opentracing-python

Release 1.2

Jun 27, 2018

Contents

This library is a Python platform API for OpenTracing.

CHAPTER 1

Required Reading

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

CHAPTER 2

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.

CHAPTER 3

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, 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 steps 1 and 3. Implementation of request-local storage needed for step 2 is specific to the service and/or frameworks / instrumentation libraries it is using (TODO: reference to other OSS projects with examples of instrumentation).

3.1 Inbound request

Somewhere in your server's request handler code:

```
def handle_request(request):
    span = before_request(request, opentracing.tracer)
    # use span as Context Manager to ensure span.finish() will be called
    with span:
        # store span in some request-local storage
        with RequestContext(span):
            # actual business logic
            handle_request_for_real(request)


def before_request(request, tracer):
    span_context = tracer.extract(
```

(continues on next page)

(continued from previous page)

```
        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
```

CHAPTER 4

Outbound request

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

```
# create and serialize a child span and use it as context manager
with before_http_request(
    request=out_request,
    current_span_extractor=RequestContext.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.tracer.start_span(
        operation_name=op,
        parent=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.tracer.inject(
        span=outbound_span,
        format=Format.HTTP_HEADERS,
        carrier=http_header_carrier)
```

(continues on next page)

(continued from previous page)

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

    return outbound_span
```

CHAPTER 5

Development

5.1 Tests

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

5.2 Documentation

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

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

5.3 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
```

5.3.1 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()

context

Provides access to the SpanContext associated with this Span.

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

Returns 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 *finish()* has been invoked.

Parameters **finish_time** – 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

:rtype : str :return: 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

Returns Returns the Span itself, for call chaining.

Return type *Span*

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

:rtype : Span :return: itself, for chaining the calls.

set_operation_name(*operation_name*)
Changes the operation name.

Parameters **operation_name** – the new operation name

Returns 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** – key or name of the tag. Must be a string.
- **value** – value of the tag.

Returns Returns the Span itself, for call chaining.

Return type *Span*

tracer
Provides access to the Tracer that created this Span.

Returns returns the Tracer that created this Span.

class opentracing.**SpanContext**
SpanContext represents Span state that must propagate to descendant Spans and across process boundaries.

SpanContext is logically divided into two pieces: the user-level "Baggage" (see set_baggage_item and 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 instance (e.g., a <trace_id, span_id, sampled> tuple).

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 returns baggage associated with this SpanContext or {}.

class opentracing.**Tracer**

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.

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 opentracing.propagation.Format class/namespace for the built-in OpenTracing formats.

Implementations MUST raise opentracing.UnsupportedFormatException if *format* is unknown or disallowed.

Implementations may raise opentracing.InvalidCarrierException, opentracing.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

Returns a SpanContext instance extracted from *carrier* or None if no such span context could be found.

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

Injects *span_context* into *carrier*.

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

Implementations MUST raise opentracing.UnsupportedFormatException if *format* is unknown or disallowed.

Parameters

- **span_context** – the SpanContext instance to inject
- **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

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

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** – name of the operation represented by the new span from the perspective of the current service.
- **child_of** – (optional) a Span or SpanContext instance representing the parent in a REFERENCE_CHILD_OF Reference. If specified, the *references* parameter must be omitted.
- **references** – (optional) a list of Reference objects that identify one or more parent SpanContexts. (See the Reference documentation for detail)
- **tags** – 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** – an explicit Span start time as a unix timestamp per time.time()

Returns 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 FOLLOWS_FROM in particular.

class opentracing.**Reference**

A Reference pairs a reference type with a referenced SpanContext.

References are used by Tracer.start_span() to describe the relationships between Spans.

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 SpanContexts 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 SpanContexts 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.**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** – the Span which will act as the parent in the returned Span's child_of reference.
- **operation_name** – the operation name for the child Span instance
- **tags** – 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** – an explicit Span start time as a unix timestamp per time.time().

Returns 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** – 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** – 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 span context 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().

5.3.2 History

1.2.3 (unreleased)

- Added sphinx-generated documentation.

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