opentracing-python

Release 1.2

The OpenTracing Authors

Jan 20, 2023

CONTENTS

This library is a Python platform API for OpenTracing.

REQUIRED READING

In order to understand the Python platform API, one must first be familiar with the OpenTracing project and terminology more specifically.

TWO

STATUS

In the current version, `opentracing-python` provides only the API and a basic no-op implementation that can be used by instrumentation libraries to collect and propagate distributed tracing context.

Future versions will include a reference implementation utilizing an abstract Recorder interface, as well as a Zipkin-compatible Tracer.

USAGE

The work of instrumentation libraries generally consists of three steps:

1. When a service receives a new request (over HTTP or some other protocol), it uses OpenTracing's inject/extract API to continue an active trace, creating a Span object in the process. If the request does not contain an active trace, the service starts a new trace and a new *root* Span.

2. The service needs to store the current Span in some request-local storage, (called Span *activation*) where it can be retrieved from when a child Span must be created, e.g. in case of the service making an RPC to another service.

3. When making outbound calls to another service, the current Span must be retrieved from request-local storage, a child span must be created (e.g., by using the `start_child_span()` helper), and that child span must be embedded into the outbound request (e.g., using HTTP headers) via OpenTracing's inject/extract API.

Below are the code examples for the previously mentioned steps. Implementation of request-local storage needed for step 2 is specific to the service and/or frameworks / instrumentation libraries it is using, exposed as a `ScopeManager` child contained as `Tracer.scope_manager`. See details below.

3.1 Inbound request

Somewhere in your server's request handler code:

```python
def handle_request(request):
    span = before_request(request, opentracing.global_tracer())
    # store span in some request-local storage using Tracer.scope_manager,
    # using the returned `Scope` as Context Manager to ensure
    # `Span` will be cleared and (in this case) `Span.finish()` be called.
    with tracer.scope_manager.activate(span, True) as scope:
        # actual business logic
        handle_request_for_real(request)


def before_request(request, tracer):
    span_context = tracer.extract(
        format=Format.HTTP_HEADERS,
        carrier=request.headers,
    )
    span = tracer.start_span(
        operation_name=request.operation,
        child_of=span_context)
    span.set_tag('http.url', request.full_url)
```

```
    remote_ip = request.remote_ip
    if remote_ip:
        span.set_tag(tags.PEER_HOST_IPV4, remote_ip)

    caller_name = request.caller_name
    if caller_name:
        span.set_tag(tags.PEER_SERVICE, caller_name)

    remote_port = request.remote_port
    if remote_port:
        span.set_tag(tags.PEER_PORT, remote_port)

    return span
```

OUTBOUND REQUEST

Somewhere in your service that's about to make an outgoing call:

```python
from opentracing import tags
from opentracing.propagation import Format
from opentracing_instrumentation import request_context

# create and serialize a child span and use it as context manager
with before_http_request(
    request=out_request,
    current_span_extractor=request_context.get_current_span):

    # actual call
    return urllib2.urlopen(request)


def before_http_request(request, current_span_extractor):
    op = request.operation
    parent_span = current_span_extractor()
    outbound_span = opentracing.global_tracer().start_span(
        operation_name=op,
        child_of=parent_span
    )

    outbound_span.set_tag('http.url', request.full_url)
    service_name = request.service_name
    host, port = request.host_port
    if service_name:
        outbound_span.set_tag(tags.PEER_SERVICE, service_name)
    if host:
        outbound_span.set_tag(tags.PEER_HOST_IPV4, host)
    if port:
        outbound_span.set_tag(tags.PEER_PORT, port)

    http_header_carrier = {}
    opentracing.global_tracer().inject(
        span_context=outbound_span,
        format=Format.HTTP_HEADERS,
        carrier=http_header_carrier)

    for key, value in http_header_carrier.iteritems():
```

(continues on next page)

```
        request.add_header(key, value)

    return outbound_span
```

4.1 Scope and within-process propagation

For getting/setting the current active `Span` in the used request-local storage, OpenTracing requires that every `Tracer` contains a `ScopeManager` that grants access to the active `Span` through a `Scope`. Any `Span` may be transferred to another task or thread, but not `Scope`.

```python
# Access to the active span is straightforward.
scope = tracer.scope_manager.active()
if scope is not None:
    scope.span.set_tag('...', '...')
```

The common case starts a `Scope` that's automatically registered for intra-process propagation via `ScopeManager`.

Note that `start_active_span('...')` automatically finishes the span on `Scope.close()` (`start_active_span('...', finish_on_close=False)` does not finish it, in contrast).

```python
# Manual activation of the Span.
span = tracer.start_span(operation_name='someWork')
with tracer.scope_manager.activate(span, True) as scope:
    # Do things.

# Automatic activation of the Span.
# finish_on_close is a required parameter.
with tracer.start_active_span('someWork', finish_on_close=True) as scope:
    # Do things.

# Handling done through a try construct:
span = tracer.start_span(operation_name='someWork')
scope = tracer.scope_manager.activate(span, True)
try:
    # Do things.
except Exception as e:
    span.set_tag('error', '...')
finally:
    scope.close()
```

If there is a Scope, it will act as the parent to any newly started Span unless the programmer passes `ignore_active_span=True` at `start_span()`/`start_active_span()` time or specified parent context explicitly:

```python
scope = tracer.start_active_span('someWork', ignore_active_span=True)
```

Each service/framework ought to provide a specific `ScopeManager` implementation that relies on their own request-local storage (thread-local storage, or coroutine-based storage for asynchronous frameworks, for example).

4.2 Scope managers

This project includes a set of `ScopeManager` implementations under the `opentracing.scope_managers` submodule, which can be imported on demand:

```python
from opentracing.scope_managers import ThreadLocalScopeManager
```

There exist implementations for `thread-local` (the default instance of the submodule `opentracing.scope_managers`), `gevent`, Tornado, `asyncio` and `contextvars`:

```python
from opentracing.scope_managers.gevent import GeventScopeManager # requires gevent
from opentracing.scope_managers.tornado import TornadoScopeManager # requires tornado<6
from opentracing.scope_managers.asyncio import AsyncioScopeManager # fits for old
→asyncio applications, requires Python 3.4 or newer.
from opentracing.scope_managers.contextvars import ContextVarsScopeManager # for asyncio
→applications, requires Python 3.7 or newer.
```

Note that for asyncio applications it's preferable to use `ContextVarsScopeManager` instead of `AsyncioScopeManager` because of automatic parent span propagation to children coroutines, tasks or scheduled callbacks.

DEVELOPMENT

5.1 Tests

```
virtualenv env
. ./env/bin/activate
make bootstrap
make test
```

You can use tox to run tests as well.

```
tox
```

5.2 Testbed suite

A testbed suite designed to test API changes and experimental features is included under the *testbed* directory. For more information, see the Testbed README.

INSTRUMENTATION TESTS

This project has a working design of interfaces for the OpenTracing API. There is a MockTracer to facilitate unit-testing of OpenTracing Python instrumentation.

```python
from opentracing.mocktracer import MockTracer

tracer = MockTracer()
with tracer.start_span('someWork') as span:
    pass

spans = tracer.finished_spans()
someWorkSpan = spans[0]
```

6.1 Documentation

```
virtualenv env
. ./env/bin/activate
make bootstrap
make docs
```

The documentation is written to *docs/_build/html*.

6.2 LICENSE

Apache 2.0 License.

6.3 Releases

Before new release, add a summary of changes since last version to CHANGELOG.rst

```
pip install zest.releaser[recommended]
prerelease
release
git push origin master --follow-tags
python setup.py sdist upload -r pypi upload_docs -r pypi
```

```
postrelease
git push
```

6.3.1 Python API

Classes

class opentracing.**Span**(*tracer*, *context*)

Span represents a unit of work executed on behalf of a trace. Examples of spans include a remote procedure call, or a in-process method call to a sub-component. Every span in a trace may have zero or more causal parents, and these relationships transitively form a DAG. It is common for spans to have at most one parent, and thus most traces are merely tree structures.

Span implements a context manager API that allows the following usage:

```
with tracer.start_span(operation_name='go_fishing') as span:
    call_some_service()
```

In the context manager syntax it's not necessary to call *Span.finish()*

property context

Provides access to the *SpanContext* associated with this *Span*.

The *SpanContext* contains state that propagates from *Span* to *Span* in a larger trace.

> **Return type**
>
> > *SpanContext*
>
> **Returns**
>
> > the *SpanContext* associated with this *Span*.

finish(*finish_time=None*)

Indicates that the work represented by this *Span* has completed or terminated.

With the exception of the *Span.context* property, the semantics of all other *Span* methods are undefined after *Span.finish()* has been invoked.

> **Parameters**
>
> > **finish_time** (*float*) – an explicit *Span* finish timestamp as a unix timestamp per time.time()

get_baggage_item(*key*)

Retrieves value of the baggage item with the given key.

> **Parameters**
>
> > **key** (*str*) – key of the baggage item
>
> **Return type**
>
> > str
>
> **Returns**
>
> > value of the baggage item with given key, or None.

log(***kwargs*)

DEPRECATED

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

    DEPRECATED

log_kv(*key_values*, *timestamp=None*)

    Adds a log record to the *Span*.

    For example:

```
span.log_kv({
    "event": "time to first byte",
    "packet.size": packet.size()})

span.log_kv({"event": "two minutes ago"}, time.time() - 120)
```

    Parameters

- **key_values** (*dict*) – A dict of string keys and values of any type

- **timestamp** (*float*) – A unix timestamp per `time.time()`; current time if `None`

    Return type

        Span

    Returns

        the *Span* itself, for call chaining.

set_baggage_item(*key*, *value*)

    Stores a Baggage item in the *Span* as a key/value pair.

    Enables powerful distributed context propagation functionality where arbitrary application data can be carried along the full path of request execution throughout the system.

    Note 1: Baggage is only propagated to the future (recursive) children of this *Span*.

    Note 2: Baggage is sent in-band with every subsequent local and remote calls, so this feature must be used with care.

    Parameters

- **key** (*str*) – Baggage item key

- **value** (*str*) – Baggage item value

    Return type

        Span

    Returns

        itself, for chaining the calls.

set_operation_name(*operation_name*)

    Changes the operation name.

    Parameters

        operation_name (*str*) – the new operation name

    Return type

        Span

    Returns

        the *Span* itself, for call chaining.

set_tag(*key*, *value*)

> Attaches a key/value pair to the *Span*.
>
> The value must be a string, a bool, or a numeric type.
>
> If the user calls set_tag multiple times for the same key, the behavior of the *Tracer* is undefined, i.e. it is implementation specific whether the *Tracer* will retain the first value, or the last value, or pick one randomly, or even keep all of them.
>
> > **Parameters**
> >
> > - **key** (*str*) – key or name of the tag. Must be a string.
> >
> > - **value** (*string or bool or int or float*) – value of the tag.
> >
> > **Return type**
> >   *Span*
> >
> > **Returns**
> >   the *Span* itself, for call chaining.

property tracer

> Provides access to the *Tracer* that created this *Span*.
>
> > **Return type**
> >   *Tracer*
> >
> > **Returns**
> >   the *Tracer* that created this *Span*.

class opentracing.**SpanContext**

> SpanContext represents *Span* state that must propagate to descendant *Span*s and across process boundaries.
>
> SpanContext is logically divided into two pieces: the user-level "Baggage" (see *Span.set_baggage_item()* and *Span.get_baggage_item()*) that propagates across *Span* boundaries and any tracer-implementation-specific fields that are needed to identify or otherwise contextualize the associated *Span* (e.g., a (trace_id, span_id, sampled) tuple).

property baggage

> Return baggage associated with this *SpanContext*. If no baggage has been added to the *Span*, returns an empty dict.
>
> The caller must not modify the returned dictionary.
>
> See also: *Span.set_baggage_item()* / *Span.get_baggage_item()*
>
> > **Return type**
> >   dict
> >
> > **Returns**
> >   baggage associated with this *SpanContext* or {}.

class opentracing.**Scope**(*manager*, *span*)

> A scope formalizes the activation and deactivation of a *Span*, usually from a CPU standpoint. Many times a *Span* will be extant (in that *Span.finish()* has not been called) despite being in a non-runnable state from a CPU/scheduler standpoint. For instance, a *Span* representing the client side of an RPC will be unfinished but blocked on IO while the RPC is still outstanding. A scope defines when a given *Span* is scheduled and on the path.
>
> > **Parameters**
> >
> > - **manager** (*ScopeManager*) – the *ScopeManager* that created this *Scope*.

- **span** (Span) – the *Span* used for this *Scope*.

close()

> Marks the end of the active period for this *Scope*, updating *ScopeManager.active* in the process.
>
> NOTE: Calling this method more than once on a single *Scope* leads to undefined behavior.

property manager

> Returns the *ScopeManager* that created this *Scope*.
>
> > **Return type**
> > *ScopeManager*

property span

> Returns the *Span* wrapped by this *Scope*.
>
> > **Return type**
> > *Span*

class opentracing.**ScopeManager**

> The *ScopeManager* interface abstracts both the activation of a *Span* and access to an active *Span*/*Scope*.
>
> **activate**(*span*, *finish_on_close*)
>
> > Makes a *Span* active.
> >
> > > **Parameters**
> > >
> > > - **span** – the *Span* that should become active.
> > >
> > > - **finish_on_close** – whether *Span* should be automatically finished when *Scope.close()* is called.
> > >
> > > **Return type**
> > > *Scope*
> > >
> > > **Returns**
> > > a *Scope* to control the end of the active period for *span*. It is a programming error to neglect to call *Scope.close()* on the returned instance.
>
> **property active**
>
> > Returns the currently active *Scope* which can be used to access the currently active *Scope.span*.
> >
> > If there is a non-null *Scope*, its wrapped *Span* becomes an implicit parent of any newly-created *Span* at *Tracer.start_active_span()* time.
> >
> > > **Return type**
> > > *Scope*
> > >
> > > **Returns**
> > > the *Scope* that is active, or None if not available.

class opentracing.**Tracer**(*scope_manager=None*)

> Tracer is the entry point API between instrumentation code and the tracing implementation.
>
> This implementation both defines the public Tracer API, and provides a default no-op behavior.
>
> **property active_span**
>
> > Provides access to the the active *Span*. This is a shorthand for Tracer.scope_manager.active.span, and None will be returned if *Scope.span* is None.
> >
> > > **Return type**
> > > *Span*

> **Returns**
> the active *Span*.

extract(*format*, *carrier*)

Returns a *SpanContext* instance extracted from a *carrier* of the given *format*, or None if no such *SpanContext* could be found.

The type of *carrier* is determined by *format*. See the *Format* class/namespace for the built-in OpenTracing formats.

Implementations *must* raise *UnsupportedFormatException* if *format* is unknown or disallowed.

Implementations may raise *InvalidCarrierException*, *SpanContextCorruptedException*, or implementation-specific errors if there are problems with *carrier*.

> **Parameters**
> - **format** – a python object instance that represents a given carrier format. *format* may be of any type, and *format* equality is defined by python == equality.
>
> - **carrier** – the format-specific carrier object to extract from
>
> **Return type**
> *SpanContext*
>
> **Returns**
> a *SpanContext* extracted from *carrier* or None if no such *SpanContext* could be found.

inject(*span_context*, *format*, *carrier*)

Injects *span_context* into *carrier*.

The type of *carrier* is determined by *format*. See the *Format* class/namespace for the built-in OpenTracing formats.

Implementations *must* raise *UnsupportedFormatException* if *format* is unknown or disallowed.

> **Parameters**
> - **span_context** (*SpanContext*) – the *SpanContext* instance to inject
>
> - **format** (*Format*) – a python object instance that represents a given carrier format. *format* may be of any type, and *format* equality is defined by python == equality.
>
> - **carrier** – the format-specific carrier object to inject into

property scope_manager

Provides access to the current *ScopeManager*.

> **Return type**
> *ScopeManager*

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

Returns a newly started and activated *Scope*.

The returned *Scope* supports with-statement contexts. For example:

```python
with tracer.start_active_span('...') as scope:
    scope.span.set_tag('http.method', 'GET')
    do_some_work()
# Span.finish() is called as part of scope deactivation through
# the with statement.
```

It's also possible to not finish the *Span* when the *Scope* context expires:

```python
with tracer.start_active_span('...',
                              finish_on_close=False) as scope:
    scope.span.set_tag('http.method', 'GET')
    do_some_work()
# Span.finish() is not called as part of Scope deactivation as
# `finish_on_close` is `False`.
```

Parameters

- **operation_name** (*str*) – name of the operation represented by the new *Span* from the perspective of the current service.

- **child_of** (*Span or SpanContext*) – (optional) a *Span* or *SpanContext* instance representing the parent in a REFERENCE_CHILD_OF reference. If specified, the *references* parameter must be omitted.

- **references** (*list* of *Reference*) – (optional) references that identify one or more parent *SpanContext*s. (See the Reference documentation for detail).

- **tags** (*dict*) – an optional dictionary of *Span* tags. The caller gives up ownership of that dictionary, because the *Tracer* may use it as-is to avoid extra data copying.

- **start_time** (*float*) – an explicit *Span* start time as a unix timestamp per `time.time()`.

- **ignore_active_span** (*bool*) – (optional) an explicit flag that ignores the current active *Scope* and creates a root *Span*.

- **finish_on_close** (*bool*) – whether *Span* should automatically be finished when *Scope.close()* is called.

Return type

　Scope

Returns

　a *Scope*, already registered via the *ScopeManager*.

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

Starts and returns a new *Span* representing a unit of work.

Starting a root *Span* (a *Span* with no causal references):

```python
tracer.start_span('...')
```

Starting a child *Span* (see also *start_child_span()*):

```python
tracer.start_span(
    '...',
    child_of=parent_span)
```

Starting a child *Span* in a more verbose way:

```python
tracer.start_span(
    '...',
    references=[opentracing.child_of(parent_span)])
```

Parameters

- **operation_name** (*str*) – name of the operation represented by the new *Span* from the perspective of the current service.

- **child_of** (*Span or SpanContext*) – (optional) a *Span* or *SpanContext* representing the parent in a REFERENCE_CHILD_OF reference. If specified, the *references* parameter must be omitted.

- **references** (*list* of *Reference*) – (optional) references that identify one or more parent *SpanContext*s. (See the Reference documentation for detail).

- **tags** (*dict*) – an optional dictionary of *Span* tags. The caller gives up ownership of that dictionary, because the *Tracer* may use it as-is to avoid extra data copying.

- **start_time** (*float*) – an explicit Span start time as a unix timestamp per `time.time()`

- **ignore_active_span** (*bool*) – an explicit flag that ignores the current active *Scope* and creates a root *Span*.

Return type

   Span

Returns

   an already-started *Span* instance.

class opentracing.**ReferenceType**

A namespace for OpenTracing reference types.

See http://opentracing.io/spec for more detail about references, reference types, and CHILD_OF and FOL-LOWS_FROM in particular.

class opentracing.**Reference**(*type*, *referenced_context*)

A Reference pairs a reference type with a referenced *SpanContext*.

References are used by *Tracer.start_span()* to describe the relationships between *Span*s.

Tracer implementations must ignore references where referenced_context is `None`. This behavior allows for simpler code when an inbound RPC request contains no tracing information and as a result *Tracer.extract()* returns `None`:

```
parent_ref = tracer.extract(opentracing.HTTP_HEADERS, request.headers)
span = tracer.start_span(
    'operation', references=child_of(parent_ref)
)
```

See *child_of()* and *follows_from()* helpers for creating these references.

class opentracing.**Format**

A namespace for builtin carrier formats.

These static constants are intended for use in the *Tracer.inject()* and *Tracer.extract()* methods. E.g.:

```
tracer.inject(span.context, Format.BINARY, binary_carrier)
```

BINARY = 'binary'

The BINARY format represents SpanContexts in an opaque bytearray carrier.

For both *Tracer.inject()* and *Tracer.extract()* the carrier should be a bytearray instance. *Tracer.inject()* must append to the bytearray carrier (rather than replace its contents).

HTTP_HEADERS = 'http_headers'

> The HTTP_HEADERS format represents *SpanContext*s in a python `dict` mapping from character-restricted strings to strings.
>
> Keys and values in the HTTP_HEADERS carrier must be suitable for use as HTTP headers (without modification or further escaping). That is, the keys have a greatly restricted character set, casing for the keys may not be preserved by various intermediaries, and the values should be URL-escaped.
>
> NOTE: The HTTP_HEADERS carrier `dict` may contain unrelated data (e.g., arbitrary gRPC metadata). As such, the *Tracer* implementation should use a prefix or other convention to distinguish tracer-specific key:value pairs.

TEXT_MAP = 'text_map'

> The TEXT_MAP format represents *SpanContext*s in a python `dict` mapping from strings to strings.
>
> Both the keys and the values have unrestricted character sets (unlike the HTTP_HEADERS format).
>
> NOTE: The TEXT_MAP carrier `dict` may contain unrelated data (e.g., arbitrary gRPC metadata). As such, the *Tracer* implementation should use a prefix or other convention to distinguish tracer-specific key:value pairs.

Utility Functions

opentracing.**global_tracer**()

> Returns the global tracer. The default value is an instance of *opentracing.Tracer*
>
> > **Return type**
> > *Tracer*
> >
> > **Returns**
> > The global tracer instance.

opentracing.**set_global_tracer**(*value*)

> Sets the global tracer. It is an error to pass `None`.
>
> > **Parameters**
> > **value** (*Tracer*) – the *Tracer* used as global instance.

opentracing.**start_child_span**(*parent_span*, *operation_name*, *tags=None*, *start_time=None*)

> A shorthand method that starts a *child_of Span* for a given parent *Span*.
>
> Equivalent to calling:

```
parent_span.tracer().start_span(
    operation_name,
    references=opentracing.child_of(parent_span.context),
    tags=tags,
    start_time=start_time)
```

> > **Parameters**
> >
> > - **parent_span** (*Span*) – the *Span* which will act as the parent in the returned *Span*s child_of reference.
> >
> > - **operation_name** (*str*) – the operation name for the child *Span* instance
> >
> > - **tags** (*dict*) – optional dict of *Span* tags. The caller gives up ownership of that dict, because the *Tracer* may use it as-is to avoid extra data copying.

- **start_time** (*float*) – an explicit *Span* start time as a unix timestamp per `time.time()`.

    Return type
    > *Span*

    Returns
    > an already-started *Span* instance.

opentracing.**child_of**(*referenced_context=None*)

> child_of is a helper that creates CHILD_OF References.

> **Parameters**
> > **referenced_context** (*SpanContext*) – the (causal parent) *SpanContext* to reference. If None is passed, this reference must be ignored by the *Tracer*.

> **Return type**
> > *Reference*

> **Returns**
> > A reference suitable for `Tracer.start_span(..., references=...)`

opentracing.**follows_from**(*referenced_context=None*)

> follows_from is a helper that creates FOLLOWS_FROM References.

> **Parameters**
> > **referenced_context** (*SpanContext*) – the (causal parent) *SpanContext* to reference. If None is passed, this reference must be ignored by the *Tracer*.

> **Return type**
> > *Reference*

> **Returns**
> > A Reference suitable for `Tracer.start_span(..., references=...)`

Exceptions

class opentracing.**InvalidCarrierException**

> InvalidCarrierException should be used when the provided carrier instance does not match what the *format* argument requires.

> See *Tracer.inject()* and *Tracer.extract()*.

class opentracing.**SpanContextCorruptedException**

> SpanContextCorruptedException should be used when the underlying *SpanContext* state is seemingly present but not well-formed.

> See *Tracer.inject()* and *Tracer.extract()*.

class opentracing.**UnsupportedFormatException**

> UnsupportedFormatException should be used when the provided format value is unknown or disallowed by the *Tracer*.

> See *Tracer.inject()* and *Tracer.extract()*.

MockTracer

class opentracing.mocktracer.**MockTracer**(*scope_manager=None*)

MockTracer makes it easy to test the semantics of OpenTracing instrumentation.

By using a MockTracer as a *Tracer* implementation for tests, a developer can assert that *Span* properties and relationships with other **Spans** are defined as expected by instrumentation code.

By default, MockTracer registers propagators for Format.TEXT_MAP, Format.HTTP_HEADERS and Format.BINARY. The user should call *register_propagator()* for each additional inject/extract format.

extract(*format*, *carrier*)

Returns a SpanContext instance extracted from a *carrier* of the given *format*, or None if no such SpanContext could be found.

The type of *carrier* is determined by *format*. See the Format class/namespace for the built-in OpenTracing formats.

Implementations *must* raise UnsupportedFormatException if *format* is unknown or disallowed.

Implementations may raise InvalidCarrierException, SpanContextCorruptedException, or implementation-specific errors if there are problems with *carrier*.

> **Parameters**
>
> - **format** – a python object instance that represents a given carrier format. *format* may be of any type, and *format* equality is defined by python == equality.
>
> - **carrier** – the format-specific carrier object to extract from
>
> **Return type**
> *SpanContext*
>
> **Returns**
> a SpanContext extracted from *carrier* or None if no such SpanContext could be found.

finished_spans()

Return a copy of all finished **Spans** started by this MockTracer (since construction or the last call to *reset()*)

> **Return type**
> list
>
> **Returns**
> a copy of the finished **Spans**.

inject(*span_context*, *format*, *carrier*)

Injects *span_context* into *carrier*.

The type of *carrier* is determined by *format*. See the Format class/namespace for the built-in OpenTracing formats.

Implementations *must* raise UnsupportedFormatException if *format* is unknown or disallowed.

> **Parameters**
>
> - **span_context** (*SpanContext*) – the SpanContext instance to inject
>
> - **format** (*Format*) – a python object instance that represents a given carrier format. *format* may be of any type, and *format* equality is defined by python == equality.
>
> - **carrier** – the format-specific carrier object to inject into

register_propagator(*format*, *propagator*)

Register a propagator with this MockTracer.

> **Parameters**
>
> - **format** (*string*) – a *Format* identifier like *TEXT_MAP*
>
> - **propagator** (*Propagator*) – a **Propagator** instance to handle inject/extract calls involving *format*

reset()

Clear the finished **Spans** queue.

Note that this does **not** have any effect on **Spans** created by MockTracer that have not finished yet; those will still be enqueued in *finished_spans()* when they finish().

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

Returns a newly started and activated Scope.

The returned Scope supports with-statement contexts. For example:

```python
with tracer.start_active_span('...') as scope:
    scope.span.set_tag('http.method', 'GET')
    do_some_work()
# Span.finish() is called as part of scope deactivation through
# the with statement.
```

It's also possible to not finish the Span when the Scope context expires:

```python
with tracer.start_active_span('...',
                              finish_on_close=False) as scope:
    scope.span.set_tag('http.method', 'GET')
    do_some_work()
# Span.finish() is not called as part of Scope deactivation as
# `finish_on_close` is `False`.
```

> **Parameters**
>
> - **operation_name** (*str*) – name of the operation represented by the new Span from the perspective of the current service.
>
> - **child_of** (*Span or SpanContext*) – (optional) a Span or SpanContext instance representing the parent in a REFERENCE_CHILD_OF reference. If specified, the *references* parameter must be omitted.
>
> - **references** (*list* of Reference) – (optional) references that identify one or more parent SpanContexts. (See the Reference documentation for detail).
>
> - **tags** (*dict*) – an optional dictionary of Span tags. The caller gives up ownership of that dictionary, because the Tracer may use it as-is to avoid extra data copying.
>
> - **start_time** (*float*) – an explicit Span start time as a unix timestamp per time.time().
>
> - **ignore_active_span** (*bool*) – (optional) an explicit flag that ignores the current active Scope and creates a root Span.
>
> - **finish_on_close** (*bool*) – whether Span should automatically be finished when Scope.close() is called.

> **Return type**
> > *Scope*
>
> **Returns**
> > a `Scope`, already registered via the `ScopeManager`.

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

> Starts and returns a new `Span` representing a unit of work.
>
> Starting a root `Span` (a `Span` with no causal references):

```
tracer.start_span('...')
```

> Starting a child `Span` (see also `start_child_span()`):

```
tracer.start_span(
    '...',
    child_of=parent_span)
```

> Starting a child `Span` in a more verbose way:

```
tracer.start_span(
    '...',
    references=[opentracing.child_of(parent_span)])
```

> **Parameters**
>
> - **operation_name** (*str*) – name of the operation represented by the new `Span` from the perspective of the current service.
>
> - **child_of** (*Span or SpanContext*) – (optional) a `Span` or `SpanContext` representing the parent in a REFERENCE_CHILD_OF reference. If specified, the *references* parameter must be omitted.
>
> - **references** (*list* of `Reference`) – (optional) references that identify one or more parent `SpanContexts`. (See the Reference documentation for detail).
>
> - **tags** (*dict*) – an optional dictionary of `Span` tags. The caller gives up ownership of that dictionary, because the `Tracer` may use it as-is to avoid extra data copying.
>
> - **start_time** (*float*) – an explicit Span start time as a unix timestamp per `time.time()`
>
> - **ignore_active_span** (*bool*) – an explicit flag that ignores the current active `Scope` and creates a root `Span`.
>
> **Return type**
> > *Span*
>
> **Returns**
> > an already-started `Span` instance.

Scope managers

class opentracing.scope_managers.`ThreadLocalScopeManager`

> *ScopeManager* implementation that stores the current active *Scope* using thread-local storage.
>
> **activate**(*span, finish_on_close*)
>
> > Make a *Span* instance active.
> >
> > **Parameters**
> >
> > > - **span** – the *Span* that should become active.
> > >
> > > - **finish_on_close** – whether *span* should automatically be finished when `Scope.close()` is called.
> >
> > **Returns**
> >
> > > a *Scope* instance to control the end of the active period for the *Span*. It is a programming error to neglect to call `Scope.close()` on the returned instance.
>
> **property active**
>
> > Return the currently active *Scope* which can be used to access the currently active `Scope.span`.
> >
> > **Returns**
> >
> > > the *Scope* that is active, or `None` if not available.

class opentracing.scope_managers.gevent.`GeventScopeManager`

> *ScopeManager* implementation for **gevent** that stores the *Scope* in the current greenlet (`gevent.getcurrent()`).
>
> Automatic *Span* propagation from parent greenlets to their children is not provided, which needs to be done manually:

```python
def child_greenlet(span):
    # activate the parent Span, but do not finish it upon
    # deactivation. That will be done by the parent greenlet.
    with tracer.scope_manager.activate(span, finish_on_close=False):
        with tracer.start_active_span('child') as scope:
            ...

def parent_greenlet():
    with tracer.start_active_span('parent') as scope:
        ...
        gevent.spawn(child_greenlet, span).join()
        ...
```

> **activate**(*span, finish_on_close*)
>
> > Make a *Span* instance active.
> >
> > **Parameters**
> >
> > > - **span** – the *Span* that should become active.
> > >
> > > - **finish_on_close** – whether *span* should automatically be finished when `Scope.close()` is called.
> >
> > **Returns**
> >
> > > a *Scope* instance to control the end of the active period for the *Span*. It is a programming error to neglect to call `Scope.close()` on the returned instance.

property active

>   Return the currently active *Scope* which can be used to access the currently active `Scope.span`.

>   >   **Returns**

>   >   >   the *Scope* that is active, or `None` if not available.

class opentracing.scope_managers.asyncio.**AsyncioScopeManager**

>   *ScopeManager* implementation for **asyncio** that stores the *Scope* in the current `Task` (`asyncio.current_task()`), falling back to thread-local storage if none was being executed.

>   Automatic *Span* propagation from parent coroutines to their children is not provided, which needs to be done manually:

```python
async def child_coroutine(span):
    # activate the parent Span, but do not finish it upon
    # deactivation. That will be done by the parent coroutine.
    with tracer.scope_manager.activate(span, finish_on_close=False):
        with tracer.start_active_span('child') as scope:
            ...

async def parent_coroutine():
    with tracer.start_active_span('parent') as scope:
        ...
        await child_coroutine(span)
        ...
```

>   **activate**(*span*, *finish_on_close*)

>   >   Make a *Span* instance active.

>   >   >   **Parameters**

>   >   >   >   - **span** – the *Span* that should become active.

>   >   >   >   - **finish_on_close** – whether *span* should automatically be finished when `Scope.close()` is called.

>   >   If no `Task` is being executed, thread-local storage will be used to store the *Scope*.

>   >   >   **Returns**

>   >   >   >   a *Scope* instance to control the end of the active period for the *Span*. It is a programming error to neglect to call `Scope.close()` on the returned instance.

property active

>   Return the currently active *Scope* which can be used to access the currently active `Scope.span`.

>   >   **Returns**

>   >   >   the *Scope* that is active, or `None` if not available.

6.3.2 History

2.4.0 (2020-11-19)

- Use current_task from asyncio module for Python 3.9 compatibility (#138) <Michael Tannenbaum>
- Drop build support for Python 3.5 (#138) <Michael Tannenbaum>

2.3.0 (2020-01-02)

- Add AsyncioScopeManager based on contextvars and supporting Tornado 6 (#118) <Vasilii Novikov>

2.2.0 (2019-05-10)

- Fix __exit__ method of Scope class (#120) <Aliaksei Urbanski>
- Add support for Python 3.5/3.7 and fix tests (#121) <Aliaksei Urbanski>

2.1.0 (2019-04-27)

- Add support for indicating if a global tracer has been registered (#109) <Mike Goldsmith>
- Use pytest-cov==2.6.0 as 2.6.1 depends on pytest>=3.6.0 (#113) <Carlos Alberto Cortez>
- Better error handling in context managers for Span/Scope. (#101) <Carlos Alberto Cortez>
- Add log fields constants to opentracing.logs. (#99) <Carlos Alberto Cortez>
- Move opentracing.ext.tags to opentracing.tags. (#103) <Carlos Alberto Cortez>
- Add SERVICE tag (#100) <Carlos Alberto Cortez>
- Fix unclosed active scope in tests (#97) <Michał Szymański>
- Initial implementation of a global Tracer. (#95) <Carlos Alberto Cortez>

2.0.0 (2018-07-10)

- Implement ScopeManager for in-process propagation.
- Added a set of default ScopeManager implementations.
- Added testbed/ for testing API changes.
- Added MockTracer for instrumentation testing.

1.3.0 (2018-01-14)

- Added sphinx-generated documentation.
- Remove 'futures' from install_requires (#62)
- Add a harness check for unicode keys and vals (#40)
- Have the harness try all tag value types (#39)

1.2.2 (2016-10-03)

- Fix KeyError when checking kwargs for optional values

1.2.1 (2016-09-22)

- Make Span.log(self, **kwargs) smarter

1.2.0 (2016-09-21)

- Add Span.log_kv and deprecate older logging methods

1.1.0 (2016-08-06)

- Move set/get_baggage back to Span; add SpanContext.baggage
- Raise exception on unknown format

2.0.0.dev3 (2016-07-26)

- Support SpanContext

2.0.0.dev1 (2016-07-12)

- Rename ChildOf/FollowsFrom to child_of/follows_from
- Rename span_context to referee in Reference
- Document expected behavior when referee=None

2.0.0.dev0 (2016-07-11)

- Support SpanContext (and real semvers)

1.0rc4 (2016-05-21)

- Add standard tags per http://opentracing.io/data-semantics/

1.0rc3 (2016-03-22)

- No changes yet

1.0rc3 (2016-03-22)

• Move to simpler carrier formats

1.0rc2 (2016-03-11)

• Remove the Injector/Extractor layer

1.0rc1 (2016-02-24)

• Upgrade to 1.0 RC specification

0.6.3 (2016-01-16)

• Rename repository back to opentracing-python

0.6.2 (2016-01-15)

• Validate chaining of logging calls

0.6.1 (2016-01-09)

• Fix typo in the attributes API test

0.6.0 (2016-01-09)

• Change inheritance to match api-go: TraceContextSource extends codecs, Tracer extends TraceContextSource
• Create API harness

0.5.2 (2016-01-08)

• Update README and meta.

0.5.1 (2016-01-08)

• Prepare for PYPI publishing.

0.5.0 (2016-01-07)

• Remove debug flag
• Allow passing tags to start methods
• Add Span.add_tags() method

0.4.2 (2016-01-07)

• Add SPAN_KIND tag

0.4.0 (2016-01-06)

• Rename marshal -> encode

0.3.1 (2015-12-30)

• Fix std context implementation to refer to Trace Attributes instead of metadata

0.3.0 (2015-12-29)

• Rename trace tags to Trace Attributes. Rename RPC tags to PEER. Add README.

0.2.0 (2015-12-28)

• Export global *tracer* variable.

0.1.4 (2015-12-28)

• Rename RPC_SERVICE tag to make it symmetric

0.1.3 (2015-12-27)

• Allow repeated keys for span tags; add standard tag names for RPC

0.1.2 (2015-12-27)

• Move creation of child context to TraceContextSource

0.1.1 (2015-12-27)

• Add log methods

0.1.0 (2015-12-27)

• Initial public API

INDEX

Some radii options for box corners used; they were ignored as pict2e was not found

Some radii options for box corners used; they were ignored as pict2e was not found