OpenTelemetry Python Contrib

OpenTelemetry Authors

Jan 11, 2023

OPENTELEMETRY EXPORTERS

Complimentary instrumentation and vendor-specific packages for use with the Python OpenTelemetry client.

Please note that this library is currently in _beta_, and shouldn't generally be used in production environments.

CHAPTER

ONE

INSTALLATION

There are several complimentary packages available on PyPI which can be installed separately via pip:

```
pip install opentelemetry-exporter-{exporter}
pip install opentelemetry-instrumentation-{instrumentation}
pip install opentelemetry-sdk-extension-{sdkextension}
```

A complete list of packages can be found at the Contrib repo instrumentation and Contrib repo exporter directories.

CHAPTER

TWO

EXTENSIONS

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

2.1 Installing Cutting Edge Packages

While the project is pre-1.0, there may be significant functionality that has not yet been released to PyPI. In that situation, you may want to install the packages directly from the repo. This can be done by cloning the repository and doing an editable install:

```
git clone https://github.com/open-telemetry/opentelemetry-python-contrib.git
cd opentelemetry-python-contrib
pip install -e ./instrumentation/opentelemetry-instrumentation-flask
pip install -e ./instrumentation/opentelemetry-instrumentation-botocore
pip install -e ./sdk-extension/opentelemetry-sdk-extension-aws
```

2.1.1 OpenTelemetry Datadog Exporter

2.1.2 OpenTelemetry aiohttp client Instrumentation

The opentelemetry-instrumentation-aiohttp-client package allows tracing HTTP requests made by the aiohttp client library.

Usage

Explicitly instrumenting a single client session:

```
import aiohttp
from opentelemetry.instrumentation.aiohttp_client import create_trace_config
import yarl

def strip_query_params(url: yarl.URL) -> str:
    return str(url.with_query(None))

async with aiohttp.ClientSession(trace_configs=[create_trace_config(
        # Remove all query params from the URL attribute on the span.
        url_filter=strip_query_params,
)]) as session:
```

(continues on next page)

(continued from previous page)

```
    async with session.get(url) as response:
        await response.text()
```

Instrumenting all client sessions:

```
import aiohttp
from opentelemetry.instrumentation.aiohttp_client import (
    AioHttpClientInstrumentor
)

# Enable instrumentation
AioHttpClientInstrumentor().instrument()

# Create a session and make an HTTP get request
async with aiohttp.ClientSession() as session:
    async with session.get(url) as response:
        await response.text()
```

Configuration

Request/Response hooks

Utilize request/reponse hooks to execute custom logic to be performed before/after performing a request.

```
def request_hook(span: Span, params: aiohttp.TraceRequestStartParams):
   if span and span.is_recording():
         span.set_attribute("custom_user_attribute_from_request_hook", "some-value")

def response_hook(span: Span, params: typing.Union[
             aiohttp.TraceRequestEndParams,
             aiohttp.TraceRequestExceptionParams,
         ]):
     if span and span.is_recording():
         span.set_attribute("custom_user_attribute_from_response_hook", "some-value")

AioHttpClientInstrumentor().instrument(request_hook=request_hook, response_hook=response_
↪hook)
```

API

opentelemetry.instrumentation.aiohttp_client.**create_trace_config**(*url_filter=None*, *request_hook=None*, *response_hook=None*, *tracer_provider=None*)

Create an aiohttp-compatible trace configuration.

One span is created for the entire HTTP request, including initial TCP/TLS setup if the connection doesn't exist.

By default the span name is set to the HTTP request method.

Example usage:

```
import aiohttp
from opentelemetry.instrumentation.aiohttp_client import create_trace_config

async with aiohttp.ClientSession(trace_configs=[create_trace_config()]) as session:
    async with session.get(url) as response:
        await response.text()
```

Parameters

- **url_filter** (Optional[Callable[[URL], str]]) – A callback to process the requested URL prior to adding it as a span attribute. This can be useful to remove sensitive data such as API keys or user personal information.
- **request_hook** (*Callable*) – Optional callback that can modify span name and request params.
- **response_hook** (*Callable*) – Optional callback that can modify span name and response params.
- **tracer_provider** (TracerProvider) – optional TracerProvider from which to get a Tracer

Returns
An object suitable for use with aiohttp.ClientSession.

Return type
aiohttp.TraceConfig

class opentelemetry.instrumentation.aiohttp_client.**AioHttpClientInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for aiohttp client sessions

See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

static uninstrument_session(*client_session*)

Disables instrumentation for the given session

2.1.3 OpenTelemetry aiopg Instrumentation

The integration with PostgreSQL supports the aiopg library, it can be enabled by using `AiopgInstrumentor`.

Usage

```
import aiopg
from opentelemetry.instrumentation.aiopg import AiopgInstrumentor

AiopgInstrumentor().instrument()

cnx = await aiopg.connect(database='Database')
cursor = await cnx.cursor()
await cursor.execute("INSERT INTO test (testField) VALUES (123)")
cursor.close()
cnx.close()

pool = await aiopg.create_pool(database='Database')
cnx = await pool.acquire()
cursor = await cnx.cursor()
await cursor.execute("INSERT INTO test (testField) VALUES (123)")
cursor.close()
cnx.close()
```

API

class opentelemetry.instrumentation.aiopg.**AiopgInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

instrument_connection(*connection*, *tracer_provider=None*)

Enable instrumentation in a aiopg connection.

Parameters

- **connection** – The connection to instrument.
- **tracer_provider** – The optional tracer provider to use. If omitted the current globally configured one is used.

Returns
An instrumented connection.

uninstrument_connection(*connection*)

Disable instrumentation in a aiopg connection.

Parameters
connection – The connection to uninstrument.

Returns
An uninstrumented connection.

2.1.4 OpenTelemetry ASGI Instrumentation

This library provides a ASGI middleware that can be used on any ASGI framework (such as Django, Starlette, FastAPI or Quart) to track requests timing through OpenTelemetry.

Installation

```
pip install opentelemetry-instrumentation-asgi
```

References

- OpenTelemetry Project
- OpenTelemetry Python Examples

API

The opentelemetry-instrumentation-asgi package provides an ASGI middleware that can be used on any ASGI framework (such as Django-channels / Quart) to track requests timing through OpenTelemetry.

Usage (Quart)

```
from quart import Quart
from opentelemetry.instrumentation.asgi import OpenTelemetryMiddleware

app = Quart(__name__)
app.asgi_app = OpenTelemetryMiddleware(app.asgi_app)

@app.route("/")
async def hello():
    return "Hello!"

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

Usage (Django 3.0)

Modify the application's `asgi.py` file as shown below.

```
import os
from django.core.asgi import get_asgi_application
from opentelemetry.instrumentation.asgi import OpenTelemetryMiddleware

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'asgi_example.settings')

application = get_asgi_application()
application = OpenTelemetryMiddleware(application)
```

Usage (Raw ASGI)

```
from opentelemetry.instrumentation.asgi import OpenTelemetryMiddleware

app = ...  # An ASGI application.
app = OpenTelemetryMiddleware(app)
```

Configuration

Request/Response hooks

Utilize request/reponse hooks to execute custom logic to be performed before/after performing a request. The server request hook takes in a server span and ASGI scope object for every incoming request. The client request hook is called with the internal span and an ASGI scope which is sent as a dictionary for when the method recieve is called. The client response hook is called with the internal span and an ASGI event which is sent as a dictionary for when the method send is called.

```
 def server_request_hook(span: Span, scope: dict):
     if span and span.is_recording():
         span.set_attribute("custom_user_attribute_from_request_hook", "some-value")

 def client_request_hook(span: Span, scope: dict):
     if span and span.is_recording():
         span.set_attribute("custom_user_attribute_from_client_request_hook", "some-value
↪")

 def client_response_hook(span: Span, message: dict):
     if span and span.is_recording():
         span.set_attribute("custom_user_attribute_from_response_hook", "some-value")

OpenTelemetryMiddleware().(application, server_request_hook=server_request_hook, client_
↪request_hook=client_request_hook, client_response_hook=client_response_hook)
```

Capture HTTP request and response headers

You can configure the agent to capture predefined HTTP headers as span attributes, according to the semantic convention.

Request headers

To capture predefined HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="content-type,custom_
↪request_header"
```

will extract `content-type` and `custom_request_header` from request headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Request header names in ASGI are case insensitive. So, giving header name as `CUStom-Header` in environment variable will be able capture header with name `custom-header`.

The name of the added span attribute will follow the format `http.request.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.request.header.custom_request_header = ["<value1>, <value2>"]`

Response headers

To capture predefined HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="content-type,custom_
↪response_header"
```

will extract `content-type` and `custom_response_header` from response headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Response header names captured in ASGI are case insensitive. So, giving header name as `CUStomHeader` in environment variable will be able capture header with name `customheader`.

The name of the added span attribute will follow the format `http.response.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.response.header.custom_response_header = ["<value1>, <value2>"]`

Note: Environment variable names to capture http headers are still experimental, and thus are subject to change.

API

class opentelemetry.instrumentation.asgi.**ASGIGetter**

Bases: Getter[dict]

get(*carrier*, *key*)

Getter implementation to retrieve a HTTP header value from the ASGI scope.

Parameters

- **carrier** (dict) – ASGI scope object
- **key** (str) – header name in scope

Return type
Optional[List[str]]

Returns

A list with a single string with the header value if it exists,
else None.

keys(*carrier*)

Function that can retrieve all the keys in a carrier object.

Parameters
carrier (dict) – An object which contains values that are used to construct a Context.

Return type
List[str]

Returns
list of keys from the carrier.

class opentelemetry.instrumentation.asgi.**ASGISetter**

Bases: Setter[dict]

set(*carrier*, *key*, *value*)

Sets response header values on an ASGI scope according to the spec.

Parameters

- **carrier** (dict) – ASGI scope object
- **key** (str) – response header name to set
- **value** (str) – response header value

Return type
None

Returns
None

opentelemetry.instrumentation.asgi.**collect_request_attributes**(*scope*)

Collects HTTP request attributes from the ASGI scope and returns a dictionary to be used as span creation attributes.

opentelemetry.instrumentation.asgi.**collect_custom_request_headers_attributes**(*scope*)

returns custom HTTP request headers to be added into SERVER span as span attributes Refer specification https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/semantic_conventions/http.md#http-request-and-response-headers

opentelemetry.instrumentation.asgi.**collect_custom_response_headers_attributes**(*message*)

returns custom HTTP response headers to be added into SERVER span as span attributes Refer specification https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/semantic_conventions/http.md#http-request-and-response-headers

opentelemetry.instrumentation.asgi.**get_host_port_url_tuple**(*scope*)

Returns (host, port, full_url) tuple.

opentelemetry.instrumentation.asgi.**set_status_code**(*span*, *status_code*)

Adds HTTP response attributes to span using the status_code argument.

opentelemetry.instrumentation.asgi.**get_default_span_details**(*scope*)

Default implementation for get_default_span_details :type scope: dict :param scope: the asgi scope dictionary

Return type
Tuple[str, dict]

Returns
a tuple of the span name, and any attributes to attach to the span.

class opentelemetry.instrumentation.asgi.**OpenTelemetryMiddleware**(*app*, *excluded_urls=None*, *default_span_details=None*, *server_request_hook=None*, *client_request_hook=None*, *client_response_hook=None*, *tracer_provider=None*, *meter_provider=None*)

Bases: object

The ASGI application middleware.

This class is an ASGI middleware that starts and annotates spans for any requests it is invoked with.

Parameters

- **app** – The ASGI application callable to forward requests to.
- **default_span_details** – Callback which should return a string and a tuple, representing the desired default span name and a dictionary with any additional span attributes to set. Optional: Defaults to get_default_span_details.
- **server_request_hook** (Optional[Callable[[Span, dict], None]]) – Optional callback which is called with the server span and ASGI scope object for every incoming request.
- **client_request_hook** (Optional[Callable[[Span, dict], None]]) – Optional callback which is called with the internal span and an ASGI scope which is sent as a dictionary for when the method recieve is called.
- **client_response_hook** (Optional[Callable[[Span, dict], None]]) – Optional callback which is called with the internal span and an ASGI event which is sent as a dictionary for when the method send is called.
- **tracer_provider** – The optional tracer provider to use. If omitted the current globally configured one is used.

2.1.5 OpenTelemetry asyncpg Instrumentation

Module contents

This library allows tracing PostgreSQL queries made by the asyncpg library.

Usage

```
import asyncpg
from opentelemetry.instrumentation.asyncpg import AsyncPGInstrumentor

# You can optionally pass a custom TracerProvider to AsyncPGInstrumentor.instrument()
AsyncPGInstrumentor().instrument()
conn = await asyncpg.connect(user='user', password='password',
                             database='database', host='127.0.0.1')
values = await conn.fetch('''SELECT 42;''')
```

API

class opentelemetry.instrumentation.asyncpg.**AsyncPGInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.6 OpenTelemetry Python Instrumentor

Submodules

opentelemetry.instrumentation.instrumentor package

OpenTelemetry Base Instrumentor

class opentelemetry.instrumentation.instrumentor.**BaseInstrumentor**(**args*, ***kwargs*)

Bases: ABC

An ABC for instrumentors

Child classes of this ABC should instrument specific third party libraries or frameworks either by using the opentelemetry-instrument command or by calling their methods directly.

Since every third party library or framework is different and has different instrumentation needs, more methods can be added to the child classes as needed to provide practical instrumentation to the end user.

property is_instrumented_by_opentelemetry

abstract instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

instrument(***kwargs*)

Instrument the library

This method will be called without any optional arguments by the `opentelemetry-instrument` command.

This means that calling this method directly without passing any optional values should do the very same thing that the `opentelemetry-instrument` command does.

uninstrument(***kwargs*)

Uninstrument the library

See `BaseInstrumentor.instrument` for more information regarding the usage of `kwargs`.

2.1.7 OpenTelemetry Boto Instrumentation

Instrument Boto to trace service requests.

There are two options for instrumenting code. The first option is to use the `opentelemetry-instrument` executable which will automatically instrument your Boto client. The second is to programmatically enable instrumentation via the following code:

Usage

```
from opentelemetry.instrumentation.boto import BotoInstrumentor
import boto


# Instrument Boto
BotoInstrumentor().instrument()

# This will create a span with Boto-specific attributes
ec2 = boto.ec2.connect_to_region("us-west-2")
ec2.get_all_instances()
```

API

class opentelemetry.instrumentation.boto.**BotoInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

A instrumentor for Boto

See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

opentelemetry.instrumentation.boto.**flatten_dict**(*dict_*, *sep='.'*, *prefix=''*)

Returns a normalized dict of depth 1 with keys in order of embedding

opentelemetry.instrumentation.boto.**add_span_arg_tags**(*span*, *aws_service*, *args*, *args_names*, *args_traced*)

2.1.8 OpenTelemetry Botocore Instrumentation

Instrument Botocore to trace service requests.

There are two options for instrumenting code. The first option is to use the `opentelemetry-instrument` executable which will automatically instrument your Botocore client. The second is to programmatically enable instrumentation via the following code:

Usage

```
from opentelemetry.instrumentation.botocore import BotocoreInstrumentor
import botocore


# Instrument Botocore
BotocoreInstrumentor().instrument()

# This will create a span with Botocore-specific attributes
session = botocore.session.get_session()
session.set_credentials(
    access_key="access-key", secret_key="secret-key"
)
ec2 = self.session.create_client("ec2", region_name="us-west-2")
ec2.describe_instances()
```

API

The *instrument* method accepts the following keyword args:

tracer_provider (TracerProvider) - an optional tracer provider request_hook (Callable) - a function with extra user-defined logic to be performed before performing the request this function signature is: def request_hook(span: Span, service_name: str, operation_name: str, api_params: dict) -> None response_hook (Callable) - a function with extra user-defined logic to be performed after performing the request this function signature is: def request_hook(span: Span, service_name: str, operation_name: str, result: dict) -> None

for example:

class opentelemetry.instrumentation.botocore.**BotocoreInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for Botocore.

See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.9 OpenTelemetry Celery Instrumentation

Instrument celery to trace Celery applications.

Usage

- Start broker backend

```
docker run -p 5672:5672 rabbitmq
```

- Run instrumented task

```
from opentelemetry.instrumentation.celery import CeleryInstrumentor

from celery import Celery
from celery.signals import worker_process_init

@worker_process_init.connect(weak=False)
def init_celery_tracing(*args, **kwargs):
    CeleryInstrumentor().instrument()

app = Celery("tasks", broker="amqp://localhost")
```

(continues on next page)

(continued from previous page)

```
@app.task
def add(x, y):
    return x + y

add.delay(42, 50)
```

Setting up tracing

When tracing a celery worker process, tracing and instrumention both must be initialized after the celery worker process is initialized. This is required for any tracing components that might use threading to work correctly such as the BatchSpanProcessor. Celery provides a signal called `worker_process_init` that can be used to accomplish this as shown in the example above.

API

class opentelemetry.instrumentation.celery.**CeleryGetter**

Bases: `Getter`

get(*carrier*, *key*)

Function that can retrieve zero or more values from the carrier. In the case that the value does not exist, returns None.

Parameters

- **carrier** – An object which contains values that are used to construct a Context.
- **key** – key of a field in carrier.

Returns: first value of the propagation key or None if the key doesn't exist.

keys(*carrier*)

Function that can retrieve all the keys in a carrier object.

Parameters
carrier – An object which contains values that are used to construct a Context.

Returns
list of keys from the carrier.

class opentelemetry.instrumentation.celery.**CeleryInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.10 OpenTelemetry Database API Instrumentation

The trace integration with Database API supports libraries that follow the Python Database API Specification v2.0. https://www.python.org/dev/peps/pep-0249/

Usage

```
import mysql.connector
import pyodbc

from opentelemetry.instrumentation.dbapi import trace_integration


# Ex: mysql.connector
trace_integration(mysql.connector, "connect", "mysql")
# Ex: pyodbc
trace_integration(pyodbc, "Connection", "odbc")
```

API

opentelemetry.instrumentation.dbapi.**trace_integration**(*connect_module*, *connect_method_name*, *database_system*, *connection_attributes=None*, *tracer_provider=None*, *capture_parameters=False*, *enable_commenter=False*, *db_api_integration_factory=None*)

Integrate with DB API library. https://www.python.org/dev/peps/pep-0249/

Parameters

- **connect_module** (Callable[..., Any]) – Module name where connect method is available.
- **connect_method_name** (str) – The connect method name.
- **database_system** (str) – An identifier for the database management system (DBMS) product being used.
- **connection_attributes** (Dict) – Attribute names for database, port, host and user in Connection object.
- **tracer_provider** (Optional[TracerProvider]) – The opentelemetry.trace.TracerProvider to use. If omitted the current configured one is used.
- **capture_parameters** (bool) – Configure if db.statement.parameters should be captured.

opentelemetry.instrumentation.dbapi.**wrap_connect**(*name*, *connect_module*, *connect_method_name*, *database_system*, *connection_attributes=None*, *version=''*, *tracer_provider=None*, *capture_parameters=False*, *enable_commenter=False*, *db_api_integration_factory=None*, *commenter_options=None*)

Integrate with DB API library. https://www.python.org/dev/peps/pep-0249/

Parameters

- **connect_module** (Callable[..., Any]) – Module name where connect method is available.
- **connect_method_name** (str) – The connect method name.
- **database_system** (str) – An identifier for the database management system (DBMS) product being used.
- **connection_attributes** (Dict) – Attribute names for database, port, host and user in Connection object.
- **tracer_provider** (Optional[TracerProvider]) – The opentelemetry.trace.TracerProvider to use. If omitted the current configured one is used.
- **capture_parameters** (bool) – Configure if db.statement.parameters should be captured.
- **enable_commenter** (bool) – Flag to enable/disable sqlcommenter.
- **commenter_options** (dict) – Configurations for tags to be appended at the sql query.

opentelemetry.instrumentation.dbapi.**unwrap_connect**(*connect_module*, *connect_method_name*)

Disable integration with DB API library. https://www.python.org/dev/peps/pep-0249/

Parameters

- **connect_module** (Callable[..., Any]) – Module name where the connect method is available.
- **connect_method_name** (str) – The connect method name.

opentelemetry.instrumentation.dbapi.**instrument_connection**(*name*, *connection*, *database_system*, *connection_attributes=None*, *version=''*, *tracer_provider=None*, *capture_parameters=False*, *enable_commenter=False*, *commenter_options=None*)

Enable instrumentation in a database connection.

Parameters

- **connection** – The connection to instrument.
- **database_system** (str) – An identifier for the database management system (DBMS) product being used.
- **connection_attributes** (Dict) – Attribute names for database, port, host and user in a connection object.
- **tracer_provider** (Optional[TracerProvider]) – The opentelemetry.trace.TracerProvider to use. If omitted the current configured one is used.
- **capture_parameters** (bool) – Configure if db.statement.parameters should be captured.

- **enable_commenter** (bool) – Flag to enable/disable sqlcommenter.
- **commenter_options** (dict) – Configurations for tags to be appended at the sql query.

Returns
An instrumented connection.

opentelemetry.instrumentation.dbapi.**uninstrument_connection**(*connection*)

Disable instrumentation in a database connection.

Parameters
connection – The connection to uninstrument.

Returns
An uninstrumented connection.

class opentelemetry.instrumentation.dbapi.**DatabaseApiIntegration**(*name*, *database_system*, *connection_attributes=None*, *version=''*, *tracer_provider=None*, *capture_parameters=False*, *enable_commenter=False*, *commenter_options=None*, *connect_module=None*)

Bases: object

wrapped_connection(*connect_method*, *args*, *kwargs*)

Add object proxy to connection object.

get_connection_attributes(*connection*)

opentelemetry.instrumentation.dbapi.**get_traced_connection_proxy**(*connection*, *db_api_integration*, **args*, ***kwargs*)

class opentelemetry.instrumentation.dbapi.**CursorTracer**(*db_api_integration*)

Bases: object

get_operation_name(*cursor*, *args*)

get_statement(*cursor*, *args*)

traced_execution(*cursor*, *query_method*, **args*, ***kwargs*)

opentelemetry.instrumentation.dbapi.**get_traced_cursor_proxy**(*cursor*, *db_api_integration*, **args*, ***kwargs*)

2.1.11 OpenTelemetry Django Instrumentation

Instrument django to trace Django applications.

SQLCOMMENTER

You can optionally configure Django instrumentation to enable sqlcommenter which enriches the query with contextual information.

Usage

```
from opentelemetry.instrumentation.django import DjangoInstrumentor

DjangoInstrumentor().instrument(is_sql_commentor_enabled=True)
```

For example,

```
Invoking Users().objects.all() will lead to sql query "select * from auth_users" but␣
↪when SQLCommenter is enabled
the query will get appended with some configurable tags like "select * from auth_users /
↪*metrics=value*/;"
```

SQLCommenter Configurations

We can configure the tags to be appended to the sqlquery log by adding below variables to the settings.py

SQLCOMMENTER_WITH_FRAMEWORK = True(Default) or False

For example, :: Enabling this flag will add django framework and it's version which is */framework='django%3A2.2.3'/*

SQLCOMMENTER_WITH_CONTROLLER = True(Default) or False

For example, :: Enabling this flag will add controller name that handles the request */controller='index'/*

SQLCOMMENTER_WITH_ROUTE = True(Default) or False

For example, :: Enabling this flag will add url path that handles the request */route='polls/'/*

SQLCOMMENTER_WITH_APP_NAME = True(Default) or False

For example, :: Enabling this flag will add app name that handles the request */app_name='polls'/*

SQLCOMMENTER_WITH_OPENTELEMETRY = True(Default) or False

For example, :: Enabling this flag will add opentelemetry traceparent */traceparent='00-fd720cffceba94bbf75940ff3caaf3cc-4fd1a2bdacf56388-01'/*

SQLCOMMENTER_WITH_DB_DRIVER = True(Default) or False

For example, :: Enabling this flag will add name of the db driver */db_driver='django.db.backends.postgresql'/*

Usage

```
from opentelemetry.instrumentation.django import DjangoInstrumentor

DjangoInstrumentor().instrument()
```

Configuration

Exclude lists

To exclude certain URLs from being tracked, set the environment variable `OTEL_PYTHON_DJANGO_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` as fallback) with comma delimited regexes representing which URLs to exclude.

For example,

```
export OTEL_PYTHON_DJANGO_EXCLUDED_URLS="client/.*/info,healthcheck"
```

will exclude requests such as `https://site/client/123/info` and `https://site/xyz/healthcheck`.

Request attributes

To extract certain attributes from Django's request object and use them as span attributes, set the environment variable `OTEL_PYTHON_DJANGO_TRACED_REQUEST_ATTRS` to a comma delimited list of request attribute names.

For example,

```
export OTEL_PYTHON_DJANGO_TRACED_REQUEST_ATTRS='path_info,content_type'
```

will extract path_info and content_type attributes from every traced request and add them as span attritbues.

Django Request object reference: https://docs.djangoproject.com/en/3.1/ref/request-response/#attributes

Request and Response hooks

The instrumentation supports specifying request and response hooks. These are functions that get called back by the instrumentation right after a Span is created for a request and right before the span is finished while processing a response. The hooks can be configured as follows:

```
def request_hook(span, request):
    pass

def response_hook(span, request, response):
    pass

DjangoInstrumentation().instrument(request_hook=request_hook, response_hook=response_
↪hook)
```

Django Request object: https://docs.djangoproject.com/en/3.1/ref/request-response/#httprequest-objects Django Response object: https://docs.djangoproject.com/en/3.1/ref/request-response/#httpresponse-objects

Capture HTTP request and response headers

You can configure the agent to capture predefined HTTP headers as span attributes, according to the semantic convention.

Request headers

To capture predefined HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="content_type,custom_
↪request_header"
```

will extract content_type and custom_request_header from request headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Request header names in django are case insensitive. So, giving header name as `CUStom_Header` in environment variable will be able capture header with name `custom-header`.

The name of the added span attribute will follow the format `http.request.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.request.header.custom_request_header = ["<value1>, <value2>"]`

Response headers

To capture predefined HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="content_type,custom_
↪response_header"
```

will extract content_type and custom_response_header from response headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Response header names captured in django are case insensitive. So, giving header name as `CUStomHeader` in environment variable will be able capture header with name `customheader`.

The name of the added span attribute will follow the format `http.response.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.response.header.custom_response_header = ["<value1>, <value2>"]`

API

class opentelemetry.instrumentation.django.**DjangoInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for Django

See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.12 OpenTelemetry Falcon Instrumentation

2.1.13 OpenTelemetry FastAPI Instrumentation

This library provides automatic and manual instrumentation of FastAPI web frameworks, instrumenting http requests served by applications utilizing the framework.

auto-instrumentation using the opentelemetry-instrumentation package is also supported.

Installation

```
pip install opentelemetry-instrumentation-fastapi
```

References

- OpenTelemetry Project
- OpenTelemetry Python Examples

API

Usage

```
import fastapi
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor

app = fastapi.FastAPI()

@app.get("/foobar")
async def foobar():
    return {"message": "hello world"}

FastAPIInstrumentor.instrument_app(app)
```

Configuration

Exclude lists

To exclude certain URLs from being tracked, set the environment variable `OTEL_PYTHON_FASTAPI_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` as fallback) with comma delimited regexes representing which URLs to exclude.

For example,

```
export OTEL_PYTHON_FASTAPI_EXCLUDED_URLS="client/.*/info,healthcheck"
```

will exclude requests such as `https://site/client/123/info` and `https://site/xyz/healthcheck`.

You can also pass the comma delimited regexes to the `instrument_app` method directly:

```
FastAPIInstrumentor.instrument_app(app, excluded_urls="client/.*/info,healthcheck")
```

Request/Response hooks

Utilize request/reponse hooks to execute custom logic to be performed before/after performing a request. The server request hook takes in a server span and ASGI scope object for every incoming request. The client request hook is called with the internal span and an ASGI scope which is sent as a dictionary for when the method recieve is called. The client response hook is called with the internal span and an ASGI event which is sent as a dictionary for when the method send is called.

```
def server_request_hook(span: Span, scope: dict):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_request_hook", "some-value")

def client_request_hook(span: Span, scope: dict):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_client_request_hook", "some-value
↪")

def client_response_hook(span: Span, message: dict):
    if span and span.is_recording():
```

(continues on next page)

(continued from previous page)

```
        span.set_attribute("custom_user_attribute_from_response_hook", "some-value")

FastAPIInstrumentor().instrument(server_request_hook=server_request_hook, client_request_
↪hook=client_request_hook, client_response_hook=client_response_hook)
```

Capture HTTP request and response headers

You can configure the agent to capture predefined HTTP headers as span attributes, according to the semantic convention.

Request headers

To capture predefined HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="content-type,custom_
↪request_header"
```

will extract `content-type` and `custom_request_header` from request headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Request header names in fastapi are case insensitive. So, giving header name as `CUStom-Header` in environment variable will be able capture header with name `custom-header`.

The name of the added span attribute will follow the format `http.request.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.request.header.custom_request_header = ["<value1>, <value2>"]`

Response headers

To capture predefined HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="content-type,custom_
↪response_header"
```

will extract `content-type` and `custom_response_header` from response headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Response header names captured in fastapi are case insensitive. So, giving header name as `CUStomHeader` in environment variable will be able capture header with name `customheader`.

The name of the added span attribute will follow the format `http.response.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.response.header.custom_response_header = ["<value1>, <value2>"]`

Note: Environment variable names to capture http headers are still experimental, and thus are subject to change.

API

class opentelemetry.instrumentation.fastapi.**FastAPIInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for FastAPI

See *BaseInstrumentor*

static instrument_app(*app*, *server_request_hook=None*, *client_request_hook=None*, *client_response_hook=None*, *tracer_provider=None*, *excluded_urls=None*)

Instrument an uninstrumented FastAPI application.

static uninstrument_app(*app*)

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.14 OpenTelemetry Flask Instrumentation

This library builds on the OpenTelemetry WSGI middleware to track web requests in Flask applications. In addition to opentelemetry-util-http, it supports Flask-specific features such as:

- The Flask url rule pattern is used as the Span name.
- The `http.route` Span attribute is set so that one can see which URL rule matched a request.

Usage

```
from flask import Flask
from opentelemetry.instrumentation.flask import FlaskInstrumentor

app = Flask(__name__)

FlaskInstrumentor().instrument_app(app)

@app.route("/")
def hello():
    return "Hello!"

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

Configuration

Exclude lists

To exclude certain URLs from being tracked, set the environment variable `OTEL_PYTHON_FLASK_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` as fallback) with comma delimited regexes representing which URLs to exclude.

For example,

```
export OTEL_PYTHON_FLASK_EXCLUDED_URLS="client/.*/info,healthcheck"
```

will exclude requests such as `https://site/client/123/info` and `https://site/xyz/healthcheck`.

You can also pass the comma delimited regexes to the `instrument_app` method directly:

```
FlaskInstrumentor().instrument_app(app, excluded_urls="client/.*/info,healthcheck")
```

Request/Response hooks

Utilize request/reponse hooks to execute custom logic to be performed before/after performing a request. Environ is an instance of WSGIEnvironment (flask.request.environ). Response_headers is a list of key-value (tuples) representing the response headers returned from the response.

```
def request_hook(span: Span, environ: WSGIEnvironment):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_request_hook", "some-value")

def response_hook(span: Span, status: str, response_headers: List):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_response_hook", "some-value")

FlaskInstrumentation().instrument(request_hook=request_hook, response_hook=response_hook)
```

Flask Request object reference: https://flask.palletsprojects.com/en/2.0.x/api/#flask.Request

Capture HTTP request and response headers

You can configure the agent to capture predefined HTTP headers as span attributes, according to the semantic convention.

Request headers

To capture predefined HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="content-type,custom_
↪request_header"
```

will extract `content-type` and `custom_request_header` from request headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Request header names in flask are case insensitive and - characters are replaced by _. So, giving header name as `CUStom_Header` in environment variable will be able capture header with name `custom-header`.

The name of the added span attribute will follow the format `http.request.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.request.header.custom_request_header = ["<value1>, <value2>"]`

Response headers

To capture predefined HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="content-type,custom_
↪response_header"
```

will extract `content-type` and `custom_response_header` from response headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Response header names captured in flask are case insensitive. So, giving header name as `CUStomHeader` in environment variable will be able capture header with name `customheader`.

The name of the added span attribute will follow the format `http.response.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.response.header.custom_response_header = ["<value1>, <value2>"]`

Note: Environment variable names to capture http headers are still experimental, and thus are subject to change.

API

opentelemetry.instrumentation.flask.**get_default_span_name**()

class opentelemetry.instrumentation.flask.**FlaskInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for flask.Flask

See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

```
def instrumentation_dependencies(self) -> Collection[str]:
    return ['requests ~= 1.0']
```

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

static instrument_app(*app*, *request_hook=None*, *response_hook=None*, *tracer_provider=None*, *excluded_urls=None*, *meter_provider=None*)

static uninstrument_app(*app*)

2.1.15 OpenTelemetry gRPC Instrumentation

Module contents

2.1.16 OpenTelemetry HTTPX Instrumentation

This library allows tracing HTTP requests made by the httpx library.

Installation

```
pip install opentelemetry-instrumentation-httpx
```

Usage

Instrumenting all clients

When using the instrumentor, all clients will automatically trace requests.

```
import httpx
from opentelemetry.instrumentation.httpx import HTTPXClientInstrumentor

url = "https://httpbin.org/get"
HTTPXClientInstrumentor().instrument()

with httpx.Client() as client:
     response = client.get(url)

async with httpx.AsyncClient() as client:
     response = await client.get(url)
```

Instrumenting single clients

If you only want to instrument requests for specific client instances, you can use the *instrument_client* method.

```
import httpx
from opentelemetry.instrumentation.httpx import HTTPXClientInstrumentor

url = "https://httpbin.org/get"

with httpx.Client(transport=telemetry_transport) as client:
    HTTPXClientInstrumentor.instrument_client(client)
    response = client.get(url)

async with httpx.AsyncClient(transport=telemetry_transport) as client:
    HTTPXClientInstrumentor.instrument_client(client)
    response = await client.get(url)
```

Uninstrument

If you need to uninstrument clients, there are two options available.

```
import httpx
from opentelemetry.instrumentation.httpx import HTTPXClientInstrumentor

HTTPXClientInstrumentor().instrument()
client = httpx.Client()

# Uninstrument a specific client
HTTPXClientInstrumentor.uninstrument_client(client)

# Uninstrument all clients
HTTPXClientInstrumentor().uninstrument()
```

Using transports directly

If you don't want to use the instrumentor class, you can use the transport classes directly.

```
import httpx
from opentelemetry.instrumentation.httpx import (
    AsyncOpenTelemetryTransport,
    SyncOpenTelemetryTransport,
)

url = "https://httpbin.org/get"
transport = httpx.HTTPTransport()
telemetry_transport = SyncOpenTelemetryTransport(transport)

with httpx.Client(transport=telemetry_transport) as client:
    response = client.get(url)

transport = httpx.AsyncHTTPTransport()
telemetry_transport = AsyncOpenTelemetryTransport(transport)

async with httpx.AsyncClient(transport=telemetry_transport) as client:
    response = await client.get(url)
```

Request and response hooks

The instrumentation supports specifying request and response hooks. These are functions that get called back by the instrumentation right after a span is created for a request and right before the span is finished while processing a response.

Note: The request hook receives the raw arguments provided to the transport layer. The response hook receives the raw return values from the transport layer.

The hooks can be configured as follows:

```
from opentelemetry.instrumentation.httpx import HTTPXClientInstrumentor

def request_hook(span, request):
    # method, url, headers, stream, extensions = request
    pass

def response_hook(span, request, response):
    # method, url, headers, stream, extensions = request
    # status_code, headers, stream, extensions = response
    pass

HTTPXClientInstrumentor().instrument(request_hook=request_hook, response_hook=response_
↪hook)
```

Or if you are using the transport classes directly:

```
from opentelemetry.instrumentation.httpx import SyncOpenTelemetryTransport

def request_hook(span, request):
    # method, url, headers, stream, extensions = request
    pass

def response_hook(span, request, response):
    # method, url, headers, stream, extensions = request
    # status_code, headers, stream, extensions = response
    pass

transport = httpx.HTTPTransport()
telemetry_transport = SyncOpenTelemetryTransport(
    transport,
    request_hook=request_hook,
    response_hook=response_hook
)
```

API

Usage

Instrumenting all clients

When using the instrumentor, all clients will automatically trace requests.

```
import httpx
from opentelemetry.instrumentation.httpx import HTTPXClientInstrumentor

url = "https://httpbin.org/get"
HTTPXClientInstrumentor().instrument()

with httpx.Client() as client:
     response = client.get(url)

async with httpx.AsyncClient() as client:
     response = await client.get(url)
```

Instrumenting single clients

If you only want to instrument requests for specific client instances, you can use the *instrument_client* method.

```
import httpx
from opentelemetry.instrumentation.httpx import HTTPXClientInstrumentor

url = "https://httpbin.org/get"

with httpx.Client(transport=telemetry_transport) as client:
    HTTPXClientInstrumentor.instrument_client(client)
    response = client.get(url)
```

(continues on next page)

(continued from previous page)

```
async with httpx.AsyncClient(transport=telemetry_transport) as client:
    HTTPXClientInstrumentor.instrument_client(client)
    response = await client.get(url)
```

Uninstrument

If you need to uninstrument clients, there are two options available.

```
import httpx
from opentelemetry.instrumentation.httpx import HTTPXClientInstrumentor

HTTPXClientInstrumentor().instrument()
client = httpx.Client()

# Uninstrument a specific client
HTTPXClientInstrumentor.uninstrument_client(client)

# Uninstrument all clients
HTTPXClientInstrumentor().uninstrument()
```

Using transports directly

If you don't want to use the instrumentor class, you can use the transport classes directly.

```
import httpx
from opentelemetry.instrumentation.httpx import (
    AsyncOpenTelemetryTransport,
    SyncOpenTelemetryTransport,
)

url = "https://httpbin.org/get"
transport = httpx.HTTPTransport()
telemetry_transport = SyncOpenTelemetryTransport(transport)

with httpx.Client(transport=telemetry_transport) as client:
    response = client.get(url)

transport = httpx.AsyncHTTPTransport()
telemetry_transport = AsyncOpenTelemetryTransport(transport)

async with httpx.AsyncClient(transport=telemetry_transport) as client:
    response = await client.get(url)
```

Request and response hooks

The instrumentation supports specifying request and response hooks. These are functions that get called back by the instrumentation right after a span is created for a request and right before the span is finished while processing a response.

Note: The request hook receives the raw arguments provided to the transport layer. The response hook receives the raw return values from the transport layer.

The hooks can be configured as follows:

```
from opentelemetry.instrumentation.httpx import HTTPXClientInstrumentor

def request_hook(span, request):
    # method, url, headers, stream, extensions = request
    pass

def response_hook(span, request, response):
    # method, url, headers, stream, extensions = request
    # status_code, headers, stream, extensions = response
    pass

HTTPXClientInstrumentor().instrument(request_hook=request_hook, response_hook=response_
↪hook)
```

Or if you are using the transport classes directly:

```
from opentelemetry.instrumentation.httpx import SyncOpenTelemetryTransport

def request_hook(span, request):
    # method, url, headers, stream, extensions = request
    pass

def response_hook(span, request, response):
    # method, url, headers, stream, extensions = request
    # status_code, headers, stream, extensions = response
    pass

transport = httpx.HTTPTransport()
telemetry_transport = SyncOpenTelemetryTransport(
    transport,
    request_hook=request_hook,
    response_hook=response_hook
)
```

API

class opentelemetry.instrumentation.httpx.**RequestInfo**(*method*, *url*, *headers*, *stream*, *extensions*)

Bases: NamedTuple

method: bytes

Alias for field number 0

url: Tuple[bytes, bytes, Optional[int], bytes]

Alias for field number 1

headers: Optional[List[Tuple[bytes, bytes]]]

Alias for field number 2

stream: Optional[Union[SyncByteStream, AsyncByteStream]]

Alias for field number 3

extensions: Optional[dict]

Alias for field number 4

class opentelemetry.instrumentation.httpx.**ResponseInfo**(*status_code*, *headers*, *stream*, *extensions*)

Bases: NamedTuple

status_code: int

Alias for field number 0

headers: Optional[List[Tuple[bytes, bytes]]]

Alias for field number 1

stream: Iterable[bytes]

Alias for field number 2

extensions: Optional[dict]

Alias for field number 3

class opentelemetry.instrumentation.httpx.**SyncOpenTelemetryTransport**(*transport*, *tracer_provider=None*, *request_hook=None*, *response_hook=None*)

Bases: BaseTransport

Sync transport class that will trace all requests made with a client.

Parameters

- **transport** (BaseTransport) – SyncHTTPTransport instance to wrap
- **tracer_provider** (Optional[TracerProvider]) – Tracer provider to use
- **request_hook** (Optional[Callable[[Span, *RequestInfo*], None]]) – A hook that receives the span and request that is called right after the span is created
- **response_hook** (Optional[Callable[[Span, *RequestInfo*, *ResponseInfo*], None]]) – A hook that receives the span, request, and response that is called right before the span ends

handle_request(**args*, ***kwargs*)

Add request info to span.

Return type

Union[Tuple[int, List[Tuple[bytes, bytes]], SyncByteStream, dict], Response]

class opentelemetry.instrumentation.httpx.**AsyncOpenTelemetryTransport**(*transport*, *tracer_provider=None*, *request_hook=None*, *response_hook=None*)

Bases: AsyncBaseTransport

Async transport class that will trace all requests made with a client.

Parameters

- **transport** (AsyncBaseTransport) – AsyncHTTPTransport instance to wrap
- **tracer_provider** (Optional[TracerProvider]) – Tracer provider to use
- **request_hook** (Optional[Callable[[Span, *RequestInfo*], None]]) – A hook that receives the span and request that is called right after the span is created
- **response_hook** (Optional[Callable[[Span, *RequestInfo*, *ResponseInfo*], None]]) – A hook that receives the span, request, and response that is called right before the span ends

async **handle_async_request**(**args*, ***kwargs*)

Add request info to span.

Return type
Union[Tuple[int, List[Tuple[bytes, bytes]], AsyncByteStream, dict], Response]

class opentelemetry.instrumentation.httpx.**HTTPXClientInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for httpx Client and AsyncClient

See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

static **instrument_client**(*client*, *tracer_provider=None*, *request_hook=None*, *response_hook=None*)

Instrument httpx Client or AsyncClient

Parameters

- **client** (Union[Client, AsyncClient]) – The httpx Client or AsyncClient instance
- **tracer_provider** (TracerProvider) – A TracerProvider, defaults to global
- **request_hook** (Optional[Callable[[Span, *RequestInfo*], None]]) – A hook that receives the span and request that is called right after the span is created
- **response_hook** (Optional[Callable[[Span, *RequestInfo*, *ResponseInfo*], None]]) – A hook that receives the span, request, and response that is called right before the span ends

Return type
None

static uninstrument_client(*client*)

Disables instrumentation for the given client instance

Parameters
client (Union[Client, AsyncClient]) – The httpx Client or AsyncClient instance

2.1.17 OpenTelemetry Jinja2 Instrumentation

Usage

The OpenTelemetry `jinja2` integration traces templates loading, compilation and rendering.

Usage

```
from jinja2 import Environment, FileSystemLoader
from opentelemetry.instrumentation.jinja2 import Jinja2Instrumentor


Jinja2Instrumentor().instrument()

env = Environment(loader=FileSystemLoader("templates"))
template = env.get_template("mytemplate.html")
```

API

class opentelemetry.instrumentation.jinja2.**Jinja2Instrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for jinja2

See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return [‘requests ~= 1.0’]

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.18 OpenTelemetry Logging Instrumentation

The OpenTelemetry `logging` integration automatically injects tracing context into log statements.

The integration registers a custom log record factory with the the standard library logging module that automatically inject tracing context into log record objects. Optionally, the integration can also call `logging.basicConfig()` to set a logging format with placeholders for span ID, trace ID and service name.

The following keys are injected into log record objects by the factory:

- `otelSpanID`
- `otelTraceID`
- `otelServiceName`

The integration uses the following logging format by default:

```
%(asctime)s %(levelname)s [%(name)s] [%(filename)s:%(lineno)d] [trace_id=%(otelTraceID)s␣
↪span_id=%(otelSpanID)s resource.service.name=%(otelServiceName)s] - %(message)s
```

Enable trace context injection

The integration is opt-in and must be enabled explicitly by setting the environment variable `OTEL_PYTHON_LOG_CORRELATION` to `true`.

The integration always registers the custom factory that injects the tracing context into the log record objects. Setting `OTEL_PYTHON_LOG_CORRELATION` to `true` calls `logging.basicConfig()` to set a logging format that actually makes use of the injected variables.

Environment variables

`OTEL_PYTHON_LOG_CORRELATION`

This env var must be set to `true` in order to enable trace context injection into logs by calling `logging.basicConfig()` and setting a logging format that makes use of the injected tracing variables.

Alternatively, `set_logging_format` argument can be set to `True` when initializing the `LoggingInstrumentor` class to achieve the same effect.

```
LoggingInstrumentor(set_logging_format=True)
```

The default value is `false`.

`OTEL_PYTHON_LOG_FORMAT`

This env var can be used to instruct the instrumentation to use a custom logging format.

Alternatively, a custom logging format can be passed to the `LoggingInstrumentor` as the `logging_format` argument. For example:

```
LoggingInstrumentor(logging_format='%(msg)s [span_id=%(span_id)s]')
```

The default value is:

```
%(asctime)s %(levelname)s [%(name)s] [%(filename)s:%(lineno)d] [trace_id=%(otelTraceID)s␣
↪span_id=%(otelSpanID)s resource.service.name=%(otelServiceName)s] - %(message)s
```

OTEL_PYTHON_LOG_LEVEL

This env var can be used to set a custom logging level.

Alternatively, log level can be passed to the `LoggingInstrumentor` during initialization. For example:

```
LoggingInstrumentor(log_level=logging.DEBUG)
```

The default value is `info`.

Options are:

- `info`
- `error`
- `debug`
- `warning`

Manually calling logging.basicConfig

`logging.basicConfig()` can be called to set a global logging level and format. Only the first ever call has any effect on the global logger. Any subsequent calls have no effect and do not override a previously configured global logger. This integration calls `logging.basicConfig()` for you when `OTEL_PYTHON_LOG_CORRELATION` is set to `true`. It uses the format and level specified by `OTEL_PYTHON_LOG_FORMAT` and `OTEL_PYTHON_LOG_LEVEL` environment variables respectively.

If you code or some other library/framework you are using calls logging.basicConfig before this integration is enabled, then this integration's logging format will not be used and log statements will not contain tracing context. For this reason, you'll need to make sure this integration is enabled as early as possible in the service lifecycle or your framework is configured to use a logging format with placeholders for tracing context. This can be achieved by adding the following placeholders to your logging format:

```
%(otelSpanID)s %(otelTraceID)s %(otelServiceName)s
```

API

```
from opentelemetry.instrumentation.logging import LoggingInstrumentor

LoggingInstrumentor().instrument(set_logging_format=True)
```

Note: If you do not set `OTEL_PYTHON_LOG_CORRELATION` to `true` but instead set the logging format manually or through your framework, you must ensure that this integration is enabled before you set the logging format. This is important because unless the integration is enabled, the tracing context variables are not injected into the log record objects. This means any attempted log statements made after setting the logging format and before enabling this integration will result in KeyError exceptions. Such exceptions are automatically swallowed by the logging module and do not result in crashes but you may still lose out on important log messages.

---

class opentelemetry.instrumentation.logging.**LoggingInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for stdlib logging module.

This instrumentor injects tracing context into logging records and optionally sets the global logging format to the following:

```
%(asctime)s %(levelname)s [%(name)s] [%(filename)s:%(lineno)d] [trace_id=
↪%(otelTraceID)s span_id=%(otelSpanID)s resource.service.name=%(otelServiceName)s]␣
↪- %(message)s

def log_hook(span: Span, record: LogRecord):
        if span and span.is_recording():
            record.custom_user_attribute_from_log_hook = "some-value"
```

Parameters

- **tracer_provider** – Tracer provider instance that can be used to fetch a tracer.
- **set_logging_format** – When set to True, it calls logging.basicConfig() and sets a logging format.
- **logging_format** – Accepts a string and sets it as the logging format when set_logging_format is set to True.
- **log_level** – Accepts one of the following values and sets the logging level to it. logging.INFO logging.DEBUG logging.WARN logging.ERROR logging.FATAL
- **log_hook** – execute custom logic when record is created

See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return [‘requests ~= 1.0’]

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.19 OpenTelemetry MySQL Instrumentation

MySQL instrumentation supporting mysql-connector, it can be enabled by using `MySQLInstrumentor`.

Usage

```
import mysql.connector
from opentelemetry.instrumentation.mysql import MySQLInstrumentor

MySQLInstrumentor().instrument()

cnx = mysql.connector.connect(database="MySQL_Database")
cursor = cnx.cursor()
cursor.execute("INSERT INTO test (testField) VALUES (123)")
```

(continues on next page)

(continued from previous page)

```
cursor.close()
cnx.close()
```

API

class opentelemetry.instrumentation.mysql.**MySQLInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

instrument_connection(*connection*, *tracer_provider=None*)

Enable instrumentation in a MySQL connection.

Parameters

- **connection** – The connection to instrument.
- **tracer_provider** – The optional tracer provider to use. If omitted the current globally configured one is used.

Returns
An instrumented connection.

uninstrument_connection(*connection*)

Disable instrumentation in a MySQL connection.

Parameters
connection – The connection to uninstrument.

Returns
An uninstrumented connection.

2.1.20 OpenTelemetry Psycopg Instrumentation

The integration with PostgreSQL supports the Psycopg library, it can be enabled by using `Psycopg2Instrumentor`.

SQLCOMMENTER

You can optionally configure Psycopg2 instrumentation to enable sqlcommenter which enriches the query with contextual information.

Usage

```
from opentelemetry.instrumentation.psycopg2 import Psycopg2Instrumentor

Psycopg2Instrumentor().instrument(enable_commenter=True, commenter_options={})
```

For example,

```
Invoking cursor.execute("select * from auth_users") will lead to sql query "select *␣
↪from auth_users" but when SQLCommenter is enabled
the query will get appended with some configurable tags like "select * from auth_users /
↪*tag=value*/;"
```

SQLCommenter Configurations

We can configure the tags to be appended to the sqlquery log by adding configuration inside commenter_options(default:{}) keyword

db_driver = True(Default) or False

For example, :: Enabling this flag will add psycopg2 and it's version which is */psycopg2%%3A2.9.3/*

dbapi_threadsafety = True(Default) or False

For example, :: Enabling this flag will add threadsafety */dbapi_threadsafety=2/*

dbapi_level = True(Default) or False

For example, :: Enabling this flag will add dbapi_level */dbapi_level='2.0'/*

libpq_version = True(Default) or False

For example, :: Enabling this flag will add libpq_version */libpq_version=140001/*

driver_paramstyle = True(Default) or False

For example, :: Enabling this flag will add driver_paramstyle */driver_paramstyle='pyformat'/*

opentelemetry_values = True(Default) or False

For example, :: Enabling this flag will add traceparent values */traceparent='00-03afa25236b8cd948fa853d67038ac79-405ff022e8247c46-01'/*

Usage

```
import psycopg2
from opentelemetry.instrumentation.psycopg2 import Psycopg2Instrumentor


Psycopg2Instrumentor().instrument()

cnx = psycopg2.connect(database='Database')
cursor = cnx.cursor()
cursor.execute("INSERT INTO test (testField) VALUES (123)")
cursor.close()
cnx.close()
```

API

class opentelemetry.instrumentation.psycopg2.**Psycopg2Instrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

static instrument_connection(*connection*, *tracer_provider=None*)

static uninstrument_connection(*connection*)

class opentelemetry.instrumentation.psycopg2.**DatabaseApiIntegration**(*name*, *database_system*, *connection_attributes=None*, *version=''*, *tracer_provider=None*, *capture_parameters=False*, *enable_commenter=False*, *commenter_options=None*, *connect_module=None*)

Bases: *DatabaseApiIntegration*

wrapped_connection(*connect_method*, *args*, *kwargs*)

Add object proxy to connection object.

class opentelemetry.instrumentation.psycopg2.**CursorTracer**(*db_api_integration*)

Bases: *CursorTracer*

get_operation_name(*cursor*, *args*)

get_statement(*cursor*, *args*)

2.1.21 OpenTelemetry pymemcache Instrumentation

Usage

The OpenTelemetry `pymemcache` integration traces pymemcache client operations

Usage

```
from opentelemetry.instrumentation.pymemcache import PymemcacheInstrumentor

PymemcacheInstrumentor().instrument()

from pymemcache.client.base import Client
client = Client(('localhost', 11211))
client.set('some_key', 'some_value')
```

API

class opentelemetry.instrumentation.pymemcache.**PymemcacheInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for pymemcache See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
`Collection[str]`

2.1.22 OpenTelemetry pymongo Instrumentation

The integration with MongoDB supports the pymongo library, it can be enabled using the `PymongoInstrumentor`.

Usage

```
from pymongo import MongoClient
from opentelemetry.instrumentation.pymongo import PymongoInstrumentor

PymongoInstrumentor().instrument()
client = MongoClient()
db = client["MongoDB_Database"]
collection = db["MongoDB_Collection"]
collection.find_one()
```

API

The *instrument* method accepts the following keyword args:

tracer_provider (TracerProvider) - an optional tracer provider request_hook (Callable) - a function with extra user-defined logic to be performed before querying mongodb this function signature is: def request_hook(span: Span, event: CommandStartedEvent) -> None response_hook (Callable) - a function with extra user-defined logic to be performed after the query returns with a successful response this function signature is: def response_hook(span: Span, event: CommandSucceededEvent) -> None failed_hook (Callable) - a function with extra user-defined logic to be performed after the query returns with a failed response this function signature is: def failed_hook(span: Span, event: CommandFailedEvent) -> None

for example:

opentelemetry.instrumentation.pymongo.**dummy_callback**(*span*, *event*)

class opentelemetry.instrumentation.pymongo.**CommandTracer**(*tracer*, *request_hook=<function dummy_callback>*, *response_hook=<function dummy_callback>*, *failed_hook=<function dummy_callback>*)

Bases: CommandListener

started(*event*)

Method to handle a pymongo CommandStartedEvent

succeeded(*event*)

Method to handle a pymongo CommandSucceededEvent

failed(*event*)

Method to handle a pymongo CommandFailedEvent

class opentelemetry.instrumentation.pymongo.**PymongoInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.23 OpenTelemetry PyMySQL Instrumentation

The integration with PyMySQL supports the PyMySQL library and can be enabled by using PyMySQLInstrumentor.

Usage

```
import pymysql
from opentelemetry.instrumentation.pymysql import PyMySQLInstrumentor


PyMySQLInstrumentor().instrument()

cnx = pymysql.connect(database="MySQL_Database")
cursor = cnx.cursor()
cursor.execute("INSERT INTO test (testField) VALUES (123)"
cnx.commit()
cursor.close()
cnx.close()
```

API

class opentelemetry.instrumentation.pymysql.**PyMySQLInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

static instrument_connection(*connection*, *tracer_provider=None*)

Enable instrumentation in a PyMySQL connection.

Parameters

- **connection** – The connection to instrument.
- **tracer_provider** – The optional tracer provider to use. If omitted the current globally configured one is used.

Returns
An instrumented connection.

static uninstrument_connection(*connection*)

Disable instrumentation in a PyMySQL connection.

Parameters
connection – The connection to uninstrument.

Returns
An uninstrumented connection.

2.1.24 OpenTelemetry Pyramid Instrumentation

Pyramid instrumentation supporting pyramid, it can be enabled by using `PyramidInstrumentor`.

Usage

There are two methods to instrument Pyramid:

Method 1 (Instrument all Configurators):

```
from pyramid.config import Configurator
from opentelemetry.instrumentation.pyramid import PyramidInstrumentor

PyramidInstrumentor().instrument()

config = Configurator()

# use your config as normal
config.add_route('index', '/')
```

Method 2 (Instrument one Configurator):

```
from pyramid.config import Configurator
from opentelemetry.instrumentation.pyramid import PyramidInstrumentor

config = Configurator()
PyramidInstrumentor().instrument_config(config)

# use your config as normal
config.add_route('index', '/')
```

Using `pyramid.tweens` setting:

If you use Method 2 and then set tweens for your application with the `pyramid.tweens` setting, you need to add `opentelemetry.instrumentation.pyramid.trace_tween_factory` explicity to the list, *as well as* instrumenting the config as shown above.

For example:

```
from pyramid.config import Configurator
from opentelemetry.instrumentation.pyramid import PyramidInstrumentor

settings = {
    'pyramid.tweens', 'opentelemetry.instrumentation.pyramid.trace_tween_factory\nyour_
↪tween_no_1\nyour_tween_no_2',
}
config = Configurator(settings=settings)
PyramidInstrumentor().instrument_config(config)

# use your config as normal.
config.add_route('index', '/')
```

Configuration

Exclude lists

To exclude certain URLs from being tracked, set the environment variable `OTEL_PYTHON_PYRAMID_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` as fallback) with comma delimited regexes representing which URLs to exclude.

For example,

```
export OTEL_PYTHON_PYRAMID_EXCLUDED_URLS="client/.*/info,healthcheck"
```

will exclude requests such as `https://site/client/123/info` and `https://site/xyz/healthcheck`.

Capture HTTP request and response headers

You can configure the agent to capture predefined HTTP headers as span attributes, according to the semantic convention.

Request headers

To capture predefined HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="content-type,custom_
↪request_header"
```

will extract `content-type` and `custom_request_header` from request headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Request header names in pyramid are case insensitive and - characters are replaced by _. So, giving header name as `CUStom_Header` in environment variable will be able capture header with name `custom-header`.

The name of the added span attribute will follow the format `http.request.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.request.header.custom_request_header = ["<value1>, <value2>"]`

Response headers

To capture predefined HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="content-type,custom_
↪response_header"
```

will extract `content-type` and `custom_response_header` from response headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Response header names captured in pyramid are case insensitive. So, giving header name as `CUStomHeader` in environment variable will be able capture header with name `customheader`.

The name of the added span attribute will follow the format `http.response.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.response.header.custom_response_header = ["<value1>, <value2>"]`

Note: Environment variable names to capture http headers are still experimental, and thus are subject to change.

API

class opentelemetry.instrumentation.pyramid.**PyramidInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return [‘requests ~= 1.0’]

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

static instrument_config(*config*)

Enable instrumentation in a Pyramid configurator.

Parameters
config – The Configurator to instrument.

static uninstrument_config(*config*)

2.1.25 OpenTelemetry Redis Instrumentation

Instrument redis to report Redis queries.

There are two options for instrumenting code. The first option is to use the `opentelemetry-instrumentation` executable which will automatically instrument your Redis client. The second is to programmatically enable instrumentation via the following code:

Usage

```
from opentelemetry.instrumentation.redis import RedisInstrumentor
import redis


# Instrument redis
RedisInstrumentor().instrument()

# This will report a span with the default settings
client = redis.StrictRedis(host="localhost", port=6379)
client.get("my-key")
```

Async Redis clients (i.e. redis.asyncio.Redis) are also instrumented in the same way:

```
from opentelemetry.instrumentation.redis import RedisInstrumentor
import redis.asyncio


# Instrument redis
RedisInstrumentor().instrument()

# This will report a span with the default settings
async def redis_get():
    client = redis.asyncio.Redis(host="localhost", port=6379)
    await client.get("my-key")
```

The *instrument* method accepts the following keyword args:

tracer_provider (TracerProvider) - an optional tracer provider

request_hook (Callable) - a function with extra user-defined logic to be performed before performing the request this function signature is: def request_hook(span: Span, instance: redis.connection.Connection, args, kwargs) -> None

response_hook (Callable) - a function with extra user-defined logic to be performed after performing the request this function signature is: def response_hook(span: Span, instance: redis.connection.Connection, response) -> None

for example:

API

class opentelemetry.instrumentation.redis.**RedisInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for Redis See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return [‘requests ~= 1.0’]

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.26 OpenTelemetry requests Instrumentation

This library allows tracing HTTP requests made by the requests library.

Usage

```
import requests
from opentelemetry.instrumentation.requests import RequestsInstrumentor

# You can optionally pass a custom TracerProvider to instrument().
RequestsInstrumentor().instrument()
response = requests.get(url="https://www.example.org/")
```

Configuration

Exclude lists

To exclude certain URLs from being tracked, set the environment variable OTEL_PYTHON_REQUESTS_EXCLUDED_URLS (or OTEL_PYTHON_EXCLUDED_URLS as fallback) with comma delimited regexes representing which URLs to exclude.

For example,

```
export OTEL_PYTHON_REQUESTS_EXCLUDED_URLS="client/.*/info,healthcheck"
```

will exclude requests such as https://site/client/123/info and https://site/xyz/healthcheck.

API

opentelemetry.instrumentation.requests.**get_default_span_name**(*method*)

Default implementation for name_callback, returns HTTP {method_name}.

class opentelemetry.instrumentation.requests.**RequestsInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for requests See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

static uninstrument_session(*session*)

Disables instrumentation on the session object.

2.1.27 OpenTelemetry SQLAlchemy Instrumentation

Instrument sqlalchemy to report SQL queries.

There are two options for instrumenting code. The first option is to use the `opentelemetry-instrument` executable which will automatically instrument your SQLAlchemy engine. The second is to programmatically enable instrumentation via the following code:

SQLCOMMENTER

You can optionally configure SQLAlchemy instrumentation to enable sqlcommenter which enriches the query with contextual information.

Usage

```
from opentelemetry.instrumentation.sqlalchemy import SQLAlchemyInstrumentor

SQLAlchemyInstrumentor().instrument(enable_commenter=True, commenter_options={})
```

For example,

```
Invoking engine.execute("select * from auth_users") will lead to sql query "select *␣
↪from auth_users" but when SQLCommenter is enabled
the query will get appended with some configurable tags like "select * from auth_users /
↪*tag=value*/;"
```

SQLCommenter Configurations

We can configure the tags to be appended to the sqlquery log by adding configuration inside commenter_options(default:{}) keyword

db_driver = True(Default) or False

For example, :: Enabling this flag will add any underlying driver like psycopg2 */db_driver='psycopg2'/*

db_framework = True(Default) or False

For example, :: Enabling this flag will add db_framework and it's version */db_framework='sqlalchemy:0.41b0'/*

opentelemetry_values = True(Default) or False

For example, :: Enabling this flag will add traceparent values */traceparent='00-03afa25236b8cd948fa853d67038ac79-405ff022e8247c46-01'/*

Usage

```
from sqlalchemy import create_engine

from opentelemetry.instrumentation.sqlalchemy import SQLAlchemyInstrumentor
import sqlalchemy

engine = create_engine("sqlite:///:memory:")
SQLAlchemyInstrumentor().instrument(
    engine=engine,
)

# of the async variant of SQLAlchemy

from sqlalchemy.ext.asyncio import create_async_engine

from opentelemetry.instrumentation.sqlalchemy import SQLAlchemyInstrumentor
import sqlalchemy

engine = create_async_engine("sqlite:///:memory:")
SQLAlchemyInstrumentor().instrument(
    engine=engine.sync_engine
)
```

API

class opentelemetry.instrumentation.sqlalchemy.**SQLAlchemyInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for SQLAlchemy See *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.28 OpenTelemetry SQLite3 Instrumentation

SQLite instrumentation supporting sqlite3, it can be enabled by using `SQLite3Instrumentor`.

Usage

```
import sqlite3
from opentelemetry.instrumentation.sqlite3 import SQLite3Instrumentor


SQLite3Instrumentor().instrument()

cnx = sqlite3.connect('example.db')
cursor = cnx.cursor()
cursor.execute("INSERT INTO test (testField) VALUES (123)")
cursor.close()
cnx.close()
```

API

class opentelemetry.instrumentation.sqlite3.**SQLite3Instrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

static instrument_connection(*connection*, *tracer_provider=None*)

Enable instrumentation in a SQLite connection.

Parameters

- **connection** – The connection to instrument.

- **tracer_provider** – The optional tracer provider to use. If omitted the current globally configured one is used.

Returns
An instrumented connection.

static uninstrument_connection(*connection*)

Disable instrumentation in a SQLite connection.

Parameters
connection – The connection to uninstrument.

Returns
An uninstrumented connection.

2.1.29 OpenTelemetry Starlette Instrumentation

This library provides automatic and manual instrumentation of Starlette web frameworks, instrumenting http requests served by applications utilizing the framework.

auto-instrumentation using the opentelemetry-instrumentation package is also supported.

Installation

```
pip install opentelemetry-instrumentation-starlette
```

References

- OpenTelemetry Project
- OpenTelemetry Python Examples

API

Usage

```
from opentelemetry.instrumentation.starlette import StarletteInstrumentor
from starlette import applications
from starlette.responses import PlainTextResponse
from starlette.routing import Route

def home(request):
    return PlainTextResponse("hi")

app = applications.Starlette(
    routes=[Route("/foobar", home)]
)
StarletteInstrumentor.instrument_app(app)
```

Configuration

Exclude lists

To exclude certain URLs from being tracked, set the environment variable `OTEL_PYTHON_STARLETTE_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` as fallback) with comma delimited regexes representing which URLs to exclude.

For example,

```
export OTEL_PYTHON_STARLETTE_EXCLUDED_URLS="client/.*/info,healthcheck"
```

will exclude requests such as `https://site/client/123/info` and `https://site/xyz/healthcheck`.

Request/Response hooks

Utilize request/response hooks to execute custom logic to be performed before/after performing a request. The server request hook takes in a server span and ASGI scope object for every incoming request. The client request hook is called with the internal span and an ASGI scope which is sent as a dictionary for when the method recieve is called. The client response hook is called with the internal span and an ASGI event which is sent as a dictionary for when the method send is called.

```
def server_request_hook(span: Span, scope: dict):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_request_hook", "some-value")
def client_request_hook(span: Span, scope: dict):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_client_request_hook", "some-value
↪")
def client_response_hook(span: Span, message: dict):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_response_hook", "some-value")

StarletteInstrumentor().instrument(server_request_hook=server_request_hook, client_
↪request_hook=client_request_hook, client_response_hook=client_response_hook)
```

Capture HTTP request and response headers

You can configure the agent to capture predefined HTTP headers as span attributes, according to the semantic convention.

Request headers

To capture predefined HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="content-type,custom_
↪request_header"
```

will extract `content-type` and `custom_request_header` from request headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Request header names in starlette are case insensitive. So, giving header name as `CUStom-Header` in environment variable will be able capture header with name `custom-header`.

The name of the added span attribute will follow the format `http.request.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.request.header.custom_request_header = ["<value1>, <value2>"]`

Response headers

To capture predefined HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="content-type,custom_
↪response_header"
```

will extract `content-type` and `custom_response_header` from response headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Response header names captured in starlette are case insensitive. So, giving header name as `CUStomHeader` in environment variable will be able capture header with name `customheader`.

The name of the added span attribute will follow the format `http.response.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.response.header.custom_response_header = ["<value1>, <value2>"]`

Note: Environment variable names to capture http headers are still experimental, and thus are subject to change.

API

class opentelemetry.instrumentation.starlette.**StarletteInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

An instrumentor for starlette

See *BaseInstrumentor*

static instrument_app(*app*, *server_request_hook=None*, *client_request_hook=None*, *client_response_hook=None*, *tracer_provider=None*)

Instrument an uninstrumented Starlette application.

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.30 OpenTelemetry Tornado Instrumentation

This library uses OpenTelemetry to track web requests in Tornado applications.

Usage

```
import tornado.web
from opentelemetry.instrumentation.tornado import TornadoInstrumentor

# apply tornado instrumentation
TornadoInstrumentor().instrument()

class Handler(tornado.web.RequestHandler):
    def get(self):
        self.set_status(200)

app = tornado.web.Application([(r"/", Handler)])
app.listen(8080)
tornado.ioloop.IOLoop.current().start()
```

Configuration

The following environment variables are supported as configuration options:

- OTEL_PYTHON_TORNADO_EXCLUDED_URLS

A comma separated list of paths that should not be automatically traced. For example, if this is set to

```
export OTEL_PYTHON_TORNADO_EXCLUDED_URLS='/healthz,/ping'
```

Then any requests made to `/healthz` and `/ping` will not be automatically traced.

Request attributes

To extract certain attributes from Tornado's request object and use them as span attributes, set the environment variable `OTEL_PYTHON_TORNADO_TRACED_REQUEST_ATTRS` to a comma delimited list of request attribute names.

For example,

```
export OTEL_PYTHON_TORNADO_TRACED_REQUEST_ATTRS='uri,query'
```

will extract path_info and content_type attributes from every traced request and add them as span attributes.

Request/Response hooks

Tornado instrumentation supports extending tracing behaviour with the help of hooks. Its `instrument()` method accepts three optional functions that get called back with the created span and some other contextual information. Example:

```
# will be called for each incoming request to Tornado
# web server. `handler` is an instance of
# `tornado.web.RequestHandler`.
def server_request_hook(span, handler):
    pass

# will be called just before sending out a request with
# `tornado.httpclient.AsyncHTTPClient.fetch`.
# `request` is an instance of ``tornado.httpclient.HTTPRequest`.
def client_request_hook(span, request):
    pass

# will be called after a outgoing request made with
# `tornado.httpclient.AsyncHTTPClient.fetch` finishes.
# `response`` is an instance of ``Future[tornado.httpclient.HTTPResponse]`.
def client_resposne_hook(span, future):
    pass

# apply tornado instrumentation with hooks
TornadoInstrumentor().instrument(
    server_request_hook=server_request_hook,
    client_request_hook=client_request_hook,
    client_response_hook=client_resposne_hook
)
```

Capture HTTP request and response headers

You can configure the agent to capture predefined HTTP headers as span attributes, according to the semantic convention.

Request headers

To capture predefined HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="content-type,custom_
↪request_header"
```

will extract `content-type` and `custom_request_header` from request headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Request header names in tornado are case insensitive. So, giving header name as `CUStomHeader` in environment variable will be able capture header with name `customheader`.

The name of the added span attribute will follow the format `http.request.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.request.header.custom_request_header = ["<value1>, <value2>"]`

Response headers

To capture predefined HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="content-type,custom_
↪response_header"
```

will extract `content-type` and `custom_response_header` from response headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Response header names captured in tornado are case insensitive. So, giving header name as `CUStomHeader` in environment variable will be able capture header with name `customheader`.

The name of the added span attribute will follow the format `http.response.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.response.header.custom_response_header = ["<value1>, <value2>"]`

Note: Environment variable names to capture http headers are still experimental, and thus are subject to change.

API

class opentelemetry.instrumentation.tornado.**TornadoInstrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

patched_handlers = []

original_handler_new = None

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

opentelemetry.instrumentation.tornado.**patch_handler_class**(*tracer*, *cls*, *request_hook=None*)

opentelemetry.instrumentation.tornado.**unpatch_handler_class**(*cls*)

2.1.31 OpenTelemetry urllib3 Instrumentation

This library allows tracing HTTP requests made by the urllib3 library.

Usage

```
import urllib3
from opentelemetry.instrumentation.urllib3 import URLLib3Instrumentor

def strip_query_params(url: str) -> str:
    return url.split("?")[0]

URLLib3Instrumentor().instrument(
    # Remove all query params from the URL attribute on the span.
    url_filter=strip_query_params,
)

http = urllib3.PoolManager()
response = http.request("GET", "https://www.example.org/")
```

Configuration

Request/Response hooks

The urllib3 instrumentation supports extending tracing behavior with the help of request and response hooks. These are functions that are called back by the instrumentation right after a Span is created for a request and right before the span is finished processing a response respectively. The hooks can be configured as follows:

```
# `request` is an instance of urllib3.connectionpool.HTTPConnectionPool
def request_hook(span, request):
    pass

# `request` is an instance of urllib3.connectionpool.HTTPConnectionPool
# `response` is an instance of urllib3.response.HTTPResponse
def response_hook(span, request, response):
    pass

URLLib3Instrumentor.instrument(
    request_hook=request_hook, response_hook=response_hook)
)
```

API

class opentelemetry.instrumentation.urllib3.**URLLib3Instrumentor**(**args*, ***kwargs*)

Bases: *BaseInstrumentor*

instrumentation_dependencies()

Return a list of python packages with versions that the will be instrumented.

The format should be the same as used in requirements.txt or setup.py.

For example, if an instrumentation instruments requests 1.x, this method should look like:

def instrumentation_dependencies(self) -> Collection[str]:
return ['requests ~= 1.0']

This will ensure that the instrumentation will only be used when the specified library is present in the environment.

Return type
Collection[str]

2.1.32 OpenTelemetry WSGI Instrumentation

This library provides a WSGI middleware that can be used on any WSGI framework (such as Django / Flask / Web.py) to track requests timing through OpenTelemetry.

Usage (Flask)

```
from flask import Flask
from opentelemetry.instrumentation.wsgi import OpenTelemetryMiddleware

app = Flask(__name__)
app.wsgi_app = OpenTelemetryMiddleware(app.wsgi_app)

@app.route("/")
def hello():
    return "Hello!"

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

Usage (Django)

Modify the application's `wsgi.py` file as shown below.

```
import os
from opentelemetry.instrumentation.wsgi import OpenTelemetryMiddleware
from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'application.settings')

application = get_wsgi_application()
application = OpenTelemetryMiddleware(application)
```

Usage (Web.py)

```
import web
from opentelemetry.instrumentation.wsgi import OpenTelemetryMiddleware
from cheroot import wsgi

urls = ('/', 'index')


class index:

    def GET(self):
        return "Hello, world!"


if __name__ == "__main__":
    app = web.application(urls, globals())
    func = app.wsgifunc()

    func = OpenTelemetryMiddleware(func)

    server = wsgi.WSGIServer(
```

(continues on next page)

(continued from previous page)

```
        ("localhost", 5100), func, server_name="localhost"
    )
    server.start()
```

Configuration

Request/Response hooks

Utilize request/reponse hooks to execute custom logic to be performed before/after performing a request. Environ is an instance of WSGIEnvironment. Response_headers is a list of key-value (tuples) representing the response headers returned from the response.

```
def request_hook(span: Span, environ: WSGIEnvironment):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_request_hook", "some-value")

def response_hook(span: Span, environ: WSGIEnvironment, status: str, response_headers:␣
↪List):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_response_hook", "some-value")

OpenTelemetryMiddleware(request_hook=request_hook, response_hook=response_hook)
```

Capture HTTP request and response headers

You can configure the agent to capture predefined HTTP headers as span attributes, according to the semantic convention.

Request headers

To capture predefined HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="content-type,custom_
↪request_header"
```

will extract `content-type` and `custom_request_header` from request headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Request header names in wsgi are case insensitive and - characters are replaced by _. So, giving header name as `CUStom_Header` in environment variable will be able capture header with name `custom-header`.

The name of the added span attribute will follow the format `http.request.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.request.header.custom_request_header = ["<value1>, <value2>"]`

Response headers

To capture predefined HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma-separated list of HTTP header names.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="content-type,custom_
↪response_header"
```

will extract `content-type` and `custom_response_header` from response headers and add them as span attributes.

It is recommended that you should give the correct names of the headers to be captured in the environment variable. Response header names captured in wsgi are case insensitive. So, giving header name as `CUStomHeader` in environment variable will be able capture header with name `customheader`.

The name of the added span attribute will follow the format `http.response.header.<header_name>` where `<header_name>` being the normalized HTTP header name (lowercase, with - characters replaced by _). The value of the attribute will be single item list containing all the header values.

Example of the added span attribute, `http.response.header.custom_response_header = ["<value1>, <value2>"]`

Note: Environment variable names to capture http headers are still experimental, and thus are subject to change.

API

class opentelemetry.instrumentation.wsgi.**WSGIGetter**

Bases: `Getter[dict]`

get(*carrier*, *key*)

Getter implementation to retrieve a HTTP header value from the PEP3333-conforming WSGI environ

Parameters

- **carrier** (`dict`) – WSGI environ object
- **key** (`str`) – header name in environ object

Returns:
A list with a single string with the header value if it exists, else None.

Return type
`Optional[List[str]]`

keys(*carrier*)

Function that can retrieve all the keys in a carrier object.

Parameters
carrier – An object which contains values that are used to construct a Context.

Returns
list of keys from the carrier.

opentelemetry.instrumentation.wsgi.**setifnotnone**(*dic*, *key*, *value*)

opentelemetry.instrumentation.wsgi.**collect_request_attributes**(*environ*)

Collects HTTP request attributes from the PEP3333-conforming WSGI environ and returns a dictionary to be used as span creation attributes.

opentelemetry.instrumentation.wsgi.**collect_custom_request_headers_attributes**(*environ*)

Returns custom HTTP request headers which are configured by the user from the PEP3333-conforming WSGI environ to be used as span creation attributes as described in the specification https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/semantic_conventions/http.md#http-request-and-response-headers

opentelemetry.instrumentation.wsgi.**collect_custom_response_headers_attributes**(*response_headers*)

Returns custom HTTP response headers which are configured by the user from the PEP3333-conforming WSGI environ as described in the specification https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/semantic_conventions/http.md#http-request-and-response-headers

opentelemetry.instrumentation.wsgi.**add_response_attributes**(*span*, *start_response_status*, *response_headers*)

Adds HTTP response attributes to span using the arguments passed to a PEP3333-conforming start_response callable.

opentelemetry.instrumentation.wsgi.**get_default_span_name**(*environ*)

Default implementation for name_callback, returns HTTP {METHOD_NAME}.

class opentelemetry.instrumentation.wsgi.**OpenTelemetryMiddleware**(*wsgi*, *request_hook=None*, *response_hook=None*, *tracer_provider=None*, *meter_provider=None*)

Bases: `object`

The WSGI application middleware.

This class is a PEP 3333 conforming WSGI middleware that starts and annotates spans for any requests it is invoked with.

Parameters

- **wsgi** – The WSGI application callable to forward requests to.
- **request_hook** – Optional callback which is called with the server span and WSGI environ object for every incoming request.
- **response_hook** – Optional callback which is called with the server span, WSGI environ, status_code and response_headers for every incoming request.
- **tracer_provider** – Optional tracer provider to use. If omitted the current globally configured one is used.

class opentelemetry.instrumentation.wsgi.**ResponsePropagationSetter**

Bases: `object`

set(*carrier*, *key*, *value*)

2.1.33 OpenTelemetry Python - AWS X-Ray Propagator

class opentelemetry.propagators.aws.**AwsXRayPropagator**

Bases: TextMapPropagator

Propagator for the AWS X-Ray Trace Header propagation protocol.

See: https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html#xray-concepts-tracingheader

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

Create a Context from values in the carrier.

The extract function should retrieve values from the carrier object using getter, and use values to populate a Context value and return it.

Parameters

- **getter** (Getter[TypeVar(CarrierT)]) – a function that can retrieve zero or more values from the carrier. In the case that the value does not exist, return an empty list.
- **carrier** (TypeVar(CarrierT)) – and object which contains values that are used to construct a Context. This object must be paired with an appropriate getter which understands how to extract a value from it.
- **context** (Optional[Context]) – an optional Context to use. Defaults to root context if not set.

Return type
Context

Returns
A Context with configuration found in the carrier.

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

Inject values from a Context into a carrier.

inject enables the propagation of values into HTTP clients or other objects which perform an HTTP request. Implementations should use the Setter 's set method to set values on the carrier.

Parameters

- **carrier** (TypeVar(CarrierT)) – An object that a place to define HTTP headers. Should be paired with setter, which should know how to set header values on the carrier.
- **context** (Optional[Context]) – an optional Context to use. Defaults to current context if not set.
- **setter** (Setter[TypeVar(CarrierT)]) – An optional Setter object that can set values on the carrier.

Return type
None

property fields

Returns a set with the fields set in *inject*.

2.1.34 Performance Tests - Benchmarks

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

2.1.35 OpenTelemetry Python - AWS SDK Extension

Installation

```
pip install opentelemetry-sdk-extension-aws
```

AWS X-Ray IDs Generator

The **AWS X-Ray IDs Generator** provides a custom IDs Generator to make traces generated using the OpenTelemetry SDKs `TracerProvider` compatible with the AWS X-Ray backend service trace ID format.

Usage

Configure the OTel SDK TracerProvider with the provided custom IDs Generator to make spans compatible with the AWS X-Ray backend tracing service.

Install the OpenTelemetry SDK package.

```
pip install opentelemetry-sdk
```

Next, use the provided *AwsXRayIdGenerator* to initialize the `TracerProvider`.

```
import opentelemetry.trace as trace
from opentelemetry.sdk.extension.aws.trace import AwsXRayIdGenerator
from opentelemetry.sdk.trace import TracerProvider

trace.set_tracer_provider(
    TracerProvider(id_generator=AwsXRayIdGenerator())
)
```

API

class opentelemetry.sdk.extension.aws.trace.aws_xray_id_generator.**AwsXRayIdGenerator**

Bases: `IdGenerator`

Generates tracing IDs compatible with the AWS X-Ray tracing service. In the X-Ray system, the first 32 bits of the `TraceId` are the Unix epoch time in seconds. Since spans (AWS calls them segments) with an embedded timestamp more than 30 days ago are rejected, a purely random `TraceId` risks being rejected by the service.

See: https://docs.aws.amazon.com/xray/latest/devguide/xray-api-sendingdata.html#xray-api-traceids

random_id_generator = <opentelemetry.sdk.trace.id_generator.RandomIdGenerator object>

`generate_span_id()`

Get a new span ID.

Return type
`int`

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

`static 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

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

J

K

L

M

O

W