opentelemetry.trace.Span*. Could possibly have been modified by the sampler.

class opentelemetry.sdk.trace.sampling.**Sampler**

Bases: abc.ABC

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

Return type *SamplingResult*

abstract get_description()

Return type str

class opentelemetry.sdk.trace.sampling.**StaticSampler**(*decision*)

Bases: *opentelemetry.sdk.trace.sampling.Sampler*

Sampler that always returns the same decision.

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

Return type *SamplingResult*

get_description()

Return type str

opentelemetry.sdk.trace.sampling.**ALWAYS_OFF** = <opentelemetry.sdk.trace.sampling.StaticSampler object>
Sampler that never samples spans, regardless of the parent span's sampling decision.

opentelemetry.sdk.trace.sampling.**ALWAYS_ON** = <opentelemetry.sdk.trace.sampling.StaticSampler object>
Sampler that always samples spans, regardless of the parent span's sampling decision.

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

Sampler that makes sampling decisions probabalistically based on *rate*, while also respecting the parent span sampling decision.

Parameters **rate** (float) – Probability (between 0 and 1) that a span will be sampled

TRACE_ID_LIMIT = 18446744073709551615

classmethod get_bound_for_rate(*rate*)

Return type int

property rate: float

Return type float

property bound: int

Return type int

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

Return type *SamplingResult*

get_description()

Return type str

class opentelemetry.sdk.trace.sampling.**ParentBased**(*root*, *remote_parent_sampled=<opentelemetry.sdk.trace.sampling.StaticSa object>*, *remote_parent_not_sampled=<opentelemetry.sdk.trace.sampling.Stati object>*, *local_parent_sampled=<opentelemetry.sdk.trace.sampling.StaticSamp object>*, *local_parent_not_sampled=<opentelemetry.sdk.trace.sampling.Static object>*)
Bases: *opentelemetry.sdk.trace.sampling.Sampler*

If a parent is set, applies the respective delegate sampler. Otherwise, uses the root provided at initialization to make a decision.

Parameters

- **root** (*Sampler*) – Sampler called for spans with no parent (root spans).

- **remote_parent_sampled** (*Sampler*) – Sampler called for a remote sampled parent.
- **remote_parent_not_sampled** (*Sampler*) – Sampler called for a remote parent that is not sampled.
- **local_parent_sampled** (*Sampler*) – Sampler called for a local sampled parent.
- **local_parent_not_sampled** (*Sampler*) – Sampler called for a local parent that is not sampled.

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

Return type *SamplingResult*

get_description()

opentelemetry.sdk.trace.sampling.**DEFAULT_OFF** = <opentelemetry.sdk.trace.sampling.ParentBased object>
Sampler that respects its parent span's sampling decision, but otherwise never samples.

opentelemetry.sdk.trace.sampling.**DEFAULT_ON** = <opentelemetry.sdk.trace.sampling.ParentBased object>
Sampler that respects its parent span's sampling decision, but otherwise always samples.

class opentelemetry.sdk.trace.sampling.**ParentBasedTraceIdRatio**(*rate*)
Bases: *opentelemetry.sdk.trace.sampling.ParentBased*

Sampler that respects its parent span's sampling decision, but otherwise samples probabalistically based on *rate*.

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* for the meaning of these properties.

property version: str

Return type str

property name: str

Return type str

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*, *parent_context=None*)
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.

- **parent_context** (Optional[*Context*, None]) – The parent context of the 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** (*ReadableSpan*) – 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.**SynchronousMultiSpanProcessor**
Bases: *opentelemetry.sdk.trace.SpanProcessor*

Implementation of class:*SpanProcessor* that forwards all received events to a list of span processors sequentially.

The underlying span processors are called in sequential order as they were added.

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

Return type None

on_start(*span*, *parent_context=None*)
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.
- **parent_context** (Optional[*Context*, None]) – The parent context of the 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** (*ReadableSpan*) – The *opentelemetry.trace.Span* that just ended.

Return type None

shutdown()
Sequentially shuts down all underlying span processors.

Return type None

force_flush(*timeout_millis=30000*)
Sequentially calls force_flush on all underlying *SpanProcessor*

Parameters **timeout_millis** (int) – The maximum amount of time over all span processors to wait for spans to be exported. In case the first n span processors exceeded the timeout followup span processors will be skipped.

Return type bool

Returns True if all span processors flushed their spans within the given timeout, False otherwise.

class opentelemetry.sdk.trace.**ConcurrentMultiSpanProcessor**(*num_threads=2*)
Bases: *opentelemetry.sdk.trace.SpanProcessor*

Implementation of *SpanProcessor* that forwards all received events to a list of span processors in parallel.

Calls to the underlying span processors are forwarded in parallel by submitting them to a thread pool executor and waiting until each span processor finished its work.

Parameters **num_threads** (int) – The number of threads managed by the thread pool executor and thus defining how many span processors can work in parallel.

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

Return type None

on_start(*span*, *parent_context=None*)
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.
- **parent_context** (Optional[*Context*, None]) – The parent context of the 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** (*ReadableSpan*) – The *opentelemetry.trace.Span* that just ended.

Return type None

shutdown()
Shuts down all underlying span processors in parallel.

Return type None

force_flush(*timeout_millis=30000*)
Calls force_flush on all underlying span processors in parallel.

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

Return type bool

Returns True if all span processors flushed their spans within the given timeout, False otherwise.

class opentelemetry.sdk.trace.**EventBase**(*name*, *timestamp=None*)

Bases: abc.ABC

property **name: str**

Return type str

property **timestamp: int**

Return type int

abstract property **attributes: Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]**

Return type Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]

class opentelemetry.sdk.trace.**Event**(*name*, *attributes=None*, *timestamp=None*, *limit=128*)

Bases: *opentelemetry.sdk.trace.EventBase*

A text annotation with a set of attributes. The attributes of an event are immutable.

Parameters

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

property **attributes: Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]**

Return type Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]

class opentelemetry.sdk.trace.**ReadableSpan**(*name=None*, *context=None*, *parent=None*, *resource=<opentelemetry.sdk.resources.Resource object>*, *attributes=None*, *events=None*, *links=()*, *kind=SpanKind.INTERNAL*, *instrumentation_info=None*, *status=<opentelemetry.trace.status.Status object>*, *start_time=None*, *end_time=None*)

Bases: object

Provides read-only access to span attributes

property **dropped_attributes: int**

Return type int

property **dropped_events: int**

Return type int

property **dropped_links: int**

Return type int

property name: str

Return type str

get_span_context()

property context

property kind: opentelemetry.trace.SpanKind

Return type *SpanKind*

property parent: Optional[opentelemetry.trace.span.SpanContext]

Return type Optional[*SpanContext*, None]

property start_time: Optional[int]

Return type Optional[int, None]

property end_time: Optional[int]

Return type Optional[int, None]

property status: opentelemetry.trace.status.Status

Return type *Status*

property attributes: Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]]]

Return type Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]

property events: Sequence[opentelemetry.sdk.trace.Event]

Return type Sequence[*Event*]

property links: Sequence[opentelemetry.trace.Link]

Return type Sequence[*Link*]

property resource: opentelemetry.sdk.resources.Resource

Return type *Resource*

property instrumentation_info: opentelemetry.sdk.util.instrumentation.InstrumentationInfo

Return type *InstrumentationInfo*

to_json(*indent=4*)

class opentelemetry.sdk.trace.**SpanLimits**(*max_attributes=None*, *max_events=None*, *max_links=None*, *max_event_attributes=None*, *max_link_attributes=None*)

Bases: object

The limits that should be enforce on recorded data such as events, links, attributes etc.

This class does not enforce any limits itself. It only provides an a way read limits from env, default values and from user provided arguments.

All limit arguments must be either a non-negative integer, None or SpanLimits.UNSET.

- All limit arguments are optional.
- If a limit argument is not set, the class will try to read its value from the corresponding environment variable.
- If the environment variable is not set, the default value for the limit is used.

Parameters

- **max_attributes** (Optional[int, None]) – Maximum number of attributes that can be added to a Span. Environment variable: OTEL_SPAN_ATTRIBUTE_COUNT_LIMIT Default: {_DEFAULT_SPAN_ATTRIBUTE_COUNT_LIMIT}
- **max_events** (Optional[int, None]) – Maximum number of events that can be added to a Span. Environment variable: OTEL_SPAN_EVENT_COUNT_LIMIT Default: {_DEFAULT_SPAN_EVENT_COUNT_LIMIT}
- **max_links** (Optional[int, None]) – Maximum number of links that can be added to a Span. Environment variable: OTEL_SPAN_LINK_COUNT_LIMIT Default: {_DEFAULT_SPAN_LINK_COUNT_LIMIT}
- **max_event_attributes** (Optional[int, None]) – Maximum number of attributes that can be added to an Event. Default: {_DEFAULT_OTEL_EVENT_ATTRIBUTE_COUNT_LIMIT}
- **max_link_attributes** (Optional[int, None]) – Maximum number of attributes that can be added to a Link. Default: {_DEFAULT_OTEL_LINK_ATTRIBUTE_COUNT_LIMIT}

UNSET = -1

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*, *span_processor=<opentelemetry.sdk.trace.SpanProcessor object>*, *instrumentation_info=None*, *record_exception=True*, *set_status_on_exception=True*, *limits=SpanLimits(max_attributes=None, max_events=None, max_links=None, max_event_attributes=None, max_link_attributes=None)*)

Bases: *opentelemetry.trace.span.Span*, *opentelemetry.sdk.trace.ReadableSpan*

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*, None]) – This span's parent's *opentelemetry.trace.SpanContext*, or None if this is a root span
- **sampler** (Optional[*Sampler*, None]) – The sampler used to create this span
- **trace_config** (None) – TODO
- **resource** (*Resource*) – Entity producing telemetry
- **attributes** (Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]) – The span's attributes to be exported
- **events** (Optional[Sequence[*Event*], None]) – 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*.

- **limits** – *SpanLimits* instance that was passed to the *TracerProvider*

get_span_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 *opentelemetry.trace.SpanContext* with a copy of this span's immutable state.

set_attributes(*attributes*)
Sets Attributes.

Sets Attributes with the key and value passed as arguments dict.

Note: The behavior of `None` value attributes is undefined, and hence strongly discouraged.

Return type None

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

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

Note: The behavior of `None` value attributes is undefined, and hence strongly discouraged.

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

start(*start_time=None*, *parent_context=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(**args*, ***kwargs*)
Updates the *Span* name.

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

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

is_recording()
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(**args*, ***kwargs*)
Sets the Status of the Span. If used, this will override the default Span status.

record_exception(*exception*, *attributes=None*, *timestamp=None*, *escaped=False*)
Records an exception as a span event.

Return type None

class opentelemetry.sdk.trace.**Tracer**(*sampler*, *resource*, *span_processor*, *id_generator*, *instrumentation_info*, *span_limits*)

Bases: *opentelemetry.trace.Tracer*

See *opentelemetry.trace.Tracer*.

start_as_current_span(*name*, *context=None*, *kind=SpanKind.INTERNAL*, *attributes=None*, *links=()*, *start_time=None*, *record_exception=True*, *set_status_on_exception=True*, *end_on_exit=True*)
Context manager for creating a new span and set it as the current span in this tracer's context.

Exiting the context manager will call the span's end method, as well as return the current span to its previous value by returning to the previous context.

Example:

```
with tracer.start_as_current_span("one") as parent:
    parent.add_event("parent's event")
    with trace.start_as_current_span("two") as child:
        child.add_event("child's event")
        trace.get_current_span()  # returns child
    trace.get_current_span()      # returns parent
trace.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 opentelemetry.trace.use_span(span, end_on_exit=True):
    do_work()
```

Parameters

- **name** (str) – The name of the span to be created.
- **context** (Optional[*Context*, None]) – An optional Context containing the span's parent. Defaults to the global context.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.
- **attributes** (Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]) – The span's attributes.
- **links** (Sequence[*Link*]) – Links span to other spans
- **start_time** (Optional[int, None]) – Sets the start time of a span
- **record_exception** (bool) – Whether to record any exceptions raised within the context as error event on the 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 ERROR when an uncaught exception is raised in the span with block. The span status won't be set by this mechanism if it was previously set manually.
- **end_on_exit** (bool) – Whether to end the span automatically when leaving the context manager.

Yields The newly-created span.

Return type Iterator[*Span*]

start_span(*name*, *context=None*, *kind=SpanKind.INTERNAL*, *attributes=None*, *links=()*, *start_time=None*, *record_exception=True*, *set_status_on_exception=True*)

Starts a span.

Create a new span. Start the span without setting it as the current span in the context. To start the span and use the context in a single method, see *start_as_current_span()*.

By default the current span in the context will be used as parent, but an explicit context can also be specified, by passing in a *Context* containing a current *Span*. If there is no current span in the global *Context* or in the specified context, the created span will be a root span.

The span can be used as a context manager. On exiting the context manager, the span's end() method will be called.

Example:

```
# trace.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")
```

Parameters

- **name** (str) – The name of the span to be created.
- **context** (Optional[*Context*, None]) – An optional Context containing the span's parent. Defaults to the global context.
- **kind** (*SpanKind*) – The span's kind (relationship to parent). Note that is meaningful even if there is no parent.
- **attributes** (Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]) – The span's attributes.
- **links** (Sequence[*Link*]) – Links span to other spans
- **start_time** (Optional[int, None]) – Sets the start time of a span
- **record_exception** (bool) – Whether to record any exceptions raised within the context as error event on the 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 ERROR when an uncaught exception is raised in the span with block. The span status won't be set by this mechanism if it was previously set manually.

Return type *Span*

Returns The newly-created span.

class opentelemetry.sdk.trace.**TracerProvider**(*sampler=<opentelemetry.sdk.trace.sampling.ParentBased object>*, *resource=<opentelemetry.sdk.resources.Resource object>*, *shutdown_on_exit=True*, *active_span_processor=None*, *id_generator=None*, *span_limits=None*)

Bases: *opentelemetry.trace.TracerProvider*

See *opentelemetry.trace.TracerProvider*.

property resource: opentelemetry.sdk.resources.Resource

Return type *Resource*

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.instrumentation.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*

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.

force_flush(*timeout_millis=30000*)

Requests the active span processor to process all spans that have not yet been processed.

By default force flush is called sequentially on all added span processors. This means that span processors further back in the list have less time to flush their spans. To have span processors flush their spans in parallel it is possible to initialize the tracer provider with an instance of *ConcurrentMultiSpanProcessor* at the cost of using multiple threads.

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

Return type bool

Returns False if the timeout is exceeded, True otherwise.

opentelemetry.sdk.error_handler package

Global Error Handler

This module provides a global error handler and an interface that allows error handlers to be registered with the global error handler via entry points. A default error handler is also provided.

To use this feature, users can create an error handler that is registered using the `opentelemetry_error_handler` entry point. A class is to be registered in this entry point, this class must inherit from the `opentelemetry.sdk.error_handler.ErrorHandler` class and implement the corresponding `handle` method. This method will receive the exception object that is to be handled. The error handler class should also inherit from the exception classes it wants to handle. For example, this would be an error handler that handles `ZeroDivisionError`:

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

logger = getLogger(__name__)


class ErrorHandler0(ErrorHandler, ZeroDivisionError):

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

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

To use the global error handler, just instantiate it as a context manager where you want exceptions to be handled:

```
from opentelemetry.sdk.error_handler import GlobalErrorHandler

with GlobalErrorHandler():
    1 / 0
```

If the class of the exception raised in the scope of the `GlobalErrorHandler` object is not parent of any registered error handler, then the default error handler will handle the exception. This default error handler will only log the exception to standard logging, the exception won't be raised any further.

class opentelemetry.sdk.error_handler.**ErrorHandler**
Bases: `abc.ABC`

class opentelemetry.sdk.error_handler.**GlobalErrorHandler**
Bases: `object`

Global error handler

This is a singleton class that can be instantiated anywhere to get the global error handler. This object provides a handle method that receives an exception object that will be handled by the registered error handlers.

opentelemetry.sdk.environment_variables

opentelemetry.sdk.environment_variables.**OTEL_RESOURCE_ATTRIBUTES = 'OTEL_RESOURCE_ATTRIBUTES'**

OTEL_RESOURCE_ATTRIBUTES

The *OTEL_RESOURCE_ATTRIBUTES* environment variable allows resource attributes to be passed to the SDK at process invocation. The attributes from *OTEL_RESOURCE_ATTRIBUTES* are merged with those passed to *Resource.create*, meaning *OTEL_RESOURCE_ATTRIBUTES* takes *lower* priority. Attributes should be in the format `key1=value1,key2=value2`. Additional details are available in the specification.

```
$ OTEL_RESOURCE_ATTRIBUTES="service.name=shoppingcard,will_be_overridden=foo"␣
↪python - <<EOF
import pprint
from opentelemetry.sdk.resources import Resource
pprint.pprint(Resource.create({"will_be_overridden": "bar"}).attributes)
EOF
{'service.name': 'shoppingcard',
'telemetry.sdk.language': 'python',
'telemetry.sdk.name': 'opentelemetry',
'telemetry.sdk.version': '0.13.dev0',
'will_be_overridden': 'bar'}
```

opentelemetry.sdk.environment_variables.**OTEL_LOG_LEVEL = 'OTEL_LOG_LEVEL'**

OTEL_LOG_LEVEL

The *OTEL_LOG_LEVEL* environment variable sets the log level used by the SDK logger Default: “info”

opentelemetry.sdk.environment_variables.**OTEL_TRACES_SAMPLER = 'OTEL_TRACES_SAMPLER'**

OTEL_TRACES_SAMPLER

The *OTEL_TRACES_SAMPLER* environment variable sets the sampler to be used for traces. Sampling is a mechanism to control the noise introduced by OpenTelemetry by reducing the number of traces collected and sent to the backend Default: “parentbased_always_on”

opentelemetry.sdk.environment_variables.**OTEL_TRACES_SAMPLER_ARG = 'OTEL_TRACES_SAMPLER_ARG'**

OTEL_TRACES_SAMPLER_ARG

The *OTEL_TRACES_SAMPLER_ARG* environment variable will only be used if OTEL_TRACES_SAMPLER is set. Each Sampler type defines its own expected input, if any. Invalid or unrecognized input is ignored, i.e. the SDK behaves as if OTEL_TRACES_SAMPLER_ARG is not set.

opentelemetry.sdk.environment_variables.**OTEL_BSP_SCHEDULE_DELAY = 'OTEL_BSP_SCHEDULE_DELAY'**

OTEL_BSP_SCHEDULE_DELAY

The *OTEL_BSP_SCHEDULE_DELAY* represents the delay interval between two consecutive exports. Default: 5000

opentelemetry.sdk.environment_variables.**OTEL_BSP_EXPORT_TIMEOUT = 'OTEL_BSP_EXPORT_TIMEOUT'**

OTEL_BSP_EXPORT_TIMEOUT

The *OTEL_BSP_EXPORT_TIMEOUT* represents the maximum allowed time to export data. Default: 30000

opentelemetry.sdk.environment_variables.**OTEL_BSP_MAX_QUEUE_SIZE = 'OTEL_BSP_MAX_QUEUE_SIZE'**

OTEL_BSP_MAX_QUEUE_SIZE

The *OTEL_BSP_MAX_QUEUE_SIZE* represents the maximum queue size for the data export. Default: 2048

opentelemetry.sdk.environment_variables.**OTEL_BSP_MAX_EXPORT_BATCH_SIZE = 'OTEL_BSP_MAX_EXPORT_BATCH_SIZE'**

OTEL_BSP_MAX_EXPORT_BATCH_SIZE

The *OTEL_BSP_MAX_EXPORT_BATCH_SIZE* represents the maximum batch size for the data export. Default: 512

opentelemetry.sdk.environment_variables.**OTEL_SPAN_ATTRIBUTE_COUNT_LIMIT = 'OTEL_SPAN_ATTRIBUTE_COUNT_LIMIT'**

OTEL_SPAN_ATTRIBUTE_COUNT_LIMIT

The *OTEL_SPAN_ATTRIBUTE_COUNT_LIMIT* represents the maximum allowed span attribute count. Default: 128

opentelemetry.sdk.environment_variables.**OTEL_SPAN_EVENT_COUNT_LIMIT = 'OTEL_SPAN_EVENT_COUNT_LIMIT'**

OTEL_SPAN_EVENT_COUNT_LIMIT

The *OTEL_SPAN_EVENT_COUNT_LIMIT* represents the maximum allowed span event count. Default: 128

opentelemetry.sdk.environment_variables.**OTEL_SPAN_LINK_COUNT_LIMIT = 'OTEL_SPAN_LINK_COUNT_LIMIT'**

OTEL_SPAN_LINK_COUNT_LIMIT

The *OTEL_SPAN_LINK_COUNT_LIMIT* represents the maximum allowed span link count. Default: 128

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_JAEGER_AGENT_HOST = 'OTEL_EXPORTER_JAEGER_AGENT_HOST'**

OTEL_EXPORTER_JAEGER_AGENT_HOST

The *OTEL_EXPORTER_JAEGER_AGENT_HOST* represents the hostname for the Jaeger agent. Default: "localhost"

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_JAEGER_AGENT_PORT = 'OTEL_EXPORTER_JAEGER_AGENT_PORT'**

OTEL_EXPORTER_JAEGER_AGENT_PORT

The *OTEL_EXPORTER_JAEGER_AGENT_PORT* represents the port for the Jaeger agent. Default: 6831

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_JAEGER_ENDPOINT = 'OTEL_EXPORTER_JAEGER_ENDPOINT'**

OTEL_EXPORTER_JAEGER_ENDPOINT

The *OTEL_EXPORTER_JAEGER_ENDPOINT* represents the HTTP endpoint for Jaeger traces. Default: "http://localhost:14250"

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_JAEGER_USER = 'OTEL_EXPORTER_JAEGER_USER'**

OTEL_EXPORTER_JAEGER_USER

The *OTEL_EXPORTER_JAEGER_USER* represents the username to be used for HTTP basic authentication.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_JAEGER_PASSWORD = 'OTEL_EXPORTER_JAEGER_PASSWORD'**

OTEL_EXPORTER_JAEGER_PASSWORD

The *OTEL_EXPORTER_JAEGER_PASSWORD* represents the password to be used for HTTP basic authentication.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_JAEGER_TIMEOUT = 'OTEL_EXPORTER_JAEGER_TIMEOUT'**

OTEL_EXPORTER_JAEGER_TIMEOUT

Maximum time the Jaeger exporter will wait for each batch export. Default: 10

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_ZIPKIN_ENDPOINT = 'OTEL_EXPORTER_ZIPKIN_ENDPOINT'**

OTEL_EXPORTER_ZIPKIN_ENDPOINT

Zipkin collector endpoint to which the exporter will send data. This may include a path (e.g. `http://example.com:9411/api/v2/spans`).

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_ZIPKIN_TIMEOUT = 'OTEL_EXPORTER_ZIPKIN_TIMEOUT'**

OTEL_EXPORTER_ZIPKIN_TIMEOUT

Maximum time (in seconds) the Zipkin exporter will wait for each batch export. Default: 10

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_PROTOCOL = 'OTEL_EXPORTER_OTLP_PROTOCOL'**

OTEL_EXPORTER_OTLP_PROTOCOL

The *OTEL_EXPORTER_OTLP_PROTOCOL* represents the the transport protocol for the OTLP exporter.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_CERTIFICATE = 'OTEL_EXPORTER_OTLP_CERTIFICATE'**

OTEL_EXPORTER_OTLP_CERTIFICATE

The *OTEL_EXPORTER_OTLP_CERTIFICATE* stores the path to the certificate file for TLS credentials of gRPC client. Should only be used for a secure connection.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_HEADERS = 'OTEL_EXPORTER_OTLP_HEADERS'**

OTEL_EXPORTER_OTLP_HEADERS

The *OTEL_EXPORTER_OTLP_HEADERS* contains the key-value pairs to be used as headers associated with gRPC or HTTP requests.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_COMPRESSION = 'OTEL_EXPORTER_OTLP_COMPRESSION'**

OTEL_EXPORTER_OTLP_COMPRESSION

Specifies a gRPC compression method to be used in the OTLP exporters. Possible values are:

- `gzip` corresponding to `grpc.Compression.Gzip`.

- `deflate` corresponding to `grpc.Compression.Deflate`.

If no `OTEL_EXPORTER_OTLP_*COMPRESSION` environment variable is present or `compression` argument passed to the exporter, the default `grpc.Compression.NoCompression` will be used. Additional details are available in the specification.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_TIMEOUT = 'OTEL_EXPORTER_OTLP_TIMEOUT'**

OTEL_EXPORTER_OTLP_TIMEOUT

The *OTEL_EXPORTER_OTLP_TIMEOUT* is the maximum time the OTLP exporter will wait for each batch export. Default: 10

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_ENDPOINT = 'OTEL_EXPORTER_OTLP_ENDPOINT'**

OTEL_EXPORTER_OTLP_ENDPOINT

The *OTEL_EXPORTER_OTLP_ENDPOINT* target to which the exporter is going to send spans or metrics. The endpoint MUST be a valid URL with scheme (http or https) and host, and MAY contain a port and path. A scheme of https indicates a secure connection. Default: "https://localhost:4317"

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_TRACES_ENDPOINT = 'OTEL_EXPORTER_OTLP_TRACES_ENDPOINT'**

OTEL_EXPORTER_OTLP_TRACES_ENDPOINT

The *OTEL_EXPORTER_OTLP_TRACES_ENDPOINT* target to which the span exporter is going to send spans. The endpoint MUST be a valid URL with scheme (http or https) and host, and MAY contain a port and path. A scheme of https indicates a secure connection.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_TRACES_PROTOCOL = 'OTEL_EXPORTER_OTLP_TRACES_PROTOCOL'**

OTEL_EXPORTER_OTLP_TRACES_PROTOCOL

The *OTEL_EXPORTER_OTLP_TRACES_PROTOCOL* represents the the transport protocol for spans.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_TRACES_CERTIFICATE = 'OTEL_EXPORTER_OTLP_TRACES_CERTIFICATE'**

OTEL_EXPORTER_OTLP_TRACES_CERTIFICATE

The *OTEL_EXPORTER_OTLP_TRACES_CERTIFICATE* stores the path to the certificate file for TLS credentials of gRPC client for traces. Should only be used for a secure connection for tracing.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_TRACES_HEADERS = 'OTEL_EXPORTER_OTLP_TRACES_HEADERS'**

OTEL_EXPORTER_OTLP_TRACES_HEADERS

The *OTEL_EXPORTER_OTLP_TRACES_HEADERS* contains the key-value pairs to be used as headers for spans associated with gRPC or HTTP requests.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_TRACES_COMPRESSION = 'OTEL_EXPORTER_OTLP_TRACES_COMPRESSION'**

OTEL_EXPORTER_OTLP_TRACES_COMPRESSION

Same as *OTEL_EXPORTER_OTLP_COMPRESSION* but only for the span exporter. If both are present, this takes higher precendence.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_TRACES_TIMEOUT = 'OTEL_EXPORTER_OTLP_TRACES_TIMEOUT'**

OTEL_EXPORTER_OTLP_TRACES_TIMEOUT

The *OTEL_EXPORTER_OTLP_TRACES_TIMEOUT* is the maximum time the OTLP exporter will wait for each batch export for spans.

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_JAEGER_CERTIFICATE = 'OTEL_EXPORTER_JAEGER_CERTIFICATE'**

OTEL_EXPORTER_JAEGER_CERTIFICATE

The *OTEL_EXPORTER_JAEGER_CERTIFICATE* stores the path to the certificate file for TLS credentials of gRPC client for Jaeger. Should only be used for a secure connection with Jaeger.

opentelemetry.sdk.environment_variables.
OTEL_EXPORTER_JAEGER_AGENT_SPLIT_OVERSIZED_BATCHES = 'OTEL_EXPORTER_JAEGER_AGENT_SPLIT_OVERSIZED_BATCHES'

OTEL_EXPORTER_JAEGER_AGENT_SPLIT_OVERSIZED_BATCHES

The *OTEL_EXPORTER_JAEGER_AGENT_SPLIT_OVERSIZED_BATCHES* is a boolean flag to determine whether to split a large span batch to admire the udp packet size limit.

opentelemetry.sdk.environment_variables.**OTEL_SERVICE_NAME = 'OTEL_SERVICE_NAME'**

OTEL_SERVICE_NAME

Convenience environment variable for setting the service name resource attribute. The following two environment variables have the same effect

```
OTEL_SERVICE_NAME=my-python-service

OTEL_RESOURCE_ATTRIBUTES=service.name=my-python-service
```

If both are set, *OTEL_SERVICE_NAME* takes precedence.

3.1.5 OpenTelemetry Jaeger Exporters

OpenTelemetry Jaeger Thrift Exporter

The **OpenTelemetry Jaeger Thrift Exporter** allows to export OpenTelemetry traces to Jaeger. This exporter always sends traces to the configured agent using the Thrift compact protocol over UDP. When it is not feasible to deploy Jaeger Agent next to the application, for example, when the application code is running as Lambda function, a collector can be configured to send spans using Thrift over HTTP. If both agent and collector are configured, the exporter sends traces only to the collector to eliminate the duplicate entries.

Usage

```
from opentelemetry import trace
from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

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

# create a JaegerExporter
jaeger_exporter = JaegerExporter(
    # configure agent
    agent_host_name='localhost',
    agent_port=6831,
    # optional: configure also collector
    # collector_endpoint='http://localhost:14268/api/traces?format=jaeger.thrift',
    # username=xxxx, # optional
    # password=xxxx, # optional
    # max_tag_value_length=None # optional
)

# Create a BatchSpanProcessor and add the exporter to it
span_processor = BatchSpanProcessor(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!')
```

You can configure the exporter with the following environment variables:

- *OTEL_EXPORTER_JAEGER_USER*
- *OTEL_EXPORTER_JAEGER_PASSWORD*
- *OTEL_EXPORTER_JAEGER_ENDPOINT*
- *OTEL_EXPORTER_JAEGER_AGENT_PORT*
- *OTEL_EXPORTER_JAEGER_AGENT_HOST*
- *OTEL_EXPORTER_JAEGER_AGENT_SPLIT_OVERSIZED_BATCHES*
- *OTEL_EXPORTER_JAEGER_TIMEOUT*

API

class opentelemetry.exporter.jaeger.thrift.**JaegerExporter**(*agent_host_name=None*, *agent_port=None*, *collector_endpoint=None*, *username=None*, *password=None*, *max_tag_value_length=None*, *udp_split_oversized_batches=None*, *timeout=None*)

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

Jaeger span exporter for OpenTelemetry.

Parameters

- **agent_host_name** (Optional[str, None]) – The host name of the Jaeger-Agent.
- **agent_port** (Optional[int, None]) – The port of the Jaeger-Agent.
- **collector_endpoint** (Optional[str, None]) – The endpoint of the Jaeger collector that uses Thrift over HTTP/HTTPS.
- **username** (Optional[str, None]) – The user name of the Basic Auth if authentication is required.
- **password** (Optional[str, None]) – The password of the Basic Auth if authentication is required.
- **max_tag_value_length** (Optional[int, None]) – Max length string attribute values can have. Set to None to disable.
- **udp_split_oversized_batches** (Optional[bool, None]) – Re-emit oversized batches in smaller chunks.
- **timeout** (Optional[int, None]) – Maximum time the Jaeger exporter should wait for each batch export.

export(*spans*)

Exports a batch of telemetry data.

Parameters **spans** – 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.

OpenTelemetry Jaeger Protobuf Exporter

The **OpenTelemetry Jaeger Protobuf Exporter** allows to export OpenTelemetry traces to Jaeger. This exporter always sends traces to the configured agent using Protobuf via gRPC.

Usage

```
from opentelemetry import trace
from opentelemetry.exporter.jaeger.proto.grpc import JaegerExporter
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

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

# create a JaegerExporter
jaeger_exporter = JaegerExporter(
    # optional: configure collector
    # collector_endpoint='localhost:14250',
    # insecure=True, # optional
    # credentials=xxx # optional channel creds
    # max_tag_value_length=None # optional
)

# Create a BatchSpanProcessor and add the exporter to it
span_processor = BatchSpanProcessor(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!')
```

You can configure the exporter with the following environment variables:

- *OTEL_EXPORTER_JAEGER_ENDPOINT*
- *OTEL_EXPORTER_JAEGER_CERTIFICATE*
- *OTEL_EXPORTER_JAEGER_TIMEOUT*

API

class opentelemetry.exporter.jaeger.proto.grpc.**JaegerExporter**(*collector_endpoint=None*, *insecure=None*, *credentials=None*, *max_tag_value_length=None*, *timeout=None*)

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

Jaeger span exporter for OpenTelemetry.

Parameters

- **collector_endpoint** (Optional[str, None]) – The endpoint of the Jaeger collector that uses Protobuf via gRPC.
- **insecure** (Optional[bool, None]) – True if collector has no encryption or authentication
- **credentials** (Optional[ChannelCredentials, None]) – Credentials for server authentication.

- **max_tag_value_length** (Optional[int, None]) – Max length string attribute values can have. Set to None to disable.
- **timeout** (Optional[int, None]) – Maximum time the Jaeger exporter should wait for each batch export.

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

Parameters **spans** – 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.

Submodules

class opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.**TagType**
Bases: object

STRING = 0

DOUBLE = 1

BOOL = 2

LONG = 3

BINARY = 4

class opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.**SpanRefType**
Bases: object

CHILD_OF = 0

FOLLOWS_FROM = 1

class opentelemetry.exporter.jaeger.thrift.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, 'vStr', 'UTF8', None), (4, 4, 'vDouble', None, None), (5, 2, 'vBool', None, None), (6, 10, 'vLong', None, None), (7, 11, 'vBinary', 'BINARY', None))

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.exporter.jaeger.thrift.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 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Tag'>, (None, (1, 11, 'key', 'UTF8', None), (2, 8, 'vType', None, None), (3, 11, 'vStr', 'UTF8', None), (4, 4, 'vDouble', None, None), (5, 2, 'vBool', None, None), (6, 10, 'vLong', None, None), (7, 11, 'vBinary', 'BINARY', None))), False), None))

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.exporter.jaeger.thrift.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), (3, 10, 'traceIdHigh', None, None), (4, 10, 'spanId', None, None))

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.exporter.jaeger.thrift.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, None), (3, 10, 'spanId', None, None), (4, 10, 'parentSpanId', None, None), (5, 11, 'operationName', 'UTF8', None), (6, 15, 'references', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.SpanRef'>, (None, (1, 8, 'refType', None, None), (2, 10, 'traceIdLow', None, None), (3, 10, 'traceIdHigh', None, None), (4, 10, 'spanId', None, None))), False), None), (7, 8, 'flags', None, None), (8, 10, 'startTime', None, None), (9, 10, 'duration', None, None), (10, 15, 'tags', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Tag'>, (None, (1, 11, 'key', 'UTF8', None), (2, 8, 'vType', None, None), (3, 11, 'vStr', 'UTF8', None), (4, 4, 'vDouble', None, None), (5, 2, 'vBool', None, None), (6, 10, 'vLong', None, None), (7, 11, 'vBinary', 'BINARY', None))), False), None), (11, 15, 'logs', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Log'>, (None, (1, 10, 'timestamp', None, None), (2, 15, 'fields', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Tag'>, (None, (1, 11, 'key', 'UTF8', None), (2, 8, 'vType', None, None), (3, 11, 'vStr', 'UTF8', None), (4, 4, 'vDouble', None, None), (5, 2, 'vBool', None, None), (6, 10, 'vLong', None, None), (7, 11, 'vBinary', 'BINARY', None))), False), None))), False), None))

read(*iprot*)

write(*oprot*)

validate()

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

Bases: object

- serviceName

- tags

thrift_spec = (None, (1, 11, 'serviceName', 'UTF8', None), (2, 15, 'tags', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Tag'>, (None, (1, 11, 'key', 'UTF8', None), (2, 8, 'vType', None, None), (3, 11, 'vStr', 'UTF8', None), (4, 4, 'vDouble', None, None), (5, 2, 'vBool', None, None), (6, 10, 'vLong', None, None), (7, 11, 'vBinary', 'BINARY', None))), False), None))

read(*iprot*)

write(*oprot*)

validate()

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

Bases: object

- process

- spans

thrift_spec = (None, (1, 12, 'process', (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Process'>, (None, (1, 11, 'serviceName', 'UTF8', None), (2, 15, 'tags', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Tag'>, (None, (1, 11, 'key', 'UTF8', None), (2, 8, 'vType', None, None), (3, 11, 'vStr', 'UTF8', None), (4, 4, 'vDouble', None, None), (5, 2, 'vBool', None, None), (6, 10, 'vLong', None, None), (7, 11, 'vBinary', 'BINARY', None))), False), None))), None), (2, 15, 'spans', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Span'>, (None, (1, 10, 'traceIdLow', None, None), (2, 10, 'traceIdHigh', None, None), (3, 10, 'spanId', None, None), (4, 10, 'parentSpanId', None, None), (5, 11, 'operationName', 'UTF8', None), (6, 15, 'references', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.SpanRef'>, (None, (1, 8, 'refType', None, None), (2, 10, 'traceIdLow', None, None), (3, 10, 'traceIdHigh', None, None), (4, 10, 'spanId', None, None))), False), None), (7, 8, 'flags', None, None), (8, 10, 'startTime', None, None), (9, 10, 'duration', None, None), (10, 15, 'tags', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Tag'>, (None, (1, 11, 'key', 'UTF8', None), (2, 8, 'vType', None, None), (3, 11, 'vStr', 'UTF8', None), (4, 4, 'vDouble', None, None), (5, 2, 'vBool', None, None), (6, 10, 'vLong', None, None), (7, 11, 'vBinary', 'BINARY', None))), False), None), (11, 15, 'logs', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Log'>, (None, (1, 10, 'timestamp', None, None), (2, 15, 'fields', (12, (<class 'opentelemetry.exporter.jaeger.thrift.gen.jaeger.ttypes.Tag'>, (None, (1, 11, 'key', 'UTF8', None), (2, 8, 'vType', None, None), (3, 11, 'vStr', 'UTF8', None), (4, 4, 'vDouble', None, None), (5, 2, 'vBool', None, None), (6, 10, 'vLong', None, None), (7, 11, 'vBinary', 'BINARY', None))), False), None))), False), None))), False), None))

read(*iprot*)

write(*oprot*)

validate()

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

Bases: object

- ok

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

read(*iprot*)

write(*oprot*)

validate()

class opentelemetry.exporter.jaeger.thrift.send.**AgentClientUDP**(*host_name*, *port*, *max_packet_size=65000*, *client=<class 'opentelemetry.exporter.jaeger.thrift.gen.agent.Agent.Client'>*, *split_oversized_batches=False*)

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.
- **split_oversized_batches** – Re-emit oversized batches in smaller chunks.

emit(*batch*)

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

class opentelemetry.exporter.jaeger.thrift.send.**Collector**(*thrift_url=''*, *auth=None*, *timeout_in_millis=None*)

Bases: object

Submits collected spans to Jaeger collector in jaeger.thrift format over binary thrift protocol. This is recommend option in cases where it is not feasible to deploy Jaeger Agent next to the application, for example, when the application code is running as AWS Lambda function. In these scenarios the Jaeger Clients can be configured to submit spans directly to the Collectors over HTTP/HTTPS.

Parameters

- **thrift_url** – Endpoint used to send spans directly to Collector the over HTTP.
- **auth** – Auth tuple that contains username and password for Basic Auth.
- **timeout_in_millis** – timeout for THttpClient.

submit(*batch*)

Submits batches to Thrift HTTP Server through Binary Protocol.

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

class opentelemetry.exporter.jaeger.proto.grpc.gen.collector_pb2_grpc.**CollectorServiceStub**(*channel*)

Bases: object

class opentelemetry.exporter.jaeger.proto.grpc.gen.collector_pb2_grpc.**CollectorServiceServicer**

Bases: object

PostSpans(*request*, *context*)

opentelemetry.exporter.jaeger.proto.grpc.gen.collector_pb2_grpc.**add_CollectorServiceServicer_to_server**(

3.1.6 OpenCensus Exporter

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

3.1.7 OpenTelemetry OTLP Exporters

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

Usage

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

You can configure the exporter with the following environment variables:

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

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

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

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

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

span_processor = BatchSpanProcessor(otlp_exporter)

trace.get_tracer_provider().add_span_processor(span_processor)

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

API

3.1.8 OpenTelemetry Zipkin Exporters

OpenTelemetry Zipkin JSON Exporter

This library allows to export tracing data to Zipkin.

Usage

The **OpenTelemetry Zipkin JSON Exporter** allows exporting of OpenTelemetry traces to Zipkin. This exporter sends traces to the configured Zipkin collector endpoint using JSON over HTTP and supports multiple versions (v1, v2).

```
from opentelemetry import trace
from opentelemetry.exporter.zipkin.json import ZipkinExporter
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

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

# create a ZipkinExporter
zipkin_exporter = ZipkinExporter(
    # version=Protocol.V2
    # optional:
    # endpoint="http://localhost:9411/api/v2/spans",
    # local_node_ipv4="192.168.0.1",
    # local_node_ipv6="2001:db8::c001",
    # local_node_port=31313,
    # max_tag_value_length=256
    # timeout=5 (in seconds)
)

# Create a BatchSpanProcessor and add the exporter to it
span_processor = BatchSpanProcessor(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!")
```

The exporter supports the following environment variable for configuration:

- *OTEL_EXPORTER_ZIPKIN_ENDPOINT*
- *OTEL_EXPORTER_ZIPKIN_TIMEOUT*

API

class opentelemetry.exporter.zipkin.json.**ZipkinExporter**(*version=Protocol.V2*, *endpoint=None*, *local_node_ipv4=None*, *local_node_ipv6=None*, *local_node_port=None*, *max_tag_value_length=None*, *timeout=None*)

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

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

OpenTelemetry Zipkin Protobuf Exporter

This library allows to export tracing data to Zipkin.

Usage

The **OpenTelemetry Zipkin Exporter** allows exporting of OpenTelemetry traces to Zipkin. This exporter sends traces to the configured Zipkin collector endpoint using HTTP and supports v2 protobuf.

```
from opentelemetry import trace
from opentelemetry.exporter.zipkin.proto.http import ZipkinExporter
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

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

# create a ZipkinExporter
zipkin_exporter = ZipkinExporter(
    # optional:
    # endpoint="http://localhost:9411/api/v2/spans",
    # local_node_ipv4="192.168.0.1",
    # local_node_ipv6="2001:db8::c001",
    # local_node_port=31313,
    # max_tag_value_length=256
    # timeout=5 (in seconds)
)
```

(continues on next page)

(continued from previous page)

```
# Create a BatchSpanProcessor and add the exporter to it
span_processor = BatchSpanProcessor(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!")
```

The exporter supports the following environment variable for configuration:

- *OTEL_EXPORTER_ZIPKIN_ENDPOINT*
- *OTEL_EXPORTER_ZIPKIN_TIMEOUT*

API

class opentelemetry.exporter.zipkin.proto.http.**ZipkinExporter**(*endpoint=None*, *local_node_ipv4=None*, *local_node_ipv6=None*, *local_node_port=None*, *max_tag_value_length=None*, *timeout=None*)

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

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

3.1.9 OpenTracing Shim for OpenTelemetry

The OpenTelemetry OpenTracing shim is a library which allows an easy migration from OpenTracing to OpenTelemetry.

The shim consists of a set of classes which implement the OpenTracing Python API while using OpenTelemetry constructs behind the scenes. Its purpose is to allow applications which are already instrumented using OpenTracing to start using OpenTelemetry with a minimal effort, without having to rewrite large portions of the codebase.

To use the shim, a *TracerShim* instance is created and then used as if it were an "ordinary" OpenTracing opentracing.Tracer, as in the following example:

```
import time

from opentelemetry import trace
```

(continues on next page)

(continued from previous page)

```
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.shim.opentracing_shim import create_tracer

# Define which OpenTelemetry Tracer provider implementation to use.
trace.set_tracer_provider(TracerProvider())

# Create an OpenTelemetry Tracer.
otel_tracer = trace.get_tracer(__name__)

# Create an OpenTracing shim.
shim = create_tracer(otel_tracer)

with shim.start_active_span("ProcessHTTPRequest"):
    print("Processing HTTP request")
    # Sleeping to mock real work.
    time.sleep(0.1)
    with shim.start_active_span("GetDataFromDB"):
        print("Getting data from DB")
        # Sleeping to mock real work.
        time.sleep(0.2)
```

Note: While the OpenTracing Python API represents time values as the number of **seconds** since the epoch expressed as `float` values, the OpenTelemetry Python API represents time values as the number of **nanoseconds** since the epoch expressed as `int` values. This fact requires the OpenTracing shim to convert time values back and forth between the two representations, which involves floating point arithmetic.

Due to the way computers represent floating point values in hardware, representation of decimal floating point values in binary-based hardware is imprecise by definition.

The above results in **slight imprecisions** in time values passed to the shim via the OpenTracing API when comparing the value passed to the shim and the value stored in the OpenTelemetry *opentelemetry.trace.Span* object behind the scenes. **This is not a bug in this library or in Python**. Rather, this is a generic problem which stems from the fact that not every decimal floating point number can be correctly represented in binary, and therefore affects other libraries and programming languages as well. More information about this problem can be found in the Floating Point Arithmetic: Issues and Limitations section of the Python documentation.

While testing this library, the aforementioned imprecisions were observed to be of *less than a microsecond*.

API

opentelemetry.shim.opentracing_shim.**create_tracer**(*otel_tracer_provider*)

Creates a *TracerShim* object from the provided OpenTelemetry *opentelemetry.trace.TracerProvider*.

The returned *TracerShim* is an implementation of opentracing.Tracer using OpenTelemetry under the hood.

Parameters **otel_tracer_provider** (*TracerProvider*) – A tracer from this provider will be used to perform the actual tracing when user code is instrumented using the OpenTracing API.

Return type *TracerShim*

Returns The created *TracerShim*.

class opentelemetry.shim.opentracing_shim.**SpanContextShim**(*otel_context*)
Implements opentracing.SpanContext by wrapping a *opentelemetry.trace.SpanContext* object.

Parameters **otel_context** (*SpanContext*) – A *opentelemetry.trace.SpanContext* to be used for constructing the *SpanContextShim*.

unwrap()
Returns the wrapped *opentelemetry.trace.SpanContext* object.

Return type *SpanContext*

Returns The *opentelemetry.trace.SpanContext* object wrapped by this *SpanContextShim*.

property baggage: opentelemetry.context.context.Context
Returns the baggage associated with this object

Return type *Context*

class opentelemetry.shim.opentracing_shim.**SpanShim**(*tracer*, *context*, *span*)
Wraps a *opentelemetry.trace.Span* object.

Parameters

- **tracer** – The opentracing.Tracer that created this *SpanShim*.
- **context** (*SpanContextShim*) – A *SpanContextShim* which contains the context for this *SpanShim*.
- **span** – A *opentelemetry.trace.Span* to wrap.

unwrap()
Returns the wrapped *opentelemetry.trace.Span* object.

Returns The *opentelemetry.trace.Span* object wrapped by this *SpanShim*.

set_operation_name(*operation_name*)
Updates the name of the wrapped OpenTelemetry span.

Parameters **operation_name** (str) – The new name to be used for the underlying *opentelemetry.trace.Span* object.

Return type *SpanShim*

Returns Returns this *SpanShim* instance to allow call chaining.

finish(*finish_time=None*)
Ends the OpenTelemetry span wrapped by this *SpanShim*.

If *finish_time* is provided, the time value is converted to the OpenTelemetry time format (number of nanoseconds since the epoch, expressed as an integer) and passed on to the OpenTelemetry tracer when ending the OpenTelemetry span. If *finish_time* isn't provided, it is up to the OpenTelemetry tracer implementation to generate a timestamp when ending the span.

Parameters **finish_time** (Optional[float, None]) – A value that represents the finish time expressed as the number of seconds since the epoch as returned by time.time().

set_tag(*key*, *value*)
Sets an OpenTelemetry attribute on the wrapped OpenTelemetry span.

Parameters

- **key** (str) – A tag key.
- **value** (*~ValueT*) – A tag value.

Return type *SpanShim*

Returns Returns this *SpanShim* instance to allow call chaining.

log_kv(*key_values*, *timestamp=None*)
Logs an event for the wrapped OpenTelemetry span.

Note: The OpenTracing API defines the values of *key_values* to be of any type. However, the OpenTelemetry API requires that the values be any one of the types defined in `opentelemetry.trace.util.Attributes` therefore, only these types are supported as values.

Parameters

- **key_values** (`Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]`) – A dictionary as specified in `opentelemetry.trace.util.Attributes`.
- **timestamp** (`Optional[float, None]`) – Timestamp of the OpenTelemetry event, will be generated automatically if omitted.

Return type *SpanShim*

Returns Returns this *SpanShim* instance to allow call chaining.

log(***kwargs*)
DEPRECATED

log_event(*event*, *payload=None*)
DEPRECATED

set_baggage_item(*key*, *value*)
Stores a Baggage item in the span as a key/value pair.

Parameters

- **key** (`str`) – A tag key.
- **value** (`str`) – A tag value.

get_baggage_item(*key*)
Retrieves value of the baggage item with the given key.

Parameters **key** (`str`) – A tag key.

Return type `Optional[object, None]`

Returns Returns this *SpanShim* instance to allow call chaining.

class opentelemetry.shim.opentracing_shim.**ScopeShim**(*manager*, *span*, *span_cm=None*)
A *ScopeShim* wraps the OpenTelemetry functionality related to span activation/deactivation while using OpenTracing `opentracing.Scope` objects for presentation.

Unlike other classes in this package, the *ScopeShim* class doesn't wrap an OpenTelemetry class because OpenTelemetry doesn't have the notion of "scope" (though it *does* have similar functionality).

There are two ways to construct a *ScopeShim* object: using the default initializer and using the *from_context_manager()* class method.

It is necessary to have both ways for constructing *ScopeShim* objects because in some cases we need to create the object from an OpenTelemetry *opentelemetry.trace.Span* context manager (as returned by *opentelemetry.trace.use_span()*), in which case our only way of retrieving a *opentelemetry.trace.Span* object is by calling the `__enter__()` method on the context manager, which makes the span active

in the OpenTelemetry tracer; whereas in other cases we need to accept a *SpanShim* object and wrap it in a *ScopeShim*. The former is used mainly when the instrumentation code retrieves the currently-active span using *ScopeManagerShim.active*. The latter is mainly used when the instrumentation code activates a span using *ScopeManagerShim.activate()*.

Parameters

- **manager** (*ScopeManagerShim*) – The *ScopeManagerShim* that created this *ScopeShim*.
- **span** (*SpanShim*) – The *SpanShim* this *ScopeShim* controls.
- **span_cm** – A Python context manager which yields an OpenTelemetry *opentelemetry.trace.Span* from its `__enter__()` method. Used by *from_context_manager()* to store the context manager as an attribute so that it can later be closed by calling its `__exit__()` method. Defaults to `None`.

classmethod from_context_manager(*manager*, *span_cm*)

Constructs a *ScopeShim* from an OpenTelemetry *opentelemetry.trace.Span* context manager.

The method extracts a *opentelemetry.trace.Span* object from the context manager by calling the context manager's `__enter__()` method. This causes the span to start in the OpenTelemetry tracer.

Example usage:

```
span = otel_tracer.start_span("TestSpan")
span_cm = opentelemetry.trace.use_span(span)
scope_shim = ScopeShim.from_context_manager(
    scope_manager_shim,
    span_cm=span_cm,
)
```

Parameters

- **manager** (*ScopeManagerShim*) – The *ScopeManagerShim* that created this *ScopeShim*.
- **span_cm** – A context manager as returned by *opentelemetry.trace.use_span()*.

close()

Closes the *ScopeShim*. If the *ScopeShim* was created from a context manager, calling this method sets the active span in the OpenTelemetry tracer back to the span which was active before this *ScopeShim* was created. In addition, if the span represented by this *ScopeShim* was activated with the *finish_on_close* argument set to `True`, calling this method will end the span.

> **Warning:** In the current state of the implementation it is possible to create a *ScopeShim* directly from a *SpanShim*, that is - without using *from_context_manager()*. For that reason we need to be able to end the span represented by the *ScopeShim* in this case, too. Please note that closing a *ScopeShim* created this way (for example as returned by *ScopeManagerShim.active()*) **always ends the associated span**, regardless of the value passed in *finish_on_close* when activating the span.

class opentelemetry.shim.opentracing_shim.**ScopeManagerShim**(*tracer*)

Implements `opentracing.ScopeManager` by setting and getting the active *opentelemetry.trace.Span* in the OpenTelemetry tracer.

This class keeps a reference to a *TracerShim* as an attribute. This reference is used to communicate with the OpenTelemetry tracer. It is necessary to have a reference to the *TracerShim* rather than the *opentelemetry.trace.Tracer* wrapped by it because when constructing a *SpanShim* we need to pass a reference to a `opentracing.Tracer`.

Parameters **tracer** (*TracerShim*) – A *TracerShim* to use for setting and getting active span state.

activate(*span*, *finish_on_close*)

Activates a *SpanShim* and returns a *ScopeShim* which represents the active span.

Parameters

- **span** (*SpanShim*) – A *SpanShim* to be activated.
- **finish_on_close** (bool) – Determines whether the OpenTelemetry span should be ended when the returned *ScopeShim* is closed.

Return type *ScopeShim*

Returns A *ScopeShim* representing the activated span.

property active: opentelemetry.shim.opentracing_shim.ScopeShim

Returns a *ScopeShim* object representing the currently-active span in the OpenTelemetry tracer.

Return type *ScopeShim*

Returns A *ScopeShim* representing the active span in the OpenTelemetry tracer, or None if no span is currently active.

> **Warning:** Calling *ScopeShim.close()* on the *ScopeShim* returned by this property **always ends the corresponding span**, regardless of the *finish_on_close* value used when activating the span. This is a limitation of the current implementation of the OpenTracing shim and is likely to be handled in future versions.

property tracer: opentelemetry.shim.opentracing_shim.TracerShim

Returns the *TracerShim* reference used by this *ScopeManagerShim* for setting and getting the active span from the OpenTelemetry tracer.

Return type *TracerShim*

Returns The *TracerShim* used for setting and getting the active span.

> **Warning:** This property is *not* a part of the OpenTracing API. It is used internally by the current implementation of the OpenTracing shim and will likely be removed in future versions.

class opentelemetry.shim.opentracing_shim.**TracerShim**(*tracer*)

Wraps a *opentelemetry.trace.Tracer* object.

This wrapper class allows using an OpenTelemetry tracer as if it were an OpenTracing tracer. It exposes the same methods as an "ordinary" OpenTracing tracer, and uses OpenTelemetry transparently for performing the actual tracing.

This class depends on the *OpenTelemetry API*. Therefore, any implementation of a *opentelemetry.trace.Tracer* should work with this class.

Parameters **tracer** (*Tracer*) – A *opentelemetry.trace.Tracer* to use for tracing. This tracer will be invoked by the shim to create actual spans.

unwrap()

Returns the *opentelemetry.trace.Tracer* object that is wrapped by this *TracerShim* and used for actual tracing.

Returns The *opentelemetry.trace.Tracer* used for actual tracing.

start_active_span(*operation_name*, *child_of=None*, *references=None*, *tags=None*, *start_time=None*, *ignore_active_span=False*, *finish_on_close=True*)

Starts and activates a span. In terms of functionality, this method behaves exactly like the same method on a "regular" OpenTracing tracer. See `opentracing.Tracer.start_active_span()` for more details.

Parameters

- **operation_name** (`str`) – Name of the operation represented by the new span from the perspective of the current service.
- **child_of** (`Union[SpanShim, SpanContextShim, None]`) – A *SpanShim* or *SpanContextShim* representing the parent in a "child of" reference. If specified, the *references* parameter must be omitted.
- **references** (`Optional[list, None]`) – A list of `opentracing.Reference` objects that identify one or more parents of type *SpanContextShim*.
- **tags** (`Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]`) – A dictionary of tags.
- **start_time** (`Optional[float, None]`) – An explicit start time expressed as the number of seconds since the epoch as returned by `time.time()`.
- **ignore_active_span** (`bool`) – Ignore the currently-active span in the OpenTelemetry tracer and make the created span the root span of a new trace.
- **finish_on_close** (`bool`) – Determines whether the created span should end automatically when closing the returned *ScopeShim*.

Return type *ScopeShim*

Returns A *ScopeShim* that is already activated by the *ScopeManagerShim*.

start_span(*operation_name=None*, *child_of=None*, *references=None*, *tags=None*, *start_time=None*, *ignore_active_span=False*)

Implements the `start_span()` method from the base class.

Starts a span. In terms of functionality, this method behaves exactly like the same method on a "regular" OpenTracing tracer. See `opentracing.Tracer.start_span()` for more details.

Parameters

- **operation_name** (`Optional[str, None]`) – Name of the operation represented by the new span from the perspective of the current service.
- **child_of** (`Union[SpanShim, SpanContextShim, None]`) – A *SpanShim* or *SpanContextShim* representing the parent in a "child of" reference. If specified, the *references* parameter must be omitted.
- **references** (`Optional[list, None]`) – A list of `opentracing.Reference` objects that identify one or more parents of type *SpanContextShim*.
- **tags** (`Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]`) – A dictionary of tags.
- **start_time** (`Optional[float, None]`) – An explicit start time expressed as the number of seconds since the epoch as returned by `time.time()`.
- **ignore_active_span** (`bool`) – Ignore the currently-active span in the OpenTelemetry tracer and make the created span the root span of a new trace.

Return type *SpanShim*

Returns An already-started *SpanShim* instance.

`inject`(*span_context*, *format*, *carrier*)
Injects `span_context` into `carrier`.

See base class for more details.

Parameters

- **`span_context`** – The `opentracing.SpanContext` to inject.
- **`format`** (`object`) – a Python object instance that represents a given carrier format. `format` may be of any type, and `format` equality is defined by Python == operator.
- **`carrier`** (`object`) – the format-specific carrier object to inject into

`extract`(*format*, *carrier*)
Returns an `opentracing.SpanContext` instance extracted from a `carrier`.

See base class for more details.

Parameters

- **`format`** (`object`) – a Python object instance that represents a given carrier format. `format` may be of any type, and `format` equality is defined by python == operator.
- **`carrier`** (`object`) – the format-specific carrier object to extract from

Returns An `opentracing.SpanContext` extracted from `carrier` or `None` if no such `SpanContext` could be found.

3.1.10 Performance Tests - Benchmarks

Click here to view the latest performance benchmarks for packages in this repo.

3.1.11 Auto-instrumentation

One of the best ways to instrument Python applications is to use OpenTelemetry automatic instrumentation (auto-instrumentation). This approach is simple, easy, and doesn't require many code changes. You only need to install a few Python packages to successfully instrument your application's code.

Overview

This example demonstrates how to use auto-instrumentation in OpenTelemetry. The example is based on a previous OpenTracing example that you can find here.

The source files for these examples are available here.

This example uses two different scripts. The main difference between them is whether or not they're instrumented manually:

1. `server_instrumented.py` - instrumented manually
2. `server_uninstrumented.py` - not instrumented manually

Run the first script without the automatic instrumentation agent and the second with the agent. They should both produce the same results, demonstrating that the automatic instrumentation agent does exactly the same thing as manual instrumentation.

To better understand auto-instrumentation, see 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",
        context=extract(request.headers),
        kind=trace.SpanKind.SERVER,
        attributes=collect_request_attributes(request.environ),
    ):
        print(request.args.get("param"))
        return "served"
```

Server not instrumented manually

server_uninstrumented.py

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

Prepare

Execute the following example in a separate virtual environment. Run the following commands to prepare for auto-instrumentation:

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

Install

Run the following commands to install the appropriate packages. The `opentelemetry-instrumentation` package provides several commands that help automatically instruments a program.

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

Execute

This section guides you through the manual process of instrumenting a server as well as the process of executing an automatically instrumented server.

Execute a manually instrumented server

Execute the server in two 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
```

When you execute `server_instrumented.py` it returns a JSON response similar to the following example:

```
{
    "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": {
        "status_code": "OK"
    },
    "attributes": {
        "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": [],
    "resource": {
        "telemetry.sdk.language": "python",
        "telemetry.sdk.name": "opentelemetry",
        "telemetry.sdk.version": "0.16b1"
    }
}
```

Execute an automatically instrumented server

Stop the execution of `server_instrumented.py` with `ctrl + c` and run the following command instead:

```
$ opentelemetry-instrument --trace-exporter console_span python server_uninstrumented.py
```

In the console where you previously executed `client.py`, run the following command again:

```
$ python client.py testing
```

When you execute `server_uninstrumented.py` it returns a JSON response similar to the following example:

```
{
    "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": {
        "status_code": "OK"
    },
    "attributes": {
        "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": [],
    "resource": {
    "telemetry.sdk.language": "python",
    "telemetry.sdk.name": "opentelemetry",
    "telemetry.sdk.version": "0.16b1",
    "service.name": ""
    }
}
```

You can see that both outputs are the same because automatic instrumentation does exactly what manual instrumentation does.

Instrumentation while debugging

The debug mode can be enabled in the Flask app like this:

```
if __name__ == "__main__":
    app.run(port=8082, debug=True)
```

The debug mode can break instrumentation from happening because it enables a reloader. To run instrumentation while the debug mode is enabled, set the `use_reloader` option to `False`:

```
if __name__ == "__main__":
    app.run(port=8082, debug=True, use_reloader=False)
```

Additional resources

In order to send telemetry to an OpenTelemetry Collector without doing any additional configuration, read about the OpenTelemetry Distro package.

3.1.12 Basic Context

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

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

The source files of these examples are available here.

Installation

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

Run the Example

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

The output will be shown in the console.

Useful links

- OpenTelemetry
- *opentelemetry.trace package*

3.1.13 Basic Trace

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

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

The source files of these examples are available here.

Installation

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

Run the Example

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

The output will be shown in the console.

Useful links

- OpenTelemetry
- *opentelemetry.trace package*

3.1.14 Datadog Exporter

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-exporter-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 basic_example.py
```


```
python basic_example.py
```


Distributed Example

- Installation

```
pip install opentelemetry-api
pip install opentelemetry-sdk
pip install opentelemetry-exporter-datadog
pip install opentelemetry-instrumentation
pip install opentelemetry-instrumentation-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-instrument python server.py
```

- Run client

```
opentelemetry-instrument python client.py testing
```

- Run client with parameter to raise error

```
opentelemetry-instrument python client.py error
```

- Run Datadog instrumented client

The OpenTelemetry instrumented server is set up with propagation of Datadog trace context.

```
pip install ddtrace
ddtrace-run python datadog_client.py testing
```

3.1.15 OpenTelemetry Distro

In order to make using OpenTelemetry and auto-instrumentation as quick as possible without sacrificing flexibility, OpenTelemetry distros provide a mechanism to automatically configure some of the more common options for users. By harnessing their power, users of OpenTelemetry can configure the components as they need. The `opentelemetry-distro` package provides some defaults to users looking to get started, it configures:

- the SDK TracerProvider
- a BatchSpanProcessor
- the OTLP `SpanExporter` to send data to an OpenTelemetry collector

The package also provides a starting point for anyone interested in producing an alternative distro. The interfaces implemented by the package are loaded by the auto-instrumentation via the `opentelemetry_distro` and `opentelemetry_configurator` entry points to configure the application before any other code is executed.

In order to automatically export data from OpenTelemetry to the OpenTelemetry collector, installing the package will setup all the required entry points.

```
$ pip install opentelemetry-distro[otlp] opentelemetry-instrumentation
```

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

```
# /tmp/otel-collector-config.yaml
receivers:
    otlp:
        protocols:
            grpc:
            http:
exporters:
    logging:
        loglevel: debug
processors:
    batch:
service:
    pipelines:
        traces:
            receivers: [otlp]
            exporters: [logging]
            processors: [batch]
```

Then start the Docker container:

```
docker run -p 4317:4317 \
    -v /tmp/otel-collector-config.yaml:/etc/otel-collector-config.yaml \
    otel/opentelemetry-collector:latest \
    --config=/etc/otel-collector-config.yaml
```

The following code will create a span with no configuration.

```
# no_configuration.py
from opentelemetry import trace

with trace.get_tracer(__name__).start_as_current_span("foo"):
    with trace.get_tracer(__name__).start_as_current_span("bar"):
        print("baz")
```

Lastly, run the `no_configuration.py` with the auto-instrumentation:

```
$ opentelemetry-instrument python no_configuration.py
```

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

```
Resource labels:
     -> telemetry.sdk.language: STRING(python)
     -> telemetry.sdk.name: STRING(opentelemetry)
     -> telemetry.sdk.version: STRING(1.1.0)
     -> service.name: STRING(unknown_service)
InstrumentationLibrarySpans #0
InstrumentationLibrary __main__
Span #0
    Trace ID       : db3c99e5bfc50ef8be1773c3765e8845
    Parent ID      : 0677126a4d110cb8
    ID             : 3163b3022808ed1b
    Name           : bar
    Kind           : SPAN_KIND_INTERNAL
    Start time     : 2021-05-06 22:54:51.23063 +0000 UTC
    End time       : 2021-05-06 22:54:51.230684 +0000 UTC
    Status code    : STATUS_CODE_UNSET
    Status message :
Span #1
    Trace ID       : db3c99e5bfc50ef8be1773c3765e8845
    Parent ID      :
    ID             : 0677126a4d110cb8
    Name           : foo
    Kind           : SPAN_KIND_INTERNAL
    Start time     : 2021-05-06 22:54:51.230549 +0000 UTC
    End time       : 2021-05-06 22:54:51.230706 +0000 UTC
    Status code    : STATUS_CODE_UNSET
    Status message :
```

3.1.16 Django Instrumentation

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

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

Preparation

This example will be executed in a separate virtual environment:

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

Installation

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

Execution

Execution of the Django app

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

Set these environment variables first:

1. `export DJANGO_SETTINGS_MODULE=instrumentation_example.settings`

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

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

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

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

Execution of the client

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

`source django_auto_instrumentation/bin/activate`

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

`python client.py hello`

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

```
{
    "name": "home_page_view",
    "context": {
        "trace_id": "0xed88755c56d95d05a506f5f70e7849b9",
        "span_id": "0x0a94c7a60e0650d5",
        "trace_state": "{}"
    },
    "kind": "SpanKind.SERVER",
    "parent_id": "0x3096ef92e621c22d",
    "start_time": "2020-04-26T01:49:57.205833Z",
    "end_time": "2020-04-26T01:49:57.206214Z",
    "status": {
        "status_code": "OK"
    },
    "attributes": {
        "http.method": "GET",
        "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.

Disabling Django Instrumentation

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

```
export OTEL_PYTHON_DJANGO_INSTRUMENT=False
```

Auto Instrumentation

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

Usage with Auto Instrumentation and uWSGI

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

```
pip install uwsgi
```

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

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

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

References

- Django
- OpenTelemetry Project
- OpenTelemetry Django extension

3.1.17 Global Error Handler

Overview

This example shows how to use the global error handler.

Preparation

This example will be executed in a separate virtual environment:

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

Installation

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

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

Execution

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

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

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

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

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

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

No error raised
```

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

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

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

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

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

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

```
Error handled by default error handler:
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "test.py", line 23, in <module>
    assert False
AssertionError
```

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

```
No error raised
```

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

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

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

logger = getLogger(__name__)


class ErrorHandler0(ErrorHandler, ZeroDivisionError):

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

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

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

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

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

3.1.18 Working With Fork Process Models

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

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

Gunicorn post_fork hook

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


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

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

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

uWSGI postfork decorator

```
from uwsgidecorators import postfork

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


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

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

(continues on next page)

(continued from previous page)

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

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

3.1.19 OpenCensus Exporter

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

The source files of this example are available here.

Installation

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

Run the Example

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

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

Now, the example can be executed:

```
python collector.py
```

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

Useful links

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

3.1.20 OpenTracing Shim

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

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

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

Installation

Jaeger

Start Jaeger

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

Redis

Install Redis following the instructions.

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

```
redis-server
```

Python Dependencies

Install the Python dependencies in requirements.txt

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

Alternatively, you can install the Python dependencies separately:

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

Run the Application

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

To run the script:

```
python main.py
```

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

Jaeger UI

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

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

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

Useful links

- OpenTelemetry
- *OpenTracing Shim for OpenTelemetry*

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

D

E

F

G

I

J

K

L

P

R

S

T

U

V

W

Z