

---

# **OpenTelemetry Python Contrib**

**OpenTelemetry Authors**

**Aug 11, 2024**



# OPENTELEMETRY INSTRUMENTATIONS

<b>1</b>	<b>Installation</b>	<b>3</b>
<b>2</b>	<b>Extensions</b>	<b>5</b>
2.1	Installing Cutting Edge Packages . . . . .	5
<b>3</b>	<b>Indices and tables</b>	<b>101</b>
	<b>Python Module Index</b>	<b>103</b>
	<b>Index</b>	<b>105</b>



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.



## 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-{sdk-extension}
```

A complete list of packages can be found at the [Contrib repo instrumentation](#) and [Contrib repo exporter](#) directories.





## EXTENSIONS

Visit [OpenTelemetry Registry](#) to find a lot of related projects like exporters, instrumentation libraries, tracer implementations, resource, 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
pip install -e ./resource/opentelemetry-resource-detector-container
```

#### 2.1.1 OpenTelemetry aio pika Instrumentation

#### 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,
```

(continues on next page)

(continued from previous page)

```

]]) as session:
    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/response 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,
                                                                    sem_conv_opt_in_mode=_HTTPStabilityMode.DEPRECATED)

```

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 pyproject.toml.

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 pyproject.toml.

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 request 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

This instrumentation supports request and response hooks. These are functions that get called right after a span is created for a request and right before the span is finished for the response.

- The server request hook is passed 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 when the method `receive` is called.
- The client response hook is called with the internal span and an ASGI event when the method `send` is called.

For example,

```
def server_request_hook(span: Span, scope: dict[str, Any]):
    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[str, Any], message: dict[str, Any]):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_client_request_hook", "some-value")

def client_response_hook(span: Span, scope: dict[str, Any], message: dict[str, Any]):
    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 specified HTTP headers as span attributes, according to the [semantic convention](#).

### Request headers

To capture HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma delimited 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 the request headers and add them as span attributes.

Request header names in ASGI are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="Accept.*,X-.*"
```

Would match all request headers that start with `Accept` and `X-`.

To capture all request headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to `".*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST=".*"
```

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

For example: `http.request.header.custom_request_header = ["<value1>", "<value2>"]`

### Response headers

To capture HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma delimited 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 the response headers and add them as span attributes.

Response header names in ASGI are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="Content.*,X-.*"
```

Would match all response headers that start with Content and X-.

To capture all response headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to `"*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="*"
```

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

For example: `http.response.header.custom_response_header = ["<value1>", "<value2>"]`

### Sanitizing headers

In order to prevent storing sensitive data such as personally identifiable information (PII), session keys, passwords, etc, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS` to a comma delimited list of HTTP header names to be sanitized. Regexes may be used, and all header names will be matched in a case-insensitive manner.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS=".*session.*,set-cookie"
```

will replace the value of headers such as `session-id` and `set-cookie` with `[REDACTED]` in the span.

---

**Note:** The environment variable names used 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,*  
*sem\_conv\_opt\_in\_mode=\_HTTPStabilityMode.DEFA*

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

opentelemetry.instrumentation.asgi.**collect\_custom\_headers\_attributes**(*scope\_or\_response\_message,*  
*sanitize, header\_regexes,*  
*normalize\_names*)

Returns custom HTTP request or response headers to be added into SERVER span as span attributes.

**Refer specifications:**

- [https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/semantic\\_conventions/http.md#http-request-and-response-headers](https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/semantic_conventions/http.md#http-request-and-response-headers)

**Return type**

dict[str, list[str]]

opentelemetry.instrumentation.asgi.**get\_host\_port\_url\_tuple**(*scope*)

Returns (host, port, full\_url) tuple.

opentelemetry.instrumentation.asgi.**set\_status\_code**(*span, status\_code, metric\_attributes=None,*  
*sem\_conv\_opt\_in\_mode=\_HTTPStabilityMode.DEFAULT*)

Adds HTTP response attributes to span using the status\_code argument.

opentelemetry.instrumentation.asgi.**get\_default\_span\_details**(*scope*)

Default span name is the HTTP method and URL path, or just the method. <https://github.com/open-telemetry/opentelemetry-specification/pull/3165> [https://opentelemetry.io/docs/reference/specification/trace/semantic\\_conventions/http/#name](https://opentelemetry.io/docs/reference/specification/trace/semantic_conventions/http/#name)

**Parameters**

**scope** (dict) – 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,
                                                                tracer=None, meter=None,
                                                                http_capture_headers_server_request=None,
                                                                http_capture_headers_server_response=None,
                                                                http_capture_headers_sanitize_fields=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[str, Any]], 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[str, Any], Dict[str, Any]], None]]) – Optional callback which is called with the internal span, and ASGI scope and event which are sent as dictionaries for when the method receive is called.
- **client\_response\_hook** (Optional[Callable[[Span, Dict[str, Any], Dict[str, Any]], None]]) – Optional callback which is called with the internal span, and ASGI scope and event which are sent as dictionaries for when the method send is called.
- **tracer\_provider** – The optional tracer provider to use. If omitted the current globally configured one is used.
- **meter\_provider** – The optional meter provider to use. If omitted the current globally configured one is used.

## 2.1.5 OpenTelemetry asyncio Instrumentation

The `opentelemetry-instrumentation-asyncio` package allows tracing asyncio applications. The metric for coroutine, future, is generated even if there is no setting to generate a span.

## Run instrumented application

### 1. coroutine

```
# export OTEL_PYTHON_ASYNCIO_COROUTINE_NAMES_TO_TRACE=sleep

import asyncio
from opentelemetry.instrumentation.asyncio import AsyncioInstrumentor

AsyncioInstrumentor().instrument()

async def main():
    await asyncio.create_task(asyncio.sleep(0.1))

asyncio.run(main())
```

### 2. future

```
# export OTEL_PYTHON_ASYNCIO_FUTURE_TRACE_ENABLED=true

loop = asyncio.get_event_loop()

future = asyncio.Future()
future.set_result(1)
task = asyncio.ensure_future(future)
loop.run_until_complete(task)
```

### 3. to\_thread

```
import asyncio
from opentelemetry.instrumentation.asyncio import AsyncioInstrumentor

AsyncioInstrumentor().instrument()

async def main():
    await asyncio.to_thread(func)

def func():
    pass

asyncio.run(main())
```

## asyncio metric types

- `asyncio.process.duration` (seconds) - Duration of asyncio process
- `asyncio.process.count` (count) - Number of asyncio process

## API

**class** `opentelemetry.instrumentation.asyncio.AsyncioInstrumentor(*args, **kwargs)`

Bases: [\*BaseInstrumentor\*](#)

An instrumentor for asyncio

See [\*BaseInstrumentor\*](#)

**methods\_with\_coroutine** = ['create\_task', 'ensure\_future', 'wait\_for', 'wait', 'as\_completed', 'run\_coroutine\_threadsafe']

**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 pyproject.toml.

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\_method\_with\_coroutine**(*method\_name*)

Instruments specified asyncio method.

**instrument\_gather()**

**instrument\_to\_thread()**

**Return type**

None

**instrument\_taskgroup\_create\_task()**

**Return type**

None

**trace\_to\_thread**(*func*)

Trace a function.

**trace\_item**(*coro\_or\_future*)

Trace a coroutine or future item.

**async trace\_coroutine**(*coro*)

**trace\_future**(*future*)

**record\_process**(*start, attr, span=None, exception=None*)

Record the processing time, update histogram and counter, and handle span.

**Parameters**

- **start** (float) – Start time of the process.
- **attr** (dict) – Attributes for the histogram and counter.
- **span** – Optional span for tracing.
- **exception** – Optional exception occurred during the process.

**Return type**

None

`opentelemetry.instrumentation.asyncio.determine_state(exception)`

**Return type**

str

`opentelemetry.instrumentation.asyncio.uninstrument_taskgroup_create_task()`

**Return type**

None

`opentelemetry.instrumentation.asyncio.uninstrument_to_thread()`

**Return type**

None

`opentelemetry.instrumentation.asyncio.uninstrument_gather()`

**Return type**

None

`opentelemetry.instrumentation.asyncio.uninstrument_method_with_coroutine(method_name)`

Uninstrument specified asyncio method.

**Return type**

None

## 2.1.6 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 pyproject.toml.

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]

### 2.1.7 OpenTelemetry AWS lambda Instrumentation

The opentelemetry-instrumentation-aws-lambda package provides an Instrumentor to traces calls within a Python AWS Lambda function.

#### Usage

```
# Copy this snippet into an AWS Lambda function  
  
import boto3  
from opentelemetry.instrumentation.botocore import BotocoreInstrumentor  
from opentelemetry.instrumentation.aws_lambda import AwsLambdaInstrumentor  
  
# Enable instrumentation  
BotocoreInstrumentor().instrument()  
AwsLambdaInstrumentor().instrument()  
  
# Lambda function  
def lambda_handler(event, context):  
    s3 = boto3.resource('s3')  
    for bucket in s3.buckets.all():  
        print(bucket.name)  
  
    return "200 OK"
```

## API

The `instrument` method accepts the following keyword args:

`tracer_provider` (`TracerProvider`) - an optional tracer provider `meter_provider` (`MeterProvider`) - an optional meter provider `event_context_extractor` (`Callable`) - a function that returns an OTEL Trace Context given the Lambda Event the AWS Lambda was invoked with this function signature is: `def event_context_extractor(lambda_event: Any) -> Context` for example:

```
from opentelemetry.instrumentation.aws_lambda import AwsLambdaInstrumentor

def custom_event_context_extractor(lambda_event):
    # If the `TraceContextTextMapPropagator` is the global propagator, we
    # can use it to parse out the context from the HTTP Headers.
    return get_global_textmap().extract(lambda_event["foo"]["headers"])

AwsLambdaInstrumentor().instrument(
    event_context_extractor=custom_event_context_extractor
)
```

---

```
class opentelemetry.instrumentation.aws_lambda.AwsLambdaInstrumentor(*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 pyproject.toml.

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]

## 2.1.8 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 pyproject.toml.

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`.

## 2.1.9 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 pyproject.toml.

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)

### 2.1.10 OpenTelemetry Boto3 SQS Instrumentation

This library allows tracing requests made by the Boto3 library to the SQS service.

#### Installation

```
pip install opentelemetry-instrumentation-boto3sqs
```

#### References

- [OpenTelemetry boto3sqs/ Tracing](#)
- [OpenTelemetry Project](#)

Instrument `boto3sqs` to trace SQS applications.

## Usage

```
import boto3
from opentelemetry.instrumentation.boto3sqs import Boto3SQSInstrumentor

Boto3SQSInstrumentor().instrument()
```

---

```
class opentelemetry.instrumentation.boto3sqs.Boto3SQSGetter
```

```
    Bases: Getter[CarrierT]
```

```
    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** (TypeVar(CarrierT)) – An object which contains values that are used to construct a Context.
- **key** (str) – key of a field in carrier.

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

**Return type**

Optional[List[str]]

```
    keys(carrier)
```

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

**Parameters**

**carrier** (TypeVar(CarrierT)) – 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.boto3sqs.Boto3SQSSetter
```

```
    Bases: Setter[CarrierT]
```

```
    set(carrier, key, value)
```

Function that can set a value into a carrier"""

**Parameters**

- **carrier** (TypeVar(CarrierT)) – An object which contains values that are used to construct a Context.
- **key** (str) – key of a field in carrier.
- **value** (str) – value for a field in carrier.

**Return type**

None

```
class opentelemetry.instrumentation.boto3sqs.Boto3SQSInstrumentor(*args, **kwargs)
```

Bases: *BaseInstrumentor*

```
received_messages_spans: Dict[str, Span] = {}
```

```
current_span_related_to_token: Span = None
```

```
current_context_token = None
```

```
class ContextableList(iterable=(), /)
```

Bases: *list*

Since the classic way to process SQS messages is using a `for` loop, without a well defined scope like a callback - we are doing something similar to the instrumentation of `Kafka-python` and instrumenting the `__iter__` functions and the `__getitem__` functions to set the span context of the addressed message. Since the return value from an `SQS.ReceiveMessage` returns a builtin list, we cannot wrap it and change all of the calls for `list.__iter__` and `list.__getitem__` - therefore we use `ContextableList`. It is bound to the `received_messages_spans` dict

```
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 `pyproject.toml`.

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]`

## 2.1.11 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"
```

(continues on next page)

(continued from previous page)

```
)  
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 response_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 pyproject.toml.

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 Cassandra Instrumentation

### 2.1.13 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")

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

add.delay(42, 50)

```

## Setting up tracing

When tracing a celery worker process, tracing and instrumentation 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`

**metrics = None**

**task\_id\_to\_start\_time = {}**

**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 pyproject.toml.

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]

**update\_task\_duration\_time(task\_id)**

**create\_celery\_metrics(meter)**

**Return type**

None

### 2.1.14 OpenTelemetry confluent-kafka Instrumentation

This library allows tracing requests made by the confluent-kafka library.

#### Installation

```
pip install opentelemetry-instrumentation-confluent-kafka
```

#### References

- [OpenTelemetry confluent-kafka/ Tracing](#)
- [OpenTelemetry Project](#)

### 2.1.15 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.
- **enable\_commenter** (bool) – Flag to enable/disable sqlcommenter.
- **db\_api\_integration\_factory** – The `DatabaseApiIntegration` to use. If none is passed the default one is used.

`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`.
- **db\_api\_integration\_factory** – The `DatabaseApiIntegration` to use. If none is passed the default one is used.
- **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)
```

## 2.1.16 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 tracking, set the environment variable `OTEL_PYTHON_DJANGO_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` to cover all instrumentations) to a string of comma delimited regexes that match the URLs.

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 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 the `path_info` and `content_type` attributes from every traced request and add them as span attributes.

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

### Request and Response hooks

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

```
def request_hook(span, request):  
    pass  
  
def response_hook(span, request, response):  
    pass  
  
DjangoInstrumentor().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 specified HTTP headers as span attributes, according to the [semantic convention](#).

## Request headers

To capture HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma delimited 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 the request headers and add them as span attributes.

Request header names in Django are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="Accept.*,X-.*"
```

Would match all request headers that start with `Accept` and `X-`.

To capture all request headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to `".*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST=".*"
```

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

For example: `http.request.header.custom_request_header = [<value1>,<value2>"]`

## Response headers

To capture HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma delimited 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 the response headers and add them as span attributes.

Response header names in Django are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="Content.*,X-.*"
```

Would match all response headers that start with `Content` and `X-`.

To capture all response headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to `".*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE=".*"
```

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

For example: `http.response.header.custom_response_header = [<value1>,<value2>"]`

## Sanitizing headers

In order to prevent storing sensitive data such as personally identifiable information (PII), session keys, passwords, etc, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS` to a comma delimited list of HTTP header names to be sanitized. Regexes may be used, and all header names will be matched in a case-insensitive manner.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS=".*session.*,set-cookie"
```

will replace the value of headers such as `session-id` and `set-cookie` with `[REDACTED]` in the span.

**Note:** The environment variable names used to capture HTTP headers are still experimental, and thus are subject to change.

## 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 pyproject.toml.

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]

## 2.1.17 OpenTelemetry elasticsearch Integration

This library allows tracing elasticsearch made by the [elasticsearch](#) library.

### Installation

```
pip install opentelemetry-instrumentation-elasticsearch
```

### References

- [OpenTelemetry elasticsearch Integration](#)
- [OpenTelemetry Project](#)
- [OpenTelemetry Python Examples](#)

This library allows tracing HTTP elasticsearch made by the [elasticsearch](#) library.

**Warning:** The elasticsearch package got native OpenTelemetry support since version 8.13. To avoid duplicated tracing this instrumentation disables itself if it finds an elasticsearch client that has OpenTelemetry support enabled. Please be aware that the two libraries may use a different semantic convention, see [elasticsearch documentation](#).

### Usage

```
from opentelemetry.instrumentation.elasticsearch import ElasticsearchInstrumentor
import elasticsearch

# instrument elasticsearch
ElasticsearchInstrumentor().instrument()

# Using elasticsearch as normal now will automatically generate spans
es = elasticsearch.Elasticsearch()
es.index(index='my-index', doc_type='my-type', id=1, body={'my': 'data', 'timestamp':
↳ datetime.now()})
es.get(index='my-index', doc_type='my-type', id=1)
```

Elasticsearch instrumentation prefixes operation names with the string “Elasticsearch”. This can be changed to a different string by either setting the `OTEL_PYTHON_ELASTICSEARCH_NAME_PREFIX` environment variable or by passing the prefix as an argument to the instrumentor. For example,

```
ElasticsearchInstrumentor("my-custom-prefix").instrument()
```

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, method: str, url: str, kwargs)`

`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, response: dict)`

for example:

## API

```
class opentelemetry.instrumentation.elasticsearch.ElasticsearchInstrumentor(*args,
                                                                           **kwargs)
```

Bases: [BaseInstrumentor](#)

An instrumentor for elasticsearch 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 pyproject.toml.

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 Falcon Instrumentation

### 2.1.19 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 tracking, set the environment variable `OTEL_PYTHON_FASTAPI_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` to cover all instrumentations) to a string of comma delimited regexes that match the URLs.

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 comma delimited regexes directly to the `instrument_app` method:

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

### Request/Response hooks

This instrumentation supports request and response hooks. These are functions that get called right after a span is created for a request and right before the span is finished for the response.

- The server request hook is passed a server span and ASGI scope object for every incoming request.
- The client request hook is called with the internal span, and ASGI scope and event when the method `receive` is called.
- The client response hook is called with the internal span, and ASGI scope and event when the method `send` is called.

```
def server_request_hook(span: Span, scope: dict[str, Any]):
    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[str, Any], message: dict[str, Any]):
    if span and span.is_recording():
```

(continues on next page)



(continued from previous page)

```

        span.set_attribute("custom_user_attribute_from_client_request_hook", "some-value")
    )

    def client_response_hook(span: Span, scope: dict[str, Any], message: dict[str, Any]):
        if span and span.is_recording():
            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 specified HTTP headers as span attributes, according to the [semantic convention](#).

### Request headers

To capture HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma delimited 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 the request headers and add them as span attributes.

Request header names in FastAPI are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="Accept.*,X-.*"
```

Would match all request headers that start with `Accept` and `X-`.

To capture all request headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to `".*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST=".*"
```

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

For example: `http.request.header.custom_request_header = ["<value1>", "<value2>"]`

## Response headers

To capture HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma delimited 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 the response headers and add them as span attributes.

Response header names in FastAPI are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="Content.*,X-.*"
```

Would match all response headers that start with `Content` and `X-`.

To capture all response headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to `"*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="*"
```

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

For example: `http.response.header.custom_response_header = ["<value1>", "<value2>"]`

## Sanitizing headers

In order to prevent storing sensitive data such as personally identifiable information (PII), session keys, passwords, etc, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS` to a comma delimited list of HTTP header names to be sanitized. Regexes may be used, and all header names will be matched in a case-insensitive manner.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS="*.session.*,set-cookie"
```

will replace the value of headers such as `session-id` and `set-cookie` with `[REDACTED]` in the span.

---

**Note:** The environment variable names used 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, meter\_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 pyproject.toml.

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]

### 2.1.20 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.

## SQLCOMMENTER

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

## Usage

```
from opentelemetry.instrumentation.flask import FlaskInstrumentor

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

For example, `FlaskInstrumentor` when used with `SQLAlchemyInstrumentor` or `Psycopg2Instrumentor`, 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 /*metrics=value*/;
```

In order for the commenter to append flask related tags to sql queries, the commenter needs to be enabled on the respective SQLAlchemyInstrumentor or Psycopg2Instrumentor framework too.

### SQLCommenter Configurations

We can configure the tags to be appended to the sqlquery log by adding configuration inside `commenter_options={}` dict.

For example, enabling this flag will add flask and its version which is `/*flask%%3A2.9.3*/` to the SQL query as a comment (default is True):

```
framework = True
```

For example, enabling this flag will add route uri `/*route='/home'*/` to the SQL query as a comment (default is True):

```
route = True
```

For example, enabling this flag will add controller name `/*controller='home_view'*/` to the SQL query as a comment (default is True):

```
controller = True
```

### 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 tracking, set the environment variable `OTEL_PYTHON_FLASK_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` to cover all instrumentations) to a string of comma delimited regexes that match the URLs.

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 comma delimited regexes directly to the `instrument_app` method:

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

## Request/Response hooks

This instrumentation supports request and response hooks. These are functions that get called right after a span is created for a request and right before the span is finished for the response.

- The client request hook is called with the internal span and an instance of `WSGIEnvironment` (`flask.request.environ`) when the method `receive` is called.
- The client response hook is called with the internal span, the status of the response and a list of key-value (tuples) representing the response headers returned from the response when the method `send` is called.

For example,

```
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")

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

Flask Request object reference: <https://flask.palletsprojects.com/en/2.1.x/api/#flask.Request>

## Capture HTTP request and response headers

You can configure the agent to capture specified HTTP headers as span attributes, according to the [semantic convention](#).

### Request headers

To capture HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma delimited 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 the request headers and add them as span attributes.

Request header names in Flask are case-insensitive and `-` characters are replaced by `_`. So, giving the header name as `CUStom_Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="Accept.*,X-.*"
```

Would match all request headers that start with `Accept` and `X-`.

To capture all request headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to `".*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST=".*"
```

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

For example: `http.request.header.custom_request_header = ["<value1>,<value2>"]`

## Response headers

To capture HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma delimited 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 the response headers and add them as span attributes.

Response header names in Flask are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="Content.*,X-.*"
```

Would match all response headers that start with `Content` and `X-`.

To capture all response headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to `".*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE=".*"
```

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

For example: `http.response.header.custom_response_header = ["<value1>,<value2>"]`

## Sanitizing headers

In order to prevent storing sensitive data such as personally identifiable information (PII), session keys, passwords, etc, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS` to a comma delimited list of HTTP header names to be sanitized. Regexes may be used, and all header names will be matched in a case-insensitive manner.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS=".*session.*,set-cookie"
```

will replace the value of headers such as `session-id` and `set-cookie` with `[REDACTED]` in the span.

---

**Note:** The environment variable names used 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 pyproject.toml.

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, enable_commenter=True, commenter_options=None,
                      meter_provider=None)
```

```
static uninstrument_app(app)
```

## 2.1.21 OpenTelemetry gRPC Instrumentation

### Module contents

## 2.1.22 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://some.url/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://some.url/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://some.url/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

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

async def async_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,
    async_request_hook=async_request_hook,
    async_response_hook=async_response_hook
)

```

Or if you are using the transport classes directly:

```

from opentelemetry.instrumentation.httpx import SyncOpenTelemetryTransport, \
    AsyncOpenTelemetryTransport

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

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

async def async_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,

```

(continues on next page)

(continued from previous page)

```

        request_hook=request_hook,
        response_hook=response_hook
    )

    async_transport = httpx.AsyncHTTPTransport()
    async_telemetry_transport = AsyncOpenTelemetryTransport(
        async_transport,
        request_hook=async_request_hook,
        response_hook=async_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://some.url/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://some.url/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://some.url/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

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

async def async_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,
    async_request_hook=async_request_hook,
    async_response_hook=async_response_hook
)

```

Or if you are using the transport classes directly:

```

from opentelemetry.instrumentation.httpx import SyncOpenTelemetryTransport, \
    AsyncOpenTelemetryTransport

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

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

async def async_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,

```

(continues on next page)

(continued from previous page)

```

        request_hook=request_hook,
        response_hook=response_hook
    )

    async_transport = httpx.AsyncHTTPTransport()
    async_telemetry_transport = AsyncOpenTelemetryTransport(
        async_transport,
        request_hook=async_request_hook,
        response_hook=async_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, int | None, bytes]`

Alias for field number 1

**headers:** `List[Tuple[bytes, bytes]] | None`

Alias for field number 2

**stream:** `SyncByteStream | AsyncByteStream | None`

Alias for field number 3

**extensions:** `dict | None`

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:** `List[Tuple[bytes, bytes]] | None`

Alias for field number 1

**stream:** `Iterable[bytes]`

Alias for field number 2

**extensions:** `dict | None`

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]

**close**()

**Return type**

None

```
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](#)], Awaitable[Any]]]) – A hook that receives the span and request that is called right after the span is created
- **response\_hook** (Optional[Callable[[Span, [RequestInfo](#), [ResponseInfo](#)], Awaitable[Any]]]) – 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]

**async aclose**()

**Return type**

None

```
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 pyproject.toml.

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** (Union[Callable[[Span, RequestInfo], None], None, Callable[[Span, RequestInfo], Awaitable[Any]]]) – A hook that receives the span and request that is called right after the span is created
- **response\_hook** (Union[Callable[[Span, RequestInfo, ResponseInfo], None], None, Callable[[Span, RequestInfo, ResponseInfo], Awaitable[Any]]]) – 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

## 2.1.23 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 pyproject.toml.

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]

## 2.1.24 OpenTelemetry kafka-python integration

### Installation

```
pip install opentelemetry-instrumentation-kafka-python
```

### API

Instrument kafka-python to report instrumentation-kafka produced and consumed messages

### Usage

..code:: python

```
from opentelemetry.instrumentation.kafka import KafkaInstrumentor
from kafka import KafkaProducer, KafkaConsumer

# Instrument kafka
KafkaInstrumentor().instrument()

# report a span of type producer with the default settings
producer = KafkaProducer(bootstrap_servers=['localhost:9092'])
producer.send('my-topic', b'raw_bytes')

# report a span of type consumer with the default settings
consumer = KafkaConsumer('my-topic', group_id='my-group', bootstrap_servers=['localhost:9092'])
for message in consumer:
    # process message
```

The `_instrument()` method accepts the following keyword args: `tracer_provider` (`TracerProvider`) - an optional tracer provider `produce_hook` (`Callable`) - a function with extra user-defined logic to be performed before sending the message this function signature is: `def produce_hook(span: Span, args, kwargs)` `consume_hook` (`Callable`) - a function with extra user-defined logic to be performed after consuming a message this function signature is: `def consume_hook(span: Span, record: kafka.record.ABCRecord, args, kwargs)` for example:

## API

**class** `opentelemetry.instrumentation.kafka.KafkaInstrumentor(*args, **kwargs)`

Bases: `BaseInstrumentor`

An instrumentor for kafka module 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 pyproject.toml.

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]`

### 2.1.25 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`
- `otelTraceSampled`

The integration uses the following logging format by default:

```
%(asctime)s %(levelname)s [%s] [%s] [trace_id=%(otelTraceID)s,
↳ span_id=%(otelSpanID)s resource.service.name=%(otelServiceName)s trace_sampled=
↳ %(otelTraceSampled)s] - %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 trace_sampled=
↳ %(otelTraceSampled)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 %(otelTraceSampled)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
↳ trace_sampled=%(otelTraceSampled)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 pyproject.toml.

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 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)")
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 pyproject.toml.

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.

## 2.1.27 OpenTelemetry mysqlclient Instrumentation

The integration with MySQLClient supports the [MySQLClient](#) library and can be enabled by using `MySQLClientInstrumentor`.

### Usage

```
import MySQLdb
from opentelemetry.instrumentation.mysqlclient import MySQLClientInstrumentor

MySQLClientInstrumentor().instrument()

cnx = MySQLdb.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.mysqlclient.**MySQLClientInstrumentor**(\*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 pyproject.toml.

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 mysqlclient 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 mysqlclient connection.

**Parameters**

**connection** – The connection to uninstrument.

**Returns**

An uninstrumented connection.

## 2.1.28 OpenTelemetry Pika Instrumentation

Instrument pika to trace RabbitMQ applications.

### Usage

- Start broker backend

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

- Run instrumented task

```
import pika
from opentelemetry.instrumentation.pika import PikaInstrumentor

PikaInstrumentor().instrument()

connection = pika.BlockingConnection(pika.URLParameters('amqp://localhost'))
channel = connection.channel()
channel.queue_declare(queue='hello')
channel.basic_publish(exchange='', routing_key='hello', body=b'Hello World!')
```

- PikaInstrumentor also supports instrumentation of a single channel

```
import pika
from opentelemetry.instrumentation.pika import PikaInstrumentor

connection = pika.BlockingConnection(pika.URLParameters('amqp://localhost'))
channel = connection.channel()
channel.queue_declare(queue='hello')

pika_instrumentation = PikaInstrumentor()
pika_instrumentation.instrument_channel(channel=channel)

channel.basic_publish(exchange='', routing_key='hello', body=b'Hello World!')

pika_instrumentation.uninstrument_channel(channel=channel)
```

- PikaInstrumentor also supports instrumentation without creating an object, and receiving a tracer\_provider

```
PikaInstrumentor.instrument_channel(channel, tracer_provider=tracer_provider)
```

- PikaInstrumentor also supports instrumenting with hooks that will be called when producing or consuming a message. The hooks should be of type “Callable[[Span, bytes, BasicProperties], None]” where the first parameter is the span, the second parameter is the message body and the third parameter is the message properties

```
def publish_hook(span: Span, body: bytes, properties: BasicProperties):
    span.set_attribute("messaging.payload", body.decode())

def consume_hook(span: Span, body: bytes, properties: BasicProperties):
    span.set_attribute("messaging.id", properties.message_id)

PikaInstrumentor.instrument_channel(channel, publish_hook=publish_hook, consume_
hook=consume_hook)
```



## Consumer Instrumentation

For consumer instrumentation, pika supports two consuming modes:

- Consumers using the `basic_consume` method which accepts a callback. This is supported for global instrumentation (`PikaInstrumentor().instrument()`) as well channel specific instrumentation (`PikaInstrumentor().instrument_channel(channel)`)
- Consumers using the `consume` method which returns a generator over messages. This is supported for global instrumentations only (`PikaInstrumentor().instrument()`)

## API

```
class opentelemetry.instrumentation.pika.PikaInstrumentor(*args, **kwargs)
```

Bases: [BaseInstrumentor](#)

```
CONSUMER_CALLBACK_ATTR = 'on_message_callback'
```

```
static instrument_channel(channel, tracer_provider=None, publish_hook=<function
dummy_callback>, consume_hook=<function dummy_callback>)
```

**Return type**

None

```
static uninstrument_channel(channel)
```

**Return type**

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 pyproject.toml.

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]

### 2.1.29 OpenTelemetry Psycopg Instrumentation

The integration with PostgreSQL supports the [Psycopg](#) library, it can be enabled by using `PsycopgInstrumentor`.

## SQLCOMMENTER

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

### Usage

```
from opentelemetry.instrumentation.psycopg import PsycopgInstrumentor

PsycopgInstrumentor().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 psycopg and it's version which is `/psycopg%%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 psycopg
from opentelemetry.instrumentation.psycopg import PsycopgInstrumentor

PsycopgInstrumentor().instrument()

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

## API

**class** opentelemetry.instrumentation.psycopg.**PsycopgInstrumentor**(\*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 pyproject.toml.

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.psycopg.**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.psycopg.DatabaseApiAsyncIntegration(name,
                                                                    database_system,
                                                                    connection_attributes=None,
                                                                    version="",
                                                                    tracer_provider=None,
                                                                    capture_parameters=False,
                                                                    enable_commenter=False,
                                                                    commenter_options=None,
                                                                    connect_module=None)
```

Bases: [DatabaseApiIntegration](#)

**async wrapped\_connection**(connect\_method, args, kwargs)

Add object proxy to connection object.

```
class opentelemetry.instrumentation.psycopg.CursorTracer(db_api_integration)
```

Bases: [CursorTracer](#)

**get\_operation\_name**(cursor, args)

**get\_statement**(cursor, args)

## 2.1.30 OpenTelemetry Psycopg2 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*;"
```

## SQLCommitter Configurations

We can configure the tags to be appended to the sqlquery log by adding configuration inside `committer_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 `pyproject.toml`.

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,
                                                                    connec-
                                                                    tion_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*)

## 2.1.31 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 pyproject.toml.

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 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` *capture\_statement* (bool) - an optional value to enable capturing the database statement that is being executed

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>,
                                                         capture_statement=False)
```

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 pyproject.toml.

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]`

### 2.1.33 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 pyproject.toml.

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.

### 2.1.34 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 explicitly add `opentelemetry.instrumentation.pyramid.trace_tween_factory` 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 tracking, set the environment variable `OTEL_PYTHON_PYRAMID_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` to cover all instrumentations) to a string of comma delimited regexes that match the URLs.

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 specified HTTP headers as span attributes, according to the [semantic convention](#).

#### Request headers

To capture HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma delimited 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 the request headers and add them as span attributes.

Request header names in Pyramid are case-insensitive and `-` characters are replaced by `_`. So, giving the header name as `CUStom_Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="Accept.*,X-.*"
```

Would match all request headers that start with `Accept` and `X-`.

To capture all request headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to `".*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST=".*"
```

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

For example: `http.request.header.custom_request_header = ["<value1>,<value2>"]`

## Response headers

To capture HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma delimited 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 the response headers and add them as span attributes.

Response header names in Pyramid are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="Content.*,X-.*"
```

Would match all response headers that start with `Content` and `X-`.

To capture all response headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to `"*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="*"
```

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

For example: `http.response.header.custom_response_header = ["<value1>,<value2>"]`

## Sanitizing headers

In order to prevent storing sensitive data such as personally identifiable information (PII), session keys, passwords, etc, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS` to a comma delimited list of HTTP header names to be sanitized. Regexes may be used, and all header names will be matched in a case-insensitive manner.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS="*.session.*,set-cookie"
```

will replace the value of headers such as `session-id` and `set-cookie` with `[REDACTED]` in the span.

---

**Note:** The environment variable names used 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 pyproject.toml.

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)**

### 2.1.35 OpenTelemetry Redis Instrumentation

Instrument *redis* to report Redis queries.

There are two options for instrumenting code. The first option is to use the `opentelemetry-instrument` 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
```

(continues on next page)

(continued from previous page)

```
# 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 pyproject.toml.

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]`

## 2.1.36 OpenTelemetry Remoulade Instrumentation

This library allows tracing requests made by the Remoulade library.

## Installation

```
pip install opentelemetry-instrumentation-remoulade
```

## References

- [OpenTelemetry Remoulade Instrumentation](#)
- [OpenTelemetry Project](#)
- [OpenTelemetry Python Examples](#)

## 2.1.37 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

### Request/Response hooks

The requests 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_obj` is an instance of requests.PreparedRequest
def request_hook(span, request_obj):
    pass

# `request_obj` is an instance of requests.PreparedRequest
# `response` is an instance of requests.Response
def response_hook(span, request_obj, response)
    pass

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

## 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}`. [https://opentelemetry.io/docs/reference/specification/trace/semantic\\_conventions/http/#name](https://opentelemetry.io/docs/reference/specification/trace/semantic_conventions/http/#name)

### Parameters

**method** – string representing HTTP method

### Returns

span 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 `pyproject.toml`.

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.

## 2.1.38 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(text("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
```

(continues on next page)

(continued from previous page)

```
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 pyproject.toml.

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]

### 2.1.39 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 pyproject.toml.

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.

## 2.1.40 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 tracking, set the environment variable `OTEL_PYTHON_STARLETTE_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` to cover all instrumentations) to a string of comma delimited regexes that match the URLs.

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

This instrumentation supports request and response hooks. These are functions that get called right after a span is created for a request and right before the span is finished for the response.

- The server request hook is passed a server span and ASGI scope object for every incoming request.
- The client request hook is called with the internal span, and ASGI scope and event when the method `receive` is called.
- The client response hook is called with the internal span, and ASGI scope and event when the method `send` is called.

For example,

```
def server_request_hook(span: Span, scope: dict[str, Any]):
    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[str, Any], message: dict[str, Any]):
    if span and span.is_recording():
        span.set_attribute("custom_user_attribute_from_client_request_hook", "some-value")
    ↪)

def client_response_hook(span: Span, scope: dict[str, Any], message: dict[str, Any]):
    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 specified HTTP headers as span attributes, according to the [semantic convention](#).

### Request headers

To capture HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma delimited 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 the request headers and add them as span attributes.

Request header names in Starlette are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="Accept.*,X-.*"
```

Would match all request headers that start with `Accept` and `X-`.

Additionally, the special keyword `all` can be used to capture all request headers.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="all"
```

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

For example: `http.request.header.custom_request_header = ["<value1>", "<value2>"]`

## Response headers

To capture HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma delimited 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 the response headers and add them as span attributes.

Response header names in Starlette are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="Content.*,X-.*"
```

Would match all response headers that start with `Content` and `X-`.

Additionally, the special keyword `all` can be used to capture all response headers.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="all"
```

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

For example: `http.response.header.custom_response_header = ["<value1>", "<value2>"]`

## Sanitizing headers

In order to prevent storing sensitive data such as personally identifiable information (PII), session keys, passwords, etc, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS` to a comma delimited list of HTTP header names to be sanitized. Regexes may be used, and all header names will be matched in a case-insensitive manner.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS=".*session.*,set-cookie"
```

will replace the value of headers such as `session-id` and `set-cookie` with `[REDACTED]` in the span.

**Note:** The environment variable names used 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, meter\_provider=None, tracer\_provider=None)

Instrument an uninstrumented Starlette 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 pyproject.toml.

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]

### 2.1.41 OpenTelemetry system metrics Instrumentation

Instrument to report system (CPU, memory, network) and process (CPU, memory, garbage collection) metrics. By default, the following metrics are configured:

```
{
  "system.cpu.time": ["idle", "user", "system", "irq"],
  "system.cpu.utilization": ["idle", "user", "system", "irq"],
  "system.memory.usage": ["used", "free", "cached"],
  "system.memory.utilization": ["used", "free", "cached"],
  "system.swap.usage": ["used", "free"],
  "system.swap.utilization": ["used", "free"],
  "system.disk.io": ["read", "write"],
  "system.disk.operations": ["read", "write"],
  "system.disk.time": ["read", "write"],
  "system.network.dropped.packets": ["transmit", "receive"],
  "system.network.packets": ["transmit", "receive"],
  "system.network.errors": ["transmit", "receive"],
```

(continues on next page)

(continued from previous page)

```

"system.network.io": ["transmit", "receive"],
"system.network.connections": ["family", "type"],
"system.thread_count": None
"process.runtime.memory": ["rss", "vms"],
"process.runtime.cpu.time": ["user", "system"],
"process.runtime.gc_count": None,
"process.runtime.thread_count": None,
"process.runtime.cpu.utilization": None,
"process.runtime.context_switches": ["involuntary", "voluntary"],
}

```

## Usage

```

from opentelemetry.metrics import set_meter_provider
from opentelemetry.instrumentation.system_metrics import SystemMetricsInstrumentor
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.sdk.metrics.export import ConsoleMetricExporter,
↳ PeriodicExportingMetricReader

exporter = ConsoleMetricExporter()

set_meter_provider(MeterProvider([PeriodicExportingMetricReader(exporter)]))
SystemMetricsInstrumentor().instrument()

# metrics are collected asynchronously
input("...")

# to configure custom metrics
configuration = {
    "system.memory.usage": ["used", "free", "cached"],
    "system.cpu.time": ["idle", "user", "system", "irq"],
    "system.network.io": ["transmit", "receive"],
    "process.runtime.memory": ["rss", "vms"],
    "process.runtime.cpu.time": ["user", "system"],
    "process.runtime.context_switches": ["involuntary", "voluntary"],
}
SystemMetricsInstrumentor(config=configuration).instrument()

```

## API

**class** opentelemetry.instrumentation.system\_metrics.**SystemMetricsInstrumentor**(*labels=None*,  
*config=None*)

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 pyproject.toml.

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]

## 2.1.42 OpenTelemetry Threading Instrumentation

Instrument threading to propagate OpenTelemetry context.

### Usage

```
from opentelemetry.instrumentation.threading import ThreadingInstrumentor

ThreadingInstrumentor().instrument()
```

This library provides instrumentation for the *OpenTelemetry Threading Instrumentation* module to ensure that the OpenTelemetry context is propagated across threads. It is important to note that this instrumentation does not produce any telemetry data on its own. It merely ensures that the context is correctly propagated when threads are used.

When instrumented, new threads created using `threading.Thread`, `threading.Timer`, or within `futures.ThreadPoolExecutor` will have the current OpenTelemetry context attached, and this context will be re-activated in the thread's run method or the executor's worker thread."

```
class opentelemetry.instrumentation.threading.ThreadingInstrumentor(*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 pyproject.toml.

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]

## 2.1.43 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
```

(continues on next page)

(continued from previous page)

```

# `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_response_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_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 tornado are case insensitive. So, giving header name as `CUSTOMHeader` in environment variable will be able to 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 pyproject.toml.

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, server_histograms, cls, request_hook=None)`

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

## 2.1.44 OpenTelemetry Tortoise ORM Instrumentation

This library allows tracing queries made by tortoise ORM backends, mysql, postgres and sqlite.

### Installation

```
pip install opentelemetry-instrumentation-tortoiseorm
```

### References

- [OpenTelemetry Project](#)
- [Tortoise ORM](#)
- [OpenTelemetry Python Examples](#)

## 2.1.45 OpenTelemetry urllib Instrumentation

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

### Usage

```
from urllib import request
from opentelemetry.instrumentation.urllib import URLLibInstrumentor

# You can optionally pass a custom TracerProvider to
# URLLibInstrumentor().instrument()

URLLibInstrumentor().instrument()
req = request.Request('https://postman-echo.com/post', method="POST")
r = request.urlopen(req)
```

### Configuration

#### Request/Response hooks

The `urllib` 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_obj` is an instance of urllib.request.Request
def request_hook(span, request_obj):
    pass

# `request_obj` is an instance of urllib.request.Request
# `response` is an instance of http.client.HTTPResponse
def response_hook(span, request_obj, response)
```

(continues on next page)

(continued from previous page)

```

pass

URLLibInstrumentor.instrument(
    request_hook=request_hook, response_hook=response_hook
)

```

## Exclude lists

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

For example,

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

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

## API

**class** `opentelemetry.instrumentation.urlib.URLLibInstrumentor(*args, **kwargs)`

Bases: [BaseInstrumentor](#)

An instrumentor for urllib 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 pyproject.toml.

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]

**uninstrument\_opener(*opener*)**

uninstrument\_opener a specific instance of urllib.request.OpenerDirector

### 2.1.46 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
)
```

### Exclude lists

To exclude certain URLs from being tracked, set the environment variable `OTEL_PYTHON_URLLIB3_EXCLUDED_URLS` (or `OTEL_PYTHON_EXCLUDED_URLS` as fallback) with comma delimited regexes representing which URLs to exclude.

For example,

```
export OTEL_PYTHON_URLLIB3_EXCLUDED_URLS="client/.*/info,healthcheck"
```

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

## 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 pyproject.toml.

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]

### 2.1.47 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(
        ("localhost", 5100), func, server_name="localhost"
    )
    server.start()
```

## Configuration

### Request/Response hooks

This instrumentation supports request and response hooks. These are functions that get called right after a span is created for a request and right before the span is finished for the response.

- The client request hook is called with the internal span and an instance of `WSGIEnvironment` when the method `receive` is called.
- The client response hook is called with the internal span, the status of the response and a list of key-value (tuples) representing the response headers returned from the response when the method `send` is called.

For example,

```
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 specified HTTP headers as span attributes, according to the [semantic convention](#).

### Request headers

To capture HTTP request headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to a comma delimited 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 the request headers and add them as span attributes.

Request header names in WSGI are case-insensitive and `-` characters are replaced by `_`. So, giving the header name as `CUStom_Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST="Accept.*,X-.*"
```

Would match all request headers that start with `Accept` and `X-`.

To capture all request headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST` to `".*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_REQUEST=".*"
```

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

For example: `http.request.header.custom_request_header = [<value1>,<value2>"]`

### Response headers

To capture HTTP response headers as span attributes, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to a comma delimited 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 the response headers and add them as span attributes.

Response header names in WSGI are case-insensitive. So, giving the header name as `CUStom-Header` in the environment variable will capture the header named `custom-header`.

Regular expressions may also be used to match multiple headers that correspond to the given pattern. For example:

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE="Content.*,X-.*"
```

Would match all response headers that start with Content and X-.

To capture all response headers, set `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE` to `".*"`.

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SERVER_RESPONSE=".*"
```

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

For example: `http.response.header.custom_response_header = [<value1>,<value2>"]`

## Sanitizing headers

In order to prevent storing sensitive data such as personally identifiable information (PII), session keys, passwords, etc, set the environment variable `OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS` to a comma delimited list of HTTP header names to be sanitized. Regexes may be used, and all header names will be matched in a case-insensitive manner.

For example,

```
export OTEL_INSTRUMENTATION_HTTP_CAPTURE_HEADERS_SANITIZE_FIELDS=".*session.*,set-cookie"
```

will replace the value of headers such as `session-id` and `set-cookie` with `[REDACTED]` in the span.

**Note:** The environment variable names used to capture HTTP headers are still experimental, and thus are subject to change.

## Sanitizing methods

In order to prevent unbound cardinality for HTTP methods by default nonstandard ones are labeled as `NONSTANDARD`. To record all of the names set the environment variable `OTEL_PYTHON_INSTRUMENTATION_HTTP_CAPTURE_ALL_METHODS` to a value that evaluates to true, e.g. `1`.

## 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.set_if_not_none(dic, key, value)``opentelemetry.instrumentation.wsgi.collect_request_attributes(environ,  
sem_conv_opt_in_mode=_HTTPStabilityMode.DEFAULT)`

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](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](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,  
duration_attrs=None,  
sem_conv_opt_in_mode=_HTTPStabilityMode.DEFAULT)`

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 span name is the HTTP method and URL path, or just the method. <https://github.com/open-telemetry/opentelemetry-specification/pull/3165> [https://opentelemetry.io/docs/reference/specification/trace/semantic\\_conventions/http/#name](https://opentelemetry.io/docs/reference/specification/trace/semantic_conventions/http/#name)**Parameters****environ** – The WSGI environ object.**Returns**

The span 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.
- **meter\_provider** – Optional meter provider to use. If omitted the current globally configured one is used.

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

Bases: object

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

## 2.1.48 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)) – an 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*.

**class** opentelemetry.propagators.aws.AwsXRayLambdaPropagator

Bases: *AwsXRayPropagator*

Implementation of the AWS X-Ray Trace Header propagation protocol but with special handling for Lambda's `_X_AMZN_TRACE_ID` environment variable.

**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.

## 2.1.49 Performance Tests - Benchmarks

Click [here](#) to view the latest performance benchmarks for packages in this repo.

## 2.1.50 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

**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

## 2.1.51 OpenTelemetry Python - Resource Detector for Containers

**class** `opentelemetry.resource.detector.container.ContainerResourceDetector`(*raise\_on\_error=False*)

Bases: `ResourceDetector`

Detects container.id only available when app is running inside the docker container and return it in a `Resource`

**detect()****Return type**

`Resource`



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### O

`opentelemetry.instrumentation`, 19  
`opentelemetry.instrumentation.aiohttp_client`, 5  
`opentelemetry.instrumentation.aiopg`, 8  
`opentelemetry.instrumentation.asgi`, 9  
`opentelemetry.instrumentation.asyncio`, 14  
`opentelemetry.instrumentation.asyncpg`, 17  
`opentelemetry.instrumentation.aws_lambda`, 18  
`opentelemetry.instrumentation.boto`, 20  
`opentelemetry.instrumentation.boto3sqs`, 21  
`opentelemetry.instrumentation.botocore`, 23  
`opentelemetry.instrumentation.celery`, 24  
`opentelemetry.instrumentation.dbapi`, 26  
`opentelemetry.instrumentation.django`, 29  
`opentelemetry.instrumentation.elasticsearch`, 34  
`opentelemetry.instrumentation.fastapi`, 36  
`opentelemetry.instrumentation.flask`, 39  
`opentelemetry.instrumentation.httpx`, 47  
`opentelemetry.instrumentation.instrumentor`, 19  
`opentelemetry.instrumentation.jinja2`, 52  
`opentelemetry.instrumentation.kafka`, 53  
`opentelemetry.instrumentation.logging`, 54  
`opentelemetry.instrumentation.mysql`, 57  
`opentelemetry.instrumentation.mysqlclient`, 58  
`opentelemetry.instrumentation.pika`, 59  
`opentelemetry.instrumentation.psycpg`, 61  
`opentelemetry.instrumentation.psycpg2`, 64  
`opentelemetry.instrumentation.pymemcache`, 66  
`opentelemetry.instrumentation.pymongo`, 67  
`opentelemetry.instrumentation.pymysql`, 68  
`opentelemetry.instrumentation.pyramid`, 69  
`opentelemetry.instrumentation.redis`, 73  
`opentelemetry.instrumentation.requests`, 75  
`opentelemetry.instrumentation.sqlalchemy`, 76  
`opentelemetry.instrumentation.sqlite3`, 78  
`opentelemetry.instrumentation.starlette`, 80  
`opentelemetry.instrumentation.system_metrics`, 83  
`opentelemetry.instrumentation.threading`, 85  
`opentelemetry.instrumentation.tornado`, 85  
`opentelemetry.instrumentation.urllib`, 89  
`opentelemetry.instrumentation.urllib3`, 90  
`opentelemetry.instrumentation.wsgi`, 92  
`opentelemetry.propagators.aws`, 97  
`opentelemetry.resource.detector.container`, 100  
`opentelemetry.sdk.extension.aws.trace.aws_xray_id_generator`, 99



## A

`aclose()` (opentelemetry.instrumentation.httpx.AsyncOpenTelemetryTransport method), 51

`add_response_attributes()` (in module *opentelemetry.instrumentation.wsgi*), 96

`add_span_arg_tags()` (in module *opentelemetry.instrumentation.boto*), 21

`AioHttpClientInstrumentor` (class in *opentelemetry.instrumentation.aiohttp\_client*), 7

`AiopgInstrumentor` (class in *opentelemetry.instrumentation.aiopg*), 8

`ASGIGetter` (class in *opentelemetry.instrumentation.asgi*), 12

`ASGISetter` (class in *opentelemetry.instrumentation.asgi*), 13

`AsyncioInstrumentor` (class in *opentelemetry.instrumentation.asyncio*), 16

`AsyncOpenTelemetryTransport` (class in *opentelemetry.instrumentation.httpx*), 51

`AsyncPGInstrumentor` (class in *opentelemetry.instrumentation.asyncpg*), 18

`AwsLambdaInstrumentor` (class in *opentelemetry.instrumentation.aws\_lambda*), 19

`AwsXRayIdGenerator` (class in *opentelemetry.sdk.extension.aws.trace.aws\_xray\_id\_generator*), 99

`AwsXRayLambdaPropagator` (class in *opentelemetry.propagators.aws*), 98

`AwsXRayPropagator` (class in *opentelemetry.propagators.aws*), 97

## B

`BaseInstrumentor` (class in *opentelemetry.instrumentation.instrumentor*), 19

`Boto3SQSGetter` (class in *opentelemetry.instrumentation.boto3sqs*), 22

`Boto3SQSInstrumentor` (class in *opentelemetry.instrumentation.boto3sqs*), 22

`Boto3SQSInstrumentor.ContextableList` (class in *opentelemetry.instrumentation.boto3sqs*), 23

`Boto3SQSSetter` (class in *opentelemetry.instrumentation.boto3sqs*), 22

`BotocoreInstrumentor` (class in *opentelemetry.instrumentation.botocore*), 24

`BotoInstrumentor` (class in *opentelemetry.instrumentation.boto*), 21

## C

`CeleryGetter` (class in *opentelemetry.instrumentation.celery*), 25

`CeleryInstrumentor` (class in *opentelemetry.instrumentation.celery*), 25

`close()` (*opentelemetry.instrumentation.httpx.SyncOpenTelemetryTransport* method), 51

`collect_custom_headers_attributes()` (in module *opentelemetry.instrumentation.asgi*), 13

`collect_custom_request_headers_attributes()` (in module *opentelemetry.instrumentation.wsgi*), 96

`collect_custom_response_headers_attributes()` (in module *opentelemetry.instrumentation.wsgi*), 96

`collect_request_attributes()` (in module *opentelemetry.instrumentation.asgi*), 13

`collect_request_attributes()` (in module *opentelemetry.instrumentation.wsgi*), 96

`CommandTracer` (class in *opentelemetry.instrumentation.pymongo*), 67

`CONSUMER_CALLBACK_ATTR` (*opentelemetry.instrumentation.pika.PikaInstrumentor* attribute), 61

`ContainerResourceDetector` (class in *opentelemetry.resource.detector.container*), 100

`create_celery_metrics()` (*opentelemetry.instrumentation.celery.CeleryInstrumentor* method), 26

`create_trace_config()` (in module *opentelemetry.instrumentation.aiohttp\_client*), 6

`current_context_token` (*opentelemetry.instrumentation.boto3sqs.Boto3SQSInstrumentor* attribute), 23

`current_span_related_to_token` (*opentelemetry-*

- try.instrumentation.boto3sqs.Boto3SQSInstrumentor* (class in *opentelemetry.instrumentation.boto3sqs*), 23
- CursorTracer* (class in *opentelemetry.instrumentation.dbapi*), 29
- CursorTracer* (class in *opentelemetry.instrumentation.psycopg*), 64
- CursorTracer* (class in *opentelemetry.instrumentation.psycopg2*), 66
- ## D
- DatabaseApiAsyncIntegration* (class in *opentelemetry.instrumentation.psycopg*), 63
- DatabaseApiIntegration* (class in *opentelemetry.instrumentation.dbapi*), 29
- DatabaseApiIntegration* (class in *opentelemetry.instrumentation.psycopg*), 63
- DatabaseApiIntegration* (class in *opentelemetry.instrumentation.psycopg2*), 66
- detect()* (*opentelemetry.resource.detector.container.ContainerResourceDetector* method), 100
- determine\_state()* (in module *opentelemetry.instrumentation.asyncio*), 17
- DjangoInstrumentor* (class in *opentelemetry.instrumentation.django*), 33
- dummy\_callback()* (in module *opentelemetry.instrumentation.pymongo*), 67
- ## E
- ElasticsearchInstrumentor* (class in *opentelemetry.instrumentation.elasticsearch*), 35
- environment variable
- OTEL\_PYTHON\_LOG\_CORRELATION*, 55
  - OTEL\_PYTHON\_LOG\_FORMAT*, 55
  - OTEL\_PYTHON\_LOG\_LEVEL*, 55
- extensions (*opentelemetry.instrumentation.httpx.RequestInfo* attribute), 50
- extensions (*opentelemetry.instrumentation.httpx.ResponseInfo* attribute), 50
- extract()* (*opentelemetry.propagators.aws.AwsXRayLambdaPropagator* method), 98
- extract()* (*opentelemetry.propagators.aws.AwsXRayPropagator* method), 97
- ## F
- failed()* (*opentelemetry.instrumentation.pymongo.CommandTracer* method), 68
- FastAPIInstrumentor* (class in *opentelemetry.instrumentation.fastapi*), 39
- fields* (*opentelemetry.propagators.aws.AwsXRayPropagator* property), 98
- FlaskInstrumentor* (class in *opentelemetry.instrumentation.flask*), 43
- flatten\_dict()* (in module *opentelemetry.instrumentation.boto*), 21
- ## G
- generate\_span\_id()* (*opentelemetry.sdk.extension.aws.trace.aws\_xray\_id\_generator.AwsXRayIdGenerator* method), 99
- generate\_trace\_id()* (*opentelemetry.sdk.extension.aws.trace.aws\_xray\_id\_generator.AwsXRayIdGenerator* method), 99
- get()* (*opentelemetry.instrumentation.asgi.ASGIGetter* method), 12
- get()* (*opentelemetry.instrumentation.boto3sqs.Boto3SQSGetter* method), 22
- get()* (*opentelemetry.instrumentation.celery.CeleryGetter* method), 25
- get()* (*opentelemetry.instrumentation.wsgi.WSGIGetter* method), 95
- get\_connection\_attributes()* (*opentelemetry.instrumentation.dbapi.DatabaseApiIntegration* method), 29
- get\_default\_span\_details()* (in module *opentelemetry.instrumentation.asgi*), 13
- get\_default\_span\_name()* (in module *opentelemetry.instrumentation.flask*), 43
- get\_default\_span\_name()* (in module *opentelemetry.instrumentation.requests*), 76
- get\_default\_span\_name()* (in module *opentelemetry.instrumentation.wsgi*), 96
- get\_host\_port\_url\_tuple()* (in module *opentelemetry.instrumentation.asgi*), 13
- get\_operation\_name()* (*opentelemetry.instrumentation.dbapi.CursorTracer* method), 29
- get\_operation\_name()* (*opentelemetry.instrumentation.psycopg.CursorTracer* method), 64
- get\_operation\_name()* (*opentelemetry.instrumentation.psycopg2.CursorTracer* method), 66
- get\_statement()* (*opentelemetry.instrumentation.dbapi.CursorTracer* method), 29
- get\_statement()* (*opentelemetry.instrumentation.psycopg.CursorTracer* method), 64
- get\_statement()* (*opentelemetry.instrumentation.psycopg2.CursorTracer* method), 66

- `get_traced_connection_proxy()` (in module `opentelemetry.instrumentation.dbapi`), 29
- `get_traced_cursor_proxy()` (in module `opentelemetry.instrumentation.dbapi`), 29
- ## H
- `handle_async_request()` (`opentelemetry.instrumentation.httpx.AsyncOpenTelemetryTransport` static method), 51
- `handle_request()` (`opentelemetry.instrumentation.httpx.SyncOpenTelemetryTransport` static method), 51
- `headers` (`opentelemetry.instrumentation.httpx.RequestInfo` attribute), 50
- `headers` (`opentelemetry.instrumentation.httpx.ResponseInfo` attribute), 50
- `HTTPXClientInstrumentor` (class in `opentelemetry.instrumentation.httpx`), 51
- ## I
- `inject()` (`opentelemetry.propagators.aws.AwsXRayPropagator` method), 97
- `instrument()` (`opentelemetry.instrumentation.instrumentor.BaseInstrumentor` method), 20
- `instrument_app()` (`opentelemetry.instrumentation.fastapi.FastAPIInstrumentor` static method), 39
- `instrument_app()` (`opentelemetry.instrumentation.flask.FlaskInstrumentor` static method), 43
- `instrument_app()` (`opentelemetry.instrumentation.starlette.StarletteInstrumentor` static method), 83
- `instrument_channel()` (`opentelemetry.instrumentation.pika.PikaInstrumentor` static method), 61
- `instrument_client()` (`opentelemetry.instrumentation.httpx.HTTPXClientInstrumentor` static method), 52
- `instrument_config()` (`opentelemetry.instrumentation.pyramid.PyramidInstrumentor` static method), 73
- `instrument_connection()` (in module `opentelemetry.instrumentation.dbapi`), 28
- `instrument_connection()` (`opentelemetry.instrumentation.aiopg.AiopgInstrumentor` method), 8
- `instrument_connection()` (`opentelemetry.instrumentation.mysql.MySQLInstrumentor` method), 58
- `instrument_connection()` (`opentelemetry.instrumentation.mysqlclient.MySQLClientInstrumentor` static method), 59
- `instrument_connection()` (`opentelemetry.instrumentation.psycopg.PsycopgInstrumentor` static method), 63
- `instrument_connection()` (`opentelemetry.instrumentation.psycopg2.Psycopg2Instrumentor` static method), 66
- `instrument_connection()` (`opentelemetry.instrumentation.pymysql.PyMySQLInstrumentor` static method), 69
- `instrument_connection()` (`opentelemetry.instrumentation.sqlite3.SQLite3Instrumentor` static method), 79
- `instrument_gather()` (`opentelemetry.instrumentation.asyncio.AsyncioInstrumentor` method), 16
- `instrument_method_with_coroutine()` (`opentelemetry.instrumentation.asyncio.AsyncioInstrumentor` method), 16
- `instrument_taskgroup_create_task()` (`opentelemetry.instrumentation.asyncio.AsyncioInstrumentor` method), 16
- `instrument_to_thread()` (`opentelemetry.instrumentation.asyncio.AsyncioInstrumentor` method), 16
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.aiohttp_client.AioHttpClientInstrumentor` method), 7
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.aiopg.AiopgInstrumentor` method), 8
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.asyncio.AsyncioInstrumentor` method), 16
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.asyncpg.AsyncPGInstrumentor` method), 18
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.aws_lambda.AwsLambdaInstrumentor` method), 19
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.boto.BotoInstrumentor` method), 21
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.boto3sqs.Boto3SQSInstrumentor` method), 23
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.botocore.BotocoreInstrumentor` method), 24
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.celery.CeleryInstrumentor` method), 26
- `instrumentation_dependencies()` (`opentelemetry.instrumentation.django.DjangoInstrumentor` method), 26

- method*), 33
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.elasticsearch.ElasticsearchInstrumentor* *method*), 35
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.fastapi.FastAPIInstrumentor* *method*), 39
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.flask.FlaskInstrumentor* *method*), 43
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.httpx.HTTPXClientInstrumentor* *method*), 51
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.instrumentor.BaseInstrumentor* *method*), 20
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.jinja2.Jinja2Instrumentor* *method*), 53
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.kafka.KafkaInstrumentor* *method*), 54
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.logging.LoggingInstrumentor* *method*), 57
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.mysql.MySQLInstrumentor* *method*), 57
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.mysqlclient.MySQLClientInstrumentor* *method*), 59
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.pika.PikaInstrumentor* *method*), 61
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.psycpg.PsycpgInstrumentor* *method*), 63
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.psycpg2.Psycpg2Instrumentor* *method*), 65
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.pymemcache.PymemcacheInstrumentor* *method*), 67
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.pymongo.PymongoInstrumentor* *method*), 68
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.pymysql.PyMySQLInstrumentor* *method*), 69
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.pyramid.PyramidInstrumentor* *method*), 73
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.redis.RedisInstrumentor* *method*), 74
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.requests.RequestsInstrumentor* *method*), 76
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.sqlalchemy.SQLAlchemyInstrumentor* *method*), 78
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.sqlite3.SQLite3Instrumentor* *method*), 79
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.starlette.StarletteInstrumentor* *method*), 83
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.system\_metrics.SystemMetricsInstrumentor* *method*), 84
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.threading.ThreadingInstrumentor* *method*), 85
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.tornado.TornadoInstrumentor* *method*), 88
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.urllib.URLLibInstrumentor* *method*), 90
- `instrumentation_dependencies()` (*opentelemetry.instrumentation.urllib3.URLLib3Instrumentor* *method*), 92
- `is_instrumented_by_opentelemetry` (*opentelemetry.instrumentation.instrumentor.BaseInstrumentor* *property*), 20
- ## J
- `Jinja2Instrumentor` (class in *opentelemetry.instrumentation.jinja2*), 53
- ## K
- `KafkaInstrumentor` (class in *opentelemetry.instrumentation.kafka*), 54
- `keys()` (*opentelemetry.instrumentation.asgi.ASGIGetter* *method*), 12
- `keys()` (*opentelemetry.instrumentation.boto3sqs.Boto3SQSGetter* *method*), 22
- `keys()` (*opentelemetry.instrumentation.celery.CeleryGetter* *method*), 25
- `keys()` (*opentelemetry.instrumentation.wsgi.WSGIGetter* *method*), 96
- ## L
- `LoggingInstrumentor` (class in *opentelemetry.instrumentation.logging*), 56
- ## M
- `method` (*opentelemetry.instrumentation.httpx.RequestInfo*



`attribute`), 50  
`methods_with_coroutine` (*opentelemetry.instrumentation.asyncio.AsyncioInstrumentor* `attribute`), 16  
`metrics` (*opentelemetry.instrumentation.celery.CeleryInstrumentor* `attribute`), 25  
`module`  
`opentelemetry.instrumentation`, 19  
`opentelemetry.instrumentation.aiohttp_client`, 5  
`opentelemetry.instrumentation.aiopg`, 8  
`opentelemetry.instrumentation.asgi`, 9  
`opentelemetry.instrumentation.asyncio`, 14  
`opentelemetry.instrumentation.asyncpg`, 17  
`opentelemetry.instrumentation.aws_lambda`, 18  
`opentelemetry.instrumentation.boto`, 20  
`opentelemetry.instrumentation.boto3sqs`, 21  
`opentelemetry.instrumentation.botocore`, 23  
`opentelemetry.instrumentation.celery`, 24  
`opentelemetry.instrumentation.dbapi`, 26  
`opentelemetry.instrumentation.django`, 29  
`opentelemetry.instrumentation.elasticsearch`, 34  
`opentelemetry.instrumentation.fastapi`, 36  
`opentelemetry.instrumentation.flask`, 39  
`opentelemetry.instrumentation.httpx`, 47  
`opentelemetry.instrumentation.instrumentor`, 19  
`opentelemetry.instrumentation.jinja2`, 52  
`opentelemetry.instrumentation.kafka`, 53  
`opentelemetry.instrumentation.logging`, 54  
`opentelemetry.instrumentation.mysql`, 57  
`opentelemetry.instrumentation.mysqlclient`, 58  
`opentelemetry.instrumentation.pika`, 59  
`opentelemetry.instrumentation.psycopg`, 61  
`opentelemetry.instrumentation.psycopg2`, 64  
`opentelemetry.instrumentation.pymemcache`, 66  
`opentelemetry.instrumentation.pymongo`, 67  
`opentelemetry.instrumentation.pymysql`, 68  
`opentelemetry.instrumentation.pyramid`, 69  
`opentelemetry.instrumentation.redis`, 73  
`opentelemetry.instrumentation.requests`, 75  
`opentelemetry.instrumentation.sqlalchemy`, 76  
`opentelemetry.instrumentation.sqlite3`, 78  
`opentelemetry.instrumentation.starlette`, 80  
`opentelemetry.instrumentation.system_metrics`, 83  
`opentelemetry.instrumentation.threading`, 85  
`opentelemetry.instrumentation.tornado`, 85  
`opentelemetry.instrumentation.urllib`, 89  
`opentelemetry.instrumentation.urllib3`, 90  
`opentelemetry.instrumentation.wsgi`, 92  
`opentelemetry.propagators.aws`, 97  
`opentelemetry.resource.detector.container`, 100  
`opentelemetry.sdk.extension.aws.trace.aws_xray_id_generator`, 99  
`MySQLClientInstrumentor` (*class in opentelemetry.instrumentation.mysqlclient*), 59  
`MySQLInstrumentor` (*class in opentelemetry.instrumentation.mysql*), 57

## O

`opentelemetry.instrumentation`  
`module`, 19  
`opentelemetry.instrumentation.aiohttp_client`  
`module`, 5  
`opentelemetry.instrumentation.aiopg`  
`module`, 8  
`opentelemetry.instrumentation.asgi`  
`module`, 9  
`opentelemetry.instrumentation.asyncio`  
`module`, 14  
`opentelemetry.instrumentation.asyncpg`  
`module`, 17  
`opentelemetry.instrumentation.aws_lambda`  
`module`, 18  
`opentelemetry.instrumentation.boto`  
`module`, 20  
`opentelemetry.instrumentation.boto3sqs`  
`module`, 21  
`opentelemetry.instrumentation.botocore`  
`module`, 23  
`opentelemetry.instrumentation.celery`  
`module`, 24  
`opentelemetry.instrumentation.dbapi`  
`module`, 26  
`opentelemetry.instrumentation.django`  
`module`, 29  
`opentelemetry.instrumentation.elasticsearch`  
`module`, 34  
`opentelemetry.instrumentation.fastapi`  
`module`, 36  
`opentelemetry.instrumentation.flask`  
`module`, 39  
`opentelemetry.instrumentation.httpx`  
`module`, 47  
`opentelemetry.instrumentation.instrumentor`

module, 19  
 opentelemetry.instrumentation.jinja2  
   module, 52  
 opentelemetry.instrumentation.kafka  
   module, 53  
 opentelemetry.instrumentation.logging  
   module, 54  
 opentelemetry.instrumentation.mysql  
   module, 57  
 opentelemetry.instrumentation.mysqlclient  
   module, 58  
 opentelemetry.instrumentation.pika  
   module, 59  
 opentelemetry.instrumentation.psycopg  
   module, 61  
 opentelemetry.instrumentation.psycopg2  
   module, 64  
 opentelemetry.instrumentation.pymemcache  
   module, 66  
 opentelemetry.instrumentation.pymongo  
   module, 67  
 opentelemetry.instrumentation.pymysql  
   module, 68  
 opentelemetry.instrumentation.pyramid  
   module, 69  
 opentelemetry.instrumentation.redis  
   module, 73  
 opentelemetry.instrumentation.requests  
   module, 75  
 opentelemetry.instrumentation.sqlalchemy  
   module, 76  
 opentelemetry.instrumentation.sqlite3  
   module, 78  
 opentelemetry.instrumentation.starlette  
   module, 80  
 opentelemetry.instrumentation.system\_metrics  
   module, 83  
 opentelemetry.instrumentation.threading  
   module, 85  
 opentelemetry.instrumentation.tornado  
   module, 85  
 opentelemetry.instrumentation.urllib  
   module, 89  
 opentelemetry.instrumentation.urllib3  
   module, 90  
 opentelemetry.instrumentation.wsgi  
   module, 92  
 opentelemetry.propagators.aws  
   module, 97  
 opentelemetry.resource.detector.container  
   module, 100  
 opentelemetry.sdk.extension.aws.trace.aws\_xray\_id\_generator  
   module, 99

OpenTelemetryMiddleware (class in *opentelemetry.instrumentation.asgi*), 14  
 OpenTelemetryMiddleware (class in *opentelemetry.instrumentation.wsgi*), 96  
 original\_handler\_new (opentelemetry.instrumentation.tornado.TornadoInstrumentor attribute), 88

## P

patch\_handler\_class() (in module *opentelemetry.instrumentation.tornado*), 88  
 patched\_handlers (opentelemetry.instrumentation.tornado.TornadoInstrumentor attribute), 88  
 PikaInstrumentor (class in *opentelemetry.instrumentation.pika*), 61  
 Psycopg2Instrumentor (class in *opentelemetry.instrumentation.psycopg2*), 65  
 PsycopgInstrumentor (class in *opentelemetry.instrumentation.psycopg*), 63  
 PymemcacheInstrumentor (class in *opentelemetry.instrumentation.pymemcache*), 67  
 PymongoInstrumentor (class in *opentelemetry.instrumentation.pymongo*), 68  
 PyMySQLInstrumentor (class in *opentelemetry.instrumentation.pymysql*), 69  
 PyramidInstrumentor (class in *opentelemetry.instrumentation.pyramid*), 73

## R

random\_id\_generator (opentelemetry.sdk.extension.aws.trace.aws\_xray\_id\_generator.AwsXRayIdGenerator attribute), 99  
 received\_messages\_spans (opentelemetry.instrumentation.boto3sqs.Boto3SQSInstrumentor attribute), 23  
 record\_process() (opentelemetry.instrumentation.asyncio.AsyncioInstrumentor method), 16  
 RedisInstrumentor (class in *opentelemetry.instrumentation.redis*), 74  
 RequestInfo (class in *opentelemetry.instrumentation.httpx*), 50  
 RequestsInstrumentor (class in *opentelemetry.instrumentation.requests*), 76  
 ResponseInfo (class in *opentelemetry.instrumentation.httpx*), 50  
 ResponsePropagationSetter (class in *opentelemetry.instrumentation.wsgi*), 97

## S

set() (opentelemetry.instrumentation.asgi.ASGISetter method), 13

- `set()` (`opentelemetry.instrumentation.boto3sqs.Boto3SQSInstrumentor` method), 22  
`set()` (`opentelemetry.instrumentation.wsgi.ResponsePropagationSetter` method), 29  
`set_status_code()` (in module `opentelemetry.instrumentation.asgi`), 13  
`setifnotnone()` (in module `opentelemetry.instrumentation.wsgi`), 96  
`SQLAlchemyInstrumentor` (class in `opentelemetry.instrumentation.sqlalchemy`), 78  
`SQLite3Instrumentor` (class in `opentelemetry.instrumentation.sqlite3`), 79  
`StarletteInstrumentor` (class in `opentelemetry.instrumentation.starlette`), 83  
`started()` (`opentelemetry.instrumentation.pymongo.CommandTracer` method), 68  
`status_code` (`opentelemetry.instrumentation.httpx.ResponseInfo` attribute), 50  
`stream()` (`opentelemetry.instrumentation.httpx.RequestInfo` attribute), 50  
`stream()` (`opentelemetry.instrumentation.httpx.ResponseInfo` attribute), 50  
`succeeded()` (`opentelemetry.instrumentation.pymongo.CommandTracer` method), 68  
`SyncOpenTelemetryTransport` (class in `opentelemetry.instrumentation.httpx`), 50  
`SystemMetricsInstrumentor` (class in `opentelemetry.instrumentation.system_metrics`), 84
- ## T
- `task_id_to_start_time` (`opentelemetry.instrumentation.celery.CeleryInstrumentor` attribute), 26  
`ThreadingInstrumentor` (class in `opentelemetry.instrumentation.threading`), 85  
`TornadoInstrumentor` (class in `opentelemetry.instrumentation.tornado`), 88  
`trace_coroutine()` (`opentelemetry.instrumentation.asyncio.AsyncioInstrumentor` method), 16  
`trace_future()` (`opentelemetry.instrumentation.asyncio.AsyncioInstrumentor` method), 16  
`trace_integration()` (in module `opentelemetry.instrumentation.dbapi`), 27  
`trace_item()` (`opentelemetry.instrumentation.asyncio.AsyncioInstrumentor` method), 16  
`trace_to_thread()` (`opentelemetry.instrumentation.asyncio.AsyncioInstrumentor` method), 16
- ## U
- `uninstrument()` (`opentelemetry.instrumentation.instrumentor.BaseInstrumentor` method), 20  
`uninstrument_app()` (`opentelemetry.instrumentation.fastapi.FastAPIInstrumentor` static method), 39  
`uninstrument_app()` (`opentelemetry.instrumentation.flask.FlaskInstrumentor` static method), 43  
`uninstrument_app()` (`opentelemetry.instrumentation.starlette.StarletteInstrumentor` static method), 83  
`uninstrument_channel()` (`opentelemetry.instrumentation.pika.PikaInstrumentor` static method), 61  
`uninstrument_client()` (`opentelemetry.instrumentation.httpx.HTTPXClientInstrumentor` static method), 52  
`uninstrument_config()` (`opentelemetry.instrumentation.pyramid.PyramidInstrumentor` static method), 73  
`uninstrument_connection()` (in module `opentelemetry.instrumentation.dbapi`), 28  
`uninstrument_connection()` (`opentelemetry.instrumentation.aiopg.AiopgInstrumentor` method), 8  
`uninstrument_connection()` (`opentelemetry.instrumentation.mysql.MySQLInstrumentor` method), 58  
`uninstrument_connection()` (`opentelemetry.instrumentation.mysqlclient.MySQLClientInstrumentor` static method), 59  
`uninstrument_connection()` (`opentelemetry.instrumentation.psycopg.PsycopgInstrumentor` static method), 63  
`uninstrument_connection()` (`opentelemetry.instrumentation.psycopg2.Psycopg2Instrumentor` static method), 66  
`uninstrument_connection()` (`opentelemetry.instrumentation.pymysql.PyMySQLInstrumentor` static method), 69  
`uninstrument_connection()` (`opentelemetry.instrumentation.sqlite3.SQLite3Instrumentor` static method), 79  
`uninstrument_gather()` (in module `opentelemetry.instrumentation.asyncio`), 17  
`uninstrument_method_with_coroutine()` (in module `opentelemetry.instrumentation.asyncio`), 17  
`uninstrument_opener()` (`opentelemetry`

*try.instrumentation.urllib.URLLibInstrumentor*  
*method*), 90

*uninstrument\_session()* (*opentelemetry.instrumentation.aiohttp\_client.AioHttpClientInstrumentor*  
*static method*), 7

*uninstrument\_session()* (*opentelemetry.instrumentation.requests.RequestsInstrumentor*  
*static method*), 76

*uninstrument\_taskgroup\_create\_task()* (*in module opentelemetry.instrumentation.asyncio*), 17

*uninstrument\_to\_thread()* (*in module opentelemetry.instrumentation.asyncio*), 17

*unpatch\_handler\_class()* (*in module opentelemetry.instrumentation.tornado*), 88

*unwrap\_connect()* (*in module opentelemetry.instrumentation.dbapi*), 28

*update\_task\_duration\_time()* (*opentelemetry.instrumentation.celery.CeleryInstrumentor*  
*method*), 26

*url* (*opentelemetry.instrumentation.httpx.RequestInfo* *attribute*), 50

*URLLib3Instrumentor* (*class in opentelemetry.instrumentation.urllib3*), 92

*URLLibInstrumentor* (*class in opentelemetry.instrumentation.urllib*), 90

## W

*wrap\_connect()* (*in module opentelemetry.instrumentation.dbapi*), 27

*wrapped\_connection()* (*opentelemetry.instrumentation.dbapi.DatabaseApiIntegration*  
*method*), 29

*wrapped\_connection()* (*opentelemetry.instrumentation.psycopg.DatabaseApiAsyncIntegration*  
*method*), 64

*wrapped\_connection()* (*opentelemetry.instrumentation.psycopg.DatabaseApiIntegration*  
*method*), 63

*wrapped\_connection()* (*opentelemetry.instrumentation.psycopg2.DatabaseApiIntegration*  
*method*), 66

*WSGIGetter* (*class in opentelemetry.instrumentation.wsgi*), 95