OpenTelemetry Python

OpenTelemetry Authors

Mar 14, 2022

CORE PACKAGES

Welcome to the docs for the Python OpenTelemetry implementation.

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

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

CHAPTER ONE

GETTING STARTED

1.1 OpenTelemetry Python API

1.1.1 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

1.1.2 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`

1.1.3 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:

```python
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:

```python
# Explicit parent span assignment is done via the Context
from opentelemetry.trace import set_span_in_context

```

(continues on next page)

(continued from previous page)

```
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=None*, *schema_url=None*)

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. `__name__` may not be used as this can result in different tracer names if the tracers are in different files. It is better to use a fixed string that can be imported where needed and used consistently as the name of the tracer.

  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`** (`Optional[str, None]`) – Optional. The version string of the instrumenting library. Usually this should be the same as `pkg_resources.get_distribution(instrumenting_library_name).version`.

- **`schema_url`** (`Optional[str, None]`) – Optional. Specifies the Schema URL of the emitted telemetry.

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:

```python
# 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=None*, *tracer_provider=None*, *schema_url=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.

1.1.4 opentelemetry._metrics package

Warning: OpenTelemetry Python metrics are in an experimental state. The APIs within *`opentelemetry._metrics`* are subject to change in minor/patch releases and make no backward compatability guarantees at this time.

Once metrics become stable, this package will be be renamed to `opentelemetry.metrics`.

Submodules

opentelemetry._metrics.instrument

`class opentelemetry._metrics.instrument.`**`Instrument`**`(`*name*, *unit=''*, *description=''*`)`
Bases: `abc.ABC`

`_abc_impl = <_abc._abc_data object>`

`class opentelemetry._metrics.instrument.`**`_ProxyInstrument`**`(`*name*, *unit*, *description*`)`
Bases: `abc.ABC`, `Generic[opentelemetry._metrics.instrument.InstrumentT]`

`on_meter_set``(`*meter*`)`
Called when a real meter is set on the creating _ProxyMeter

Return type `None`

`abstract _create_real_instrument``(`*meter*`)`
Create an instance of the real instrument. Implement this.

Return type ~InstrumentT

`_abc_impl = <_abc._abc_data object>`

`class opentelemetry._metrics.instrument.`**`_ProxyAsynchronousInstrument`**`(`*name*, *callback*, *unit*, *description*`)`
Bases: *`opentelemetry._metrics.instrument._ProxyInstrument`*`[opentelemetry._metrics.instrument.InstrumentT]`

`_abc_impl = <_abc._abc_data object>`

class opentelemetry._metrics.instrument.**Synchronous**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument.Instrument*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**Asynchronous**(*name*, *callback*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument.Instrument*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_Adding**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument.Instrument*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_Grouping**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument.Instrument*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_Monotonic**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument._Adding*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_NonMonotonic**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument._Adding*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**Counter**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument._Monotonic*, *opentelemetry._metrics.instrument.Synchronous*

abstract add(*amount*, *attributes=None*)

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**DefaultCounter**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument.Counter*

add(*amount*, *attributes=None*)

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_ProxyCounter**(*name*, *unit*, *description*)
Bases: *opentelemetry._metrics.instrument._ProxyInstrument*[*opentelemetry._metrics.instrument.Counter*], *opentelemetry._metrics.instrument.Counter*

add(*amount*, *attributes=None*)

_create_real_instrument(*meter*)
Create an instance of the real instrument. Implement this.

Return type *Counter*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**UpDownCounter**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument._NonMonotonic*, *opentelemetry._metrics.instrument.Synchronous*

abstract add(*amount*, *attributes=None*)

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**DefaultUpDownCounter**(*name*, *unit=''*, *description=''*)

Bases: *opentelemetry._metrics.instrument.UpDownCounter*

add(*amount*, *attributes=None*)

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_ProxyUpDownCounter**(*name*, *unit*, *description*)

Bases: *opentelemetry._metrics.instrument._ProxyInstrument*[*opentelemetry._metrics.instrument.UpDownCounter*], *opentelemetry._metrics.instrument.UpDownCounter*

add(*amount*, *attributes=None*)

_create_real_instrument(*meter*)

Create an instance of the real instrument. Implement this.

Return type *UpDownCounter*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**ObservableCounter**(*name*, *callback*, *unit=''*, *description=''*)

Bases: *opentelemetry._metrics.instrument._Monotonic*, *opentelemetry._metrics.instrument.Asynchronous*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**DefaultObservableCounter**(*name*, *callback*, *unit=''*, *description=''*)

Bases: *opentelemetry._metrics.instrument.ObservableCounter*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_ProxyObservableCounter**(*name*, *callback*, *unit*, *description*)

Bases: *opentelemetry._metrics.instrument._ProxyAsynchronousInstrument*[*opentelemetry._metrics.instrument.ObservableCounter*], *opentelemetry._metrics.instrument.ObservableCounter*

_create_real_instrument(*meter*)

Create an instance of the real instrument. Implement this.

Return type *ObservableCounter*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**ObservableUpDownCounter**(*name*, *callback*, *unit=''*, *description=''*)

Bases: *opentelemetry._metrics.instrument._NonMonotonic*, *opentelemetry._metrics.instrument.Asynchronous*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**DefaultObservableUpDownCounter**(*name*, *callback*, *unit=''*, *description=''*)

Bases: *opentelemetry._metrics.instrument.ObservableUpDownCounter*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_ProxyObservableUpDownCounter**(*name*, *callback*, *unit*, *description*)

Bases: *opentelemetry._metrics.instrument._ProxyAsynchronousInstrument*[*opentelemetry._metrics.instrument.ObservableUpDownCounter*], *opentelemetry._metrics.instrument.ObservableUpDownCounter*

_create_real_instrument(*meter*)
Create an instance of the real instrument. Implement this.

Return type *ObservableUpDownCounter*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**Histogram**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument._Grouping*, *opentelemetry._metrics.instrument.Synchronous*

abstract record(*amount*, *attributes=None*)

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**DefaultHistogram**(*name*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument.Histogram*

record(*amount*, *attributes=None*)

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_ProxyHistogram**(*name*, *unit*, *description*)
Bases: *opentelemetry._metrics.instrument._ProxyInstrument*[*opentelemetry._metrics.instrument.Histogram*], *opentelemetry._metrics.instrument.Histogram*

record(*amount*, *attributes=None*)

_create_real_instrument(*meter*)
Create an instance of the real instrument. Implement this.

Return type *Histogram*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**ObservableGauge**(*name*, *callback*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument._Grouping*, *opentelemetry._metrics.instrument.Asynchronous*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**DefaultObservableGauge**(*name*, *callback*, *unit=''*, *description=''*)
Bases: *opentelemetry._metrics.instrument.ObservableGauge*

_abc_impl = <_abc._abc_data object>

class opentelemetry._metrics.instrument.**_ProxyObservableGauge**(*name*, *callback*, *unit*, *description*)
Bases: *opentelemetry._metrics.instrument._ProxyAsynchronousInstrument*[*opentelemetry._metrics.instrument.ObservableGauge*], *opentelemetry._metrics.instrument.ObservableGauge*

_create_real_instrument(*meter*)
Create an instance of the real instrument. Implement this.

Return type *ObservableGauge*

_abc_impl = <_abc._abc_data object>

opentelemetry._metrics.measurement

class opentelemetry._metrics.measurement.**Measurement**(*value*, *attributes=None*)

Bases: `object`

A measurement observed in an asynchronous instrument

Return/yield instances of this class from asynchronous instrument callbacks.

Parameters

- **value** (Union[int, float]) – The float or int measured value
- **attributes** (Optional[Mapping[str, Union[str, bool, int, float, Sequence[str], Sequence[bool], Sequence[int], Sequence[float]]], None]) – The measurement's attributes

property value: Union[float, int]

Return type Union[float, int]

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]

Module contents

This module provides abstract and concrete (but noop) classes that can be used to generate metrics.

class opentelemetry._metrics.**MeterProvider**

Bases: `abc.ABC`

abstract get_meter(*name*, *version=None*, *schema_url=None*)

Return type *Meter*

class opentelemetry._metrics.**NoOpMeterProvider**

Bases: *opentelemetry._metrics.MeterProvider*

get_meter(*name*, *version=None*, *schema_url=None*)

Return type *Meter*

class opentelemetry._metrics.**Meter**(*name*, *version=None*, *schema_url=None*)

Bases: `abc.ABC`

property name

property version

property schema_url

abstract create_counter(*name*, *unit=''*, *description=''*)

Return type *Counter*

abstract create_up_down_counter(*name*, *unit=''*, *description=''*)

Return type *UpDownCounter*

abstract create_observable_counter(*name*, *callback*, *unit=''*, *description=''*)

Creates an observable counter instrument

An observable counter observes a monotonically increasing count by calling a provided callback which returns multiple *Measurement*.

For example, an observable counter could be used to report system CPU time periodically. Here is a basic implementation:

```
def cpu_time_callback() -> Iterable[Measurement]:
    measurements = []
    with open("/proc/stat") as procstat:
        procstat.readline()  # skip the first line
        for line in procstat:
            if not line.startswith("cpu"): break
            cpu, *states = line.split()
            measurements.append(Measurement(int(states[0]) // 100, {"cpu": cpu,
↪"state": "user"}))
            measurements.append(Measurement(int(states[1]) // 100, {"cpu": cpu,
↪"state": "nice"}))
            measurements.append(Measurement(int(states[2]) // 100, {"cpu": cpu,
↪"state": "system"}))
            # ... other states
    return measurements

meter.create_observable_counter(
    "system.cpu.time",
    callback=cpu_time_callback,
    unit="s",
    description="CPU time"
)
```

To reduce memory usage, you can use generator callbacks instead of building the full list:

```
def cpu_time_callback() -> Iterable[Measurement]:
    with open("/proc/stat") as procstat:
        procstat.readline()  # skip the first line
        for line in procstat:
            if not line.startswith("cpu"): break
            cpu, *states = line.split()
            yield Measurement(int(states[0]) // 100, {"cpu": cpu, "state": "user
↪"})
            yield Measurement(int(states[1]) // 100, {"cpu": cpu, "state": "nice
↪"})
            # ... other states
```

Alternatively, you can pass a generator directly instead of a callback, which should return iterables of *Measurement*:

```
def cpu_time_callback(states_to_include: set[str]) ->␣
↪Iterable[Iterable[Measurement]]:
    while True:
        measurements = []
```

(continues on next page)

(continued from previous page)

```
        with open("/proc/stat") as procstat:
            procstat.readline()  # skip the first line
            for line in procstat:
                if not line.startswith("cpu"): break
                cpu, *states = line.split()
                if "user" in states_to_include:
                    measurements.append(Measurement(int(states[0]) // 100, {"cpu
↪": cpu, "state": "user"}))
                if "nice" in states_to_include:
                    measurements.append(Measurement(int(states[1]) // 100, {"cpu
↪": cpu, "state": "nice"}))
                # ... other states
        yield measurements

meter.create_observable_counter(
    "system.cpu.time",
    callback=cpu_time_callback({"user", "system"}),
    unit="s",
    description="CPU time"
)
```

Parameters

- **name** – The name of the instrument to be created
- **callback** – A callback that returns an iterable of *Measurement*. Alternatively, can be a generator that yields iterables of *Measurement*.
- **unit** – The unit for measurements this instrument reports. For example, By for bytes. UCUM units are recommended.
- **description** – A description for this instrument and what it measures.

Return type *ObservableCounter*

abstract create_histogram(*name*, *unit=''*, *description=''*)

Return type *Histogram*

abstract create_observable_gauge(*name*, *callback*, *unit=''*, *description=''*)

Return type *ObservableGauge*

abstract create_observable_up_down_counter(*name*, *callback*, *unit=''*, *description=''*)

Return type *ObservableUpDownCounter*

class opentelemetry._metrics.**NoOpMeter**(*name*, *version=None*, *schema_url=None*)

Bases: *opentelemetry._metrics.Meter*

create_counter(*name*, *unit=''*, *description=''*)

Return type *Counter*

create_up_down_counter(*name*, *unit=''*, *description=''*)

Return type *UpDownCounter*

create_observable_counter(*name*, *callback*, *unit=''*, *description=''*)

Creates an observable counter instrument

An observable counter observes a monotonically increasing count by calling a provided callback which returns multiple *Measurement*.

For example, an observable counter could be used to report system CPU time periodically. Here is a basic implementation:

```
def cpu_time_callback() -> Iterable[Measurement]:
    measurements = []
    with open("/proc/stat") as procstat:
        procstat.readline()  # skip the first line
        for line in procstat:
            if not line.startswith("cpu"): break
            cpu, *states = line.split()
            measurements.append(Measurement(int(states[0]) // 100, {"cpu": cpu,
↪"state": "user"}))
            measurements.append(Measurement(int(states[1]) // 100, {"cpu": cpu,
↪"state": "nice"}))
            measurements.append(Measurement(int(states[2]) // 100, {"cpu": cpu,
↪"state": "system"}))
            # ... other states
    return measurements

meter.create_observable_counter(
    "system.cpu.time",
    callback=cpu_time_callback,
    unit="s",
    description="CPU time"
)
```

To reduce memory usage, you can use generator callbacks instead of building the full list:

```
def cpu_time_callback() -> Iterable[Measurement]:
    with open("/proc/stat") as procstat:
        procstat.readline()  # skip the first line
        for line in procstat:
            if not line.startswith("cpu"): break
            cpu, *states = line.split()
            yield Measurement(int(states[0]) // 100, {"cpu": cpu, "state": "user
↪"})
            yield Measurement(int(states[1]) // 100, {"cpu": cpu, "state": "nice
↪"})
            # ... other states
```

Alternatively, you can pass a generator directly instead of a callback, which should return iterables of *Measurement*:

```
def cpu_time_callback(states_to_include: set[str]) ->␣
↪Iterable[Iterable[Measurement]]:
    while True:
        measurements = []
```

(continues on next page)

(continued from previous page)

```
        with open("/proc/stat") as procstat:
            procstat.readline()  # skip the first line
            for line in procstat:
                if not line.startswith("cpu"): break
                cpu, *states = line.split()
                if "user" in states_to_include:
                    measurements.append(Measurement(int(states[0]) // 100, {"cpu
↪": cpu, "state": "user"}))
                if "nice" in states_to_include:
                    measurements.append(Measurement(int(states[1]) // 100, {"cpu
↪": cpu, "state": "nice"}))
                # ... other states
        yield measurements

meter.create_observable_counter(
    "system.cpu.time",
    callback=cpu_time_callback({"user", "system"}),
    unit="s",
    description="CPU time"
)
```

Parameters

- `name` – The name of the instrument to be created
- `callback` – A callback that returns an iterable of *`Measurement`*. Alternatively, can be a generator that yields iterables of *`Measurement`*.
- `unit` – The unit for measurements this instrument reports. For example, `By` for bytes. UCUM units are recommended.
- `description` – A description for this instrument and what it measures.

Return type *`ObservableCounter`*

`create_histogram`(*name*, *unit=''*, *description=''*)

Return type *`Histogram`*

`create_observable_gauge`(*name*, *callback*, *unit=''*, *description=''*)

Return type *`ObservableGauge`*

`create_observable_up_down_counter`(*name*, *callback*, *unit=''*, *description=''*)

Return type *`ObservableUpDownCounter`*

`opentelemetry._metrics.get_meter`(*name*, *version=''*, *meter_provider=None*)

Returns a *`Meter`* for use by the given instrumentation library.

This function is a convenience wrapper for opentelemetry.trace.MeterProvider.get_meter.

If meter_provider is omitted the current configured one is used.

Return type *`Meter`*

`opentelemetry._metrics.`**`set_meter_provider`**`(`*meter_provider*`)`

Sets the current global `MeterProvider` object.

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

Return type `None`

`opentelemetry._metrics.`**`get_meter_provider`**`()`

Gets the current global `MeterProvider` object.

Return type `MeterProvider`

1.1.5 opentelemetry.environment_variables package

Module contents

`opentelemetry.environment_variables.`**`OTEL_LOGS_EXPORTER = 'OTEL_LOGS_EXPORTER'`**

`OTEL_LOGS_EXPORTER`

`opentelemetry.environment_variables.`**`OTEL_METRICS_EXPORTER = 'OTEL_METRICS_EXPORTER'`**

`OTEL_METRICS_EXPORTER`

`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_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`

1.2 OpenTelemetry Python SDK

1.2.1 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

1.2.2 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
```

(continues on next page)

(continued from previous page)

```
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(
    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 probabilistically based on *`rate`*.

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.StaticSam 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.StaticS 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=None*, *schema_url=None*)

Bases: `object`

Immutable information about an instrumentation library module.

See `opentelemetry.trace.TracerProvider.get_tracer` for the meaning of these properties.

`property schema_url: Optional[str]`

Return type `Optional`[`str`, `None`]

`property version: Optional[str]`

Return type `Optional`[`str`, `None`]

`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=()*, *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_span_attributes=None*, *max_event_attributes=None*, *max_link_attributes=None*, *max_attribute_length=None*, *max_span_attribute_length=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, if any, will be used.

Limit precedence:

- If a model specific limit is set, it will be used.
- Else if the corresponding global limit is set, it will be used.
- Else if the model specific limit has a default value, the default value will be used.
- Else if the global limit has a default value, the default value will be used.

Parameters

- **`max_attributes`** (`Optional[int, None]`) – Maximum number of attributes that can be added to a span, event, and link. Environment variable: OTEL_ATTRIBUTE_COUNT_LIMIT Default: {_DEFAULT_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_span_attributes`** (`Optional[int, None]`) – Maximum number of attributes that can be added to a Span. Environment variable: OTEL_SPAN_ATTRIBUTE_COUNT_LIMIT Default: {_DEFAULT_OTEL_SPAN_ATTRIBUTE_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}
- **`max_attribute_length`** (`Optional[int, None]`) – Maximum length an attribute value can have. Values longer than the specified length will be truncated.
- **`max_span_attribute_length`** (`Optional[int, None]`) – Maximum length a span attribute value can have. Values longer than the specified length will be truncated.

`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_span_attributes=None*, *max_events_attributes=None*, *max_link_attributes=None*, *max_attributes=128*, *max_events=None*, *max_links=None*, *max_attribute_length=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=None, schema_url=None*)

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. `__name__` may not be used as this can result in different tracer names if the tracers are in different files. It is better to use a fixed string that can be imported where needed and used consistently as the name of the tracer.

  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** (Optional[str, None]) – Optional. The version string of the instrumenting library. Usually this should be the same as `pkg_resources.get_distribution(instrumenting_library_name).version`.

- **schema_url** (Optional[str, None]) – Optional. Specifies the Schema URL of the emitted telemetry.

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.

1.2.3 opentelemetry.sdk._metrics package

Warning: OpenTelemetry Python metrics are in an experimental state. The APIs within *opentelemetry.sdk._metrics* are subject to change in minor/patch releases and there are no backward compatability guarantees at this time.

Once logs become stable, this package will be be renamed to `opentelemetry.sdk.metrics`.

Submodules

class opentelemetry.sdk._metrics.**Meter**(*instrumentation_info*, *measurement_consumer*)

Bases: `opentelemetry._metrics.Meter`

create_counter(*name*, *unit=None*, *description=None*)

Return type `Counter`

create_up_down_counter(*name*, *unit=None*, *description=None*)

Return type `UpDownCounter`

create_observable_counter(*name*, *callback*, *unit=None*, *description=None*)

Creates an observable counter instrument

An observable counter observes a monotonically increasing count by calling a provided callback which returns multiple `Measurement`.

For example, an observable counter could be used to report system CPU time periodically. Here is a basic implementation:

```
def cpu_time_callback() -> Iterable[Measurement]:
    measurements = []
    with open("/proc/stat") as procstat:
        procstat.readline()  # skip the first line
        for line in procstat:
            if not line.startswith("cpu"): break
            cpu, *states = line.split()
            measurements.append(Measurement(int(states[0]) // 100, {"cpu": cpu,
↪"state": "user"}))
            measurements.append(Measurement(int(states[1]) // 100, {"cpu": cpu,
↪"state": "nice"}))
            measurements.append(Measurement(int(states[2]) // 100, {"cpu": cpu,
↪"state": "system"}))
            # ... other states
    return measurements

meter.create_observable_counter(
    "system.cpu.time",
    callback=cpu_time_callback,
    unit="s",
    description="CPU time"
)
```

To reduce memory usage, you can use generator callbacks instead of building the full list:

```
def cpu_time_callback() -> Iterable[Measurement]:
    with open("/proc/stat") as procstat:
        procstat.readline()  # skip the first line
        for line in procstat:
            if not line.startswith("cpu"): break
            cpu, *states = line.split()
            yield Measurement(int(states[0]) // 100, {"cpu": cpu, "state": "user
↪"})
```

(continues on next page)

(continued from previous page)

```
                yield Measurement(int(states[1]) // 100, {"cpu": cpu, "state": "nice
↪"})
                # ... other states
```

Alternatively, you can pass a generator directly instead of a callback, which should return iterables of `Measurement`:

```
def cpu_time_callback(states_to_include: set[str]) ->␣
↪Iterable[Iterable[Measurement]]:
    while True:
        measurements = []
        with open("/proc/stat") as procstat:
            procstat.readline()  # skip the first line
            for line in procstat:
                if not line.startswith("cpu"): break
                cpu, *states = line.split()
                if "user" in states_to_include:
                    measurements.append(Measurement(int(states[0]) // 100, {"cpu
↪": cpu, "state": "user"}))
                if "nice" in states_to_include:
                    measurements.append(Measurement(int(states[1]) // 100, {"cpu
↪": cpu, "state": "nice"}))
                # ... other states
        yield measurements

meter.create_observable_counter(
    "system.cpu.time",
    callback=cpu_time_callback({"user", "system"}),
    unit="s",
    description="CPU time"
)
```

Parameters

- **`name`** – The name of the instrument to be created
- **`callback`** – A callback that returns an iterable of `Measurement`. Alternatively, can be a generator that yields iterables of `Measurement`.
- **`unit`** – The unit for measurements this instrument reports. For example, `By` for bytes. UCUM units are recommended.
- **`description`** – A description for this instrument and what it measures.

Return type `ObservableCounter`

`create_histogram`(*name*, *unit=None*, *description=None*)

Return type `Histogram`

`create_observable_gauge`(*name*, *callback*, *unit=None*, *description=None*)

Return type `ObservableGauge`

`create_observable_up_down_counter`(*name*, *callback*, *unit=None*, *description=None*)

Return type `ObservableUpDownCounter`

class `opentelemetry.sdk._metrics.`**`MeterProvider`**(*metric_readers=()*, *resource=<opentelemetry.sdk.resources.Resource object>*, *shutdown_on_exit=True*, *views=()*, *enable_default_view=True*)

Bases: `opentelemetry._metrics.MeterProvider`

See `opentelemetry._metrics.MeterProvider`.

`force_flush`()

Return type `bool`

`shutdown`()

`get_meter`(*name*, *version=None*, *schema_url=None*)

Return type `Meter`

1.2.4 opentelemetry.sdk._logs package

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

Once logs become stable, this package will be be renamed to `opentelemetry.sdk.logs`.

Submodules

opentelemetry.sdk._logs.export

class `opentelemetry.sdk._logs.export.`**`LogExportResult`**(*value*)

Bases: `enum.Enum`

An enumeration.

`SUCCESS = 0`

`FAILURE = 1`

class `opentelemetry.sdk._logs.export.`**`LogExporter`**

Bases: `abc.ABC`

Interface for exporting logs.

Interface to be implemented by services that want to export logs received in their own format.

To export data this MUST be registered to the :class`opentelemetry.sdk._logs.LogEmitter` using a log processor.

abstract **`export`**(*batch*)

Exports a batch of logs.

Parameters **`batch`** (Sequence[`LogData`]) – The list of `LogData` objects to be exported

Returns The result of the export

abstract shutdown()

Shuts down the exporter.

Called when the SDK is shut down.

class opentelemetry.sdk._logs.export.**ConsoleLogExporter**(*out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *formatter=<function ConsoleLogExporter.<lambda>>*)

Bases: *opentelemetry.sdk._logs.export.LogExporter*

Implementation of *LogExporter* that prints log records to the console.

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

export(*batch*)

Exports a batch of logs.

Parameters **batch** (Sequence[*LogData*]) – The list of *LogData* objects to be exported

Returns The result of the export

shutdown()

Shuts down the exporter.

Called when the SDK is shut down.

class opentelemetry.sdk._logs.export.**SimpleLogProcessor**(*exporter*)

Bases: *opentelemetry.sdk._logs.LogProcessor*

This is an implementation of LogProcessor which passes received logs in the export-friendly LogData representation to the configured LogExporter, as soon as they are emitted.

emit(*log_data*)

Emits the *LogData*

shutdown()

Called when a *opentelemetry.sdk._logs.LogEmitter* is shutdown

force_flush(*timeout_millis=30000*)

Export all the received logs to the configured Exporter that have not yet been exported.

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

Return type bool

Returns False if the timeout is exceeded, True otherwise.

class opentelemetry.sdk._logs.export.**BatchLogProcessor**(*exporter*, *schedule_delay_millis=5000*, *max_export_batch_size=512*, *export_timeout_millis=30000*)

Bases: *opentelemetry.sdk._logs.LogProcessor*

This is an implementation of LogProcessor which creates batches of received logs in the export-friendly LogData representation and send to the configured LogExporter, as soon as they are emitted.

worker()

emit(*log_data*)

Adds the *LogData* to queue and notifies the waiting threads when size of queue reaches max_export_batch_size.

Return type None

shutdown()
Called when a *opentelemetry.sdk._logs.LogEmitter* is shutdown

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

Parameters **timeout_millis** (Optional[int, None]) – The maximum amount of time to wait for logs to be exported.

Return type bool

Returns False if the timeout is exceeded, True otherwise.

opentelemetry.sdk._logs.severity

class opentelemetry.sdk._logs.severity.**SeverityNumber**(*value*)
Bases: enum.Enum

Numerical value of severity.

Smaller numerical values correspond to less severe events (such as debug events), larger numerical values correspond to more severe events (such as errors and critical events).

See the Log Data Model spec for more info and how to map the severity from source format to OTLP Model.

UNSPECIFIED = 0

TRACE = 1

TRACE2 = 2

TRACE3 = 3

TRACE4 = 4

DEBUG = 5

DEBUG2 = 6

DEBUG3 = 7

DEBUG4 = 8

INFO = 9

INFO2 = 10

INFO3 = 11

INFO4 = 12

WARN = 13

WARN2 = 14

WARN3 = 15

WARN4 = 16

ERROR = 17

ERROR2 = 18

ERROR3 = 19

ERROR4 = 20

FATAL = 21

FATAL2 = 22

FATAL3 = 23

FATAL4 = 24

`opentelemetry.sdk._logs.severity.`**`std_to_otlp`**`(`*levelno*`)`

Map python log levelno as defined in https://docs.python.org/3/library/logging.html#logging-levels to OTLP log severity number.

Return type *`SeverityNumber`*

class `opentelemetry.sdk._logs.`**`LogRecord`**`(`*timestamp=None*, *trace_id=None*, *span_id=None*, *trace_flags=None*, *severity_text=None*, *severity_number=None*, *name=None*, *body=None*, *resource=None*, *attributes=None*`)`

Bases: `object`

A LogRecord instance represents an event being logged.

LogRecord instances are created and emitted via *`LogEmitter`* every time something is logged. They contain all the information pertinent to the event being logged.

`to_json``()`

Return type `str`

class `opentelemetry.sdk._logs.`**`LogData`**`(`*log_record*, *instrumentation_info*`)`

Bases: `object`

Readable LogRecord data plus associated InstrumentationLibrary.

class `opentelemetry.sdk._logs.`**`LogProcessor`**

Bases: `abc.ABC`

Interface to hook the log record emitting action.

Log processors can be registered directly using *`LogEmitterProvider.add_log_processor()`* and they are invoked in the same order as they were registered.

abstract `emit``(`*log_data*`)`

Emits the *`LogData`*

abstract `shutdown``()`

Called when a *`opentelemetry.sdk._logs.LogEmitter`* is shutdown

abstract `force_flush``(`*timeout_millis=30000*`)`

Export all the received logs to the configured Exporter that have not yet been exported.

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

Returns False if the timeout is exceeded, True otherwise.

class `opentelemetry.sdk._logs.`**`SynchronousMultiLogProcessor`**

Bases: *`opentelemetry.sdk._logs.LogProcessor`*

Implementation of class:*`LogProcessor`* that forwards all received events to a list of log processors sequentially.

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

`add_log_processor``(`*log_processor*`)`

Adds a Logprocessor to the list of log processors handled by this instance

Return type `None`

`emit`(*log_data*)
Emits the `LogData`

Return type `None`

`shutdown`()
Shutdown the log processors one by one

Return type `None`

`force_flush`(*timeout_millis=30000*)
Force flush the log processors one by one

Parameters **`timeout_millis`** (`int`) – The maximum amount of time to wait for logs to be exported. If the first n log processors exceeded the timeout then remaining log processors will not be flushed.

Return type `bool`

Returns True if all the log processors flushes the logs within timeout, False otherwise.

`class opentelemetry.sdk._logs.`**`ConcurrentMultiLogProcessor`**(*max_workers=2*)
Bases: *`opentelemetry.sdk._logs.LogProcessor`*

Implementation of *`LogProcessor`* that forwards all received events to a list of log processors in parallel.

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

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

`add_log_processor`(*log_processor*)

`emit`(*log_data*)
Emits the `LogData`

`shutdown`()
Called when a *`opentelemetry.sdk._logs.LogEmitter`* is shutdown

`force_flush`(*timeout_millis=30000*)
Force flush the log processors in parallel.

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

Return type `bool`

Returns True if all the log processors flushes the logs within timeout, False otherwise.

`class opentelemetry.sdk._logs.`**`OTLPHandler`**(*level=0*, *log_emitter=None*)
Bases: `logging.Handler`

A handler class which writes logging records, in OTLP format, to a network destination or file.

`emit`(*record*)
Emit a record.

The record is translated to OTLP format, and then sent across the pipeline.

Return type `None`

`flush`()
Flushes the logging output.

Return type `None`

`class opentelemetry.sdk._logs.`**`LogEmitter`**`(`*resource*, *multi_log_processor*, *instrumentation_info*`)`

Bases: `object`

`property resource`

`emit``(`*record*`)`

Emits the *`LogData`* by associating *`LogRecord`* and instrumentation info.

`flush``()`

Ensure all logging output has been flushed.

`class opentelemetry.sdk._logs.`**`LogEmitterProvider`**`(`*resource=<opentelemetry.sdk.resources.Resource object>*, *shutdown_on_exit=True*, *multi_log_processor=None*`)`

Bases: `object`

`property resource`

`get_log_emitter``(`*instrumenting_module_name*, *instrumenting_module_verison=''*`)`

Return type *`LogEmitter`*

`add_log_processor``(`*log_processor*`)`

Registers a new *`LogProcessor`* for this *`LogEmitterProvider`* instance.

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

`shutdown``()`

Shuts down the log processors.

`force_flush``(`*timeout_millis=30000*`)`

Force flush the log processors.

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

Return type `bool`

Returns True if all the log processors flushes the logs within timeout, False otherwise.

`opentelemetry.sdk._logs.`**`get_log_emitter_provider`**`()`

Gets the current global *`LogEmitterProvider`* object.

Return type *`LogEmitterProvider`*

`opentelemetry.sdk._logs.`**`set_log_emitter_provider`**`(`*log_emitter_provider*`)`

Sets the current global *`LogEmitterProvider`* object.

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

Return type `None`

`opentelemetry.sdk._logs.`**`get_log_emitter`**`(`*instrumenting_module_name*, *instrumenting_library_version=''*, *log_emitter_provider=None*`)`

Returns a *`LogEmitter`* for use within a python process.

This function is a convenience wrapper for opentelemetry.sdk._logs.LogEmitterProvider.get_log_emitter.

If log_emitter_provider param is omitted the current configured one is used.

Return type *`LogEmitter`*

1.2.5 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.

1.2.6 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_ATTRIBUTE_COUNT_LIMIT = 'OTEL_ATTRIBUTE_COUNT_LIMIT'**
OTEL_ATTRIBUTE_COUNT_LIMIT

The *OTEL_ATTRIBUTE_COUNT_LIMIT* represents the maximum allowed attribute count for spans, events and links. This limit is overriden by model specific limits such as OTEL_SPAN_ATTRIBUTE_COUNT_LIMIT. Default: 128

opentelemetry.sdk.environment_variables.**OTEL_ATTRIBUTE_VALUE_LENGTH_LIMIT = 'OTEL_ATTRIBUTE_VALUE_LENGTH_LIMIT'**
OTEL_ATTRIBUTE_VALUE_LENGTH_LIMIT

The *OTEL_ATTRIBUTE_VALUE_LENGTH_LIMIT* represents the maximum allowed attribute length.

opentelemetry.sdk.environment_variables.**OTEL_EVENT_ATTRIBUTE_COUNT_LIMIT = 'OTEL_EVENT_ATTRIBUTE_COUNT_LIMIT'**
OTEL_EVENT_ATTRIBUTE_COUNT_LIMIT

The *OTEL_EVENT_ATTRIBUTE_COUNT_LIMIT* represents the maximum allowed event attribute count. Default: 128

opentelemetry.sdk.environment_variables.**OTEL_LINK_ATTRIBUTE_COUNT_LIMIT = 'OTEL_LINK_ATTRIBUTE_COUNT_LIMIT'**
OTEL_LINK_ATTRIBUTE_COUNT_LIMIT

The *OTEL_LINK_ATTRIBUTE_COUNT_LIMIT* represents the maximum allowed link attribute count. Default: 128

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_ATTRIBUTE_VALUE_LENGTH_LIMIT = 'OTEL_SPAN_ATTRIBUTE_VALUE_LENGTH_LIMIT'**
OTEL_SPAN_ATTRIBUTE_VALUE_LENGTH_LIMIT

The *OTEL_SPAN_ATTRIBUTE_VALUE_LENGTH_LIMIT* represents the maximum allowed length span attribute values can have. This takes precedence over *OTEL_ATTRIBUTE_VALUE_LENGTH_LIMIT*.

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 host, and MAY contain a scheme (http or https), port and path. A scheme of https indicates a secure connection and takes precedence over the insecure configuration setting. Default: "http://localhost:4317"

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_INSECURE = 'OTEL_EXPORTER_OTLP_INSECURE'**

OTEL_EXPORTER_OTLP_INSECURE

The *OTEL_EXPORTER_OTLP_INSECURE* represents whether to enable client transport security for gRPC requests. A scheme of https takes precedence over this configuration setting. Default: False

opentelemetry.sdk.environment_variables.**OTEL_EXPORTER_OTLP_TRACES_INSECURE = 'OTEL_EXPORTER_OTLP_TRACES_INSECURE'**

OTEL_EXPORTER_OTLP_TRACES_INSECURE

The *OTEL_EXPORTER_OTLP_TRACES_INSECURE* represents whether to enable client transport security for gRPC requests for spans. A scheme of https takes precedence over the this configuration setting. Default: False

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 host, and MAY contain a scheme (http or https), port and path. A scheme of https indicates a secure connection and takes precedence over this configuration setting.

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_OTLP_METRICS_INSECURE = 'OTEL_EXPORTER_OTLP_METRICS_INSECURE'`**

OTEL_EXPORTER_OTLP_METRICS_INSECURE

The *OTEL_EXPORTER_OTLP_METRICS_INSECURE* represents whether to enable client transport security for gRPC requests for metrics. A scheme of https takes precedence over the this configuration setting. Default: False

`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.

1.3 Exporters

1.3.1 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.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"})
    )
)
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**(

1.3.2 OpenCensus Exporter

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

1.3.3 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*

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

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

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

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

span_processor = BatchSpanProcessor(otlp_exporter)

trace.get_tracer_provider().add_span_processor(span_processor)

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

API

1.3.4 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

1.4 Shims

1.4.1 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:

```python
import time

from opentelemetry import trace
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.

1.5 Examples

1.5.1 Auto-instrumentation

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

1.5.2 Basic Context

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

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

The source files of these examples are available here.

Installation

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

Run the Example

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

The output will be shown in the console.

Useful links

- OpenTelemetry
- *opentelemetry.trace package*

1.5.3 Basic Trace

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

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

The source files of these examples are available here.

Installation

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

Run the Example

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

The output will be shown in the console.

Useful links

- OpenTelemetry
- *opentelemetry.trace package*

1.5.4 Datadog Span Exporter

Warning: This exporter has been deprecated. To export your OTLP traces from OpenTelemetry SDK directly to Datadog Agent, please refer to OTLP Ingest in Datadog Agent .

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 \
```

(continues on next page)

(continued from previous page)

```
    -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
```

1.5.5 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",
```

(continues on next page)

(continued from previous page)

```
        "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 `DjangoInstrumentor().instrument()` in `main`, then Run the django app with `opentelemetry-instrument python manage.py runserver --noreload`. Repeat the steps with the client, the result should be the same.

Usage with Auto Instrumentation and uWSGI

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

```
pip install uwsgi
```

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

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

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

References

- Django
- OpenTelemetry Project
- OpenTelemetry Django extension

1.5.6 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):
```

(continues on next page)

(continued from previous page)

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

No error raised
```

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

1.5.7 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

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

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

1.5.8 OpenTelemetry Logs SDK

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

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

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

exporters:
logging:

processors:
batch:

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

Then start the Docker container:

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

```
$ python example.py
```

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

```
Resource SchemaURL:
Resource labels:
    -> telemetry.sdk.language: STRING(python)
    -> telemetry.sdk.name: STRING(opentelemetry)
    -> telemetry.sdk.version: STRING(1.8.0)
    -> service.name: STRING(shoppingcart)
    -> service.instance.id: STRING(instance-12)
InstrumentationLibraryLogs #0
InstrumentationLibraryMetrics SchemaURL:
InstrumentationLibrary __main__ 0.1
LogRecord #0
Timestamp: 2022-01-13 20:37:03.998733056 +0000 UTC
Severity: WARNING
ShortName:
Body: Jail zesty vixen who grabbed pay from quack.
Trace ID:
Span ID:
Flags: 0
LogRecord #1
Timestamp: 2022-01-13 20:37:04.082757888 +0000 UTC
Severity: ERROR
ShortName:
Body: The five boxing wizards jump quickly.
Trace ID:
Span ID:
Flags: 0
LogRecord #2
Timestamp: 2022-01-13 20:37:04.082979072 +0000 UTC
Severity: ERROR
ShortName:
Body: Hyderabad, we have a major problem.
Trace ID: 63491217958f126f727622e41d4460f3
Span ID: d90c57d6e1ca4f6c
Flags: 1
```

1.5.9 OpenTelemetry Metrics SDK

Warning: OpenTelemetry Python metrics are in an experimental state. The APIs within `opentelemetry.sdk._metrics` are subject to change in minor/patch releases and there are no backward compatability guarantees at this time.

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

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

exporters:
    logging:

processors:
    batch:

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

Then start the Docker container:

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

```
$ python example.py
```

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

```
```

TODO

1.5.10 OpenCensus Exporter

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

The source files of this example are available here.

Installation

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

Run the Example

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

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

Now, the example can be executed:

```
python collector.py
```

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

Useful links

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

1.5.11 OpenTracing Shim

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

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

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

Installation

Jaeger

Start Jaeger

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

Redis

Install Redis following the instructions.

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

```
redis-server
```

Python Dependencies

Install the Python dependencies in requirements.txt

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

Alternatively, you can install the Python dependencies separately:

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

Run the Application

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

To run the script:

```
python main.py
```

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

Jaeger UI

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

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

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

Useful links

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

PYTHON MODULE INDEX

INDEX