opentracing-python

Release 1.2

The OpenTracing Authors

Apr 22, 2022

CONTENTS

This library is a Python platform API for OpenTracing.

CHAPTER

ONE

REQUIRED READING

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

CHAPTER

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.

CHAPTER

THREE

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:

```
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)
```

(continues on next page)

(continued from previous page)

```
    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

FOUR

OUTBOUND REQUEST

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

```
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():
        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`.

```
# 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).

```
# 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:

```
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:

```
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`:

```
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.

CHAPTER

FIVE

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.

CHAPTER

SIX

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.

```
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

Utility Functions

Exceptions

MockTracer

Scope managers

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