Getting Started With OpenTelemetry

Observability and Monitoring for Modern Applications

CHARLES MAHLER
TECHNICAL WRITER, INFLUXDATA

CONTENTS

Cloud-native and microservice architectures have risen in popularity throughout the developer community to address a number of challenges faced in modern application development. At the same time, these architectures also created new problems. One of those challenges is how to accurately monitor the performance and availability of these distributed applications. After years of confusion and complexity, the software industry has finally rallied around a project to help simplify the problem of observability.

OpenTelemetry is an open-source collection of tools, APIs, SDKs, and specifications with the purpose of standardizing how to model and collect telemetry data. OpenTelemetry came into being from the merger of the OpenCensus project (sponsored by Google) and the OpenTracing project (sponsored by CNCF) and has backing from many of the biggest tech companies in the world. Some of the companies actively contributing to the project are Microsoft, Red Hat, Google, Amazon, Shopify, Datadog, and Facebook.

The long-term goal of the OpenTelemetry project is to provide a completely standardized vendor-agnostic solution for instrumenting applications to allow for telemetry data to be sent to any backend storage platform without having to modify your underlying code base.

By creating this ecosystem of tools and standards, the engineering burden of implementing and maintaining instrumentation is reduced. Additionally, the potential for vendor lock-in due to switching costs is eliminated.

The result is that companies using OpenTelemetry will be able to use the best tools available for working with their telemetry data. This data is inherently time-series data, which has unique properties when it comes to efficient storage and analysis that are hard to manage with more traditional databases. The demand for better performance led to the development of an entire ecosystem of tools that are optimized for working with telemetry data.

OPENTELEMETRY ARCHITECTURE

OpenTelemetry consists of a variety of tools to make collecting telemetry data easier. In this section, you'll learn about the most important components of OpenTelemetry.

APIS AND SDKS

The OpenTelemetry APIs and SDKs are the programming-language-specific portion of OpenTelemetry. The APIs are used to instrument your application's code and are what generate telemetry data.

influxdata
Act in Time

Effortlessly scale your time series workloads with InfluxDB

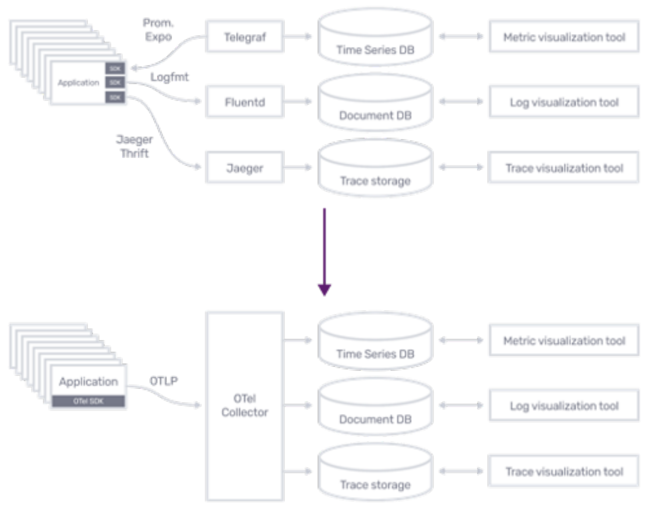
The #1 platform for building time series applications.

Try for Free

SDKs act as a bridge between the API and exporting your telemetry data. The SDKs can be used to filter or sample data before the data is exported. The SDK also allows for exporting to multiple destinations, depending on the configuration.

COLLECTOR

The OpenTelemetry Collector can be used as an alternative to the dedicated in-process exporters that are available for multiple different backends. A benefit of using the Collector is that it is completely vendor-agnostic as well as language-agnostic. This means that you can send data to the Collector from any of the language SDKs and export the data to any supported backend without modifying your application code.



Let's look at the components of the OpenTelemetry Collector:

- **Receivers** typically listen on network ports and receive telemetry data but also have the ability to pull data.
- **Processors** can be used to enrich or transform data as it is received and before it is exported. The processors work in a sequence as they are configured, so the ordering does matter.
- **Exporters** forward telemetry data to backend storage services. The Collector can have multiple exporters to send certain telemetry data to specific tools, or even replicate data to be sent to multiple locations.

The above components are assembled as pipelines, which are defined through YAML files.

Here's an example of a simple OpenTelemetry Collector pipeline configuration:

```
receivers:
  otlp:
    protocols:
      grpc:
      http:

processors:
  batch:

exporters:
  otlp:
    endpoint: otelcol:4317

extensions:
  health_check:
  pprof:
  zpages:

service:
  extensions: [health_check,pprof,zpages]
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp]
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp]
    logs:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp]
```

The OpenTelemetry Collector also gives you two options for deployment. The first option is to deploy the Collector as an Agent. In this case, the Collector runs with the application on the same host, either as a binary, sidecar using something like a service mesh, or DaemonSet if you are using Kubernetes. This is the recommended option because it allows the Collector to gather telemetry data using either push or pull methods, plus enhance data with custom tags and hardware information.

The second deployment option is known as a Gateway, where the Collector instance is run as a standalone service. Multiple Collector instances can be run, generally at the per application cluster or datacenter level. Setup for a Gateway can be more complicated but also provides some advantages like API token management and flexibility for scaling.

OPENTELEMETRY PROTOCOL

The OpenTelemetry Protocol (OTLP) is a specification that describes how to encode, transport, and deliver telemetry data between sources, intermediate nodes like collectors, and backends. The protocol uses gRPC or HTTP and defines a specific Protocol Buffer schema for sending data.

OPENTELEMETRY KEY CONCEPTS

To understand OpenTelemetry and why it exists, you need to understand observability. The goal of observability is to be able to understand what is happening at any time inside an application. This is done by collecting telemetry data, which provides insight into the performance, behavior, and overall health of the application. When properly done, observability makes it easier to find bugs and performance bottlenecks in your application, resulting in users getting a faster and more reliable experience.

INSTRUMENTATION

Instrumentation is to add the OpenTelemetry SDK to your application so that it emits observability signals (typically logs, metrics, and trace spans). In short, instrumentation is necessary to actually generate the telemetry data that is needed for observability. OpenTelemetry provides options for both automatic and manual instrumentation.

DATA SOURCES

OpenTelemetry is currently focused on three specific data sources to support, with the potential for more being added in the future. Those three data sources are traces, metrics, and logs.

TRACES

A trace is a way to track a single transaction (for example, an API request or a periodic report job) as it moves through an application, or through a network of applications. In a microservice architecture, a single transaction may touch multiple services. For each operation that a transaction touches, a segment known as a span is created. The span records interesting characteristics of the operation such as duration, errors encountered, and a reference to the parent span (representing the operation that called the current operation). By tracing the progress of a single transaction through our service architecture, we can find the service or operation that is the root cause for slow or failed requests.

As mentioned, traces are often used to debug and optimize distributed software operations. Tracing is also used to optimize manufacturing processes, answering questions like: "Why does widget production slow on Tuesdays?" And to optimize logistics pipelines, tracing can answer questions like: "Will raw materials arrive in time for batch 23 to begin on Friday?"

Traces are then exported from OpenTelemetry to a backend like Zipkin or Jaeger for analysis and visualization.

METRICS

Metrics are measurements from a service created at a specific moment in time. Metrics are typically aggregated in the emitting application, collected at fixed time intervals, ranging from every second to once per day. The goal of collecting metrics data is to give yourself a frame of reference for how your application is behaving. Based on past data, you can then set alerts or take automated actions if certain metrics exceed acceptable thresholds.

Some examples of metrics:

- Request success and error rates, e.g., 42 requests per second
- Request latency, e.g., count per histogram bucket
- Bandwidth utilization, e.g., 1.2Mb/s used of 10Mb/s capacity, or 12 percent
- Fuel level, e.g., 3.2 gallons

LOGS

Logs are time-stamped records containing metadata. The data can be structured or unstructured and serves as a record of an event that took place inside an application. Logs are often used by teams when trying to find what changes resulted in an error occurring. In OpenTelemetry, logs can be independent or attached to spans and are often used for determining the root cause of issues.

GETTING STARTED WITH OPENTELEMETRY TRACING

In this section, you will learn about the OpenTelemetry specifications for the tracing API and SDK. The specification defines the naming conventions and functionality for OpenTelemetry implementations, regardless of the programming language, so that each language library is consistent.

Currently, only the Tracing specification is considered stable, due to the work already done by the OpenTracing project. The Metrics specification is expected to have a stable V1.0 release near the end of 2021. The Log specification is still considered to be in the draft stage and may get a V1.0 release sometime in 2022. Because there may be breaking changes in those specifications, this Refcard won't go into detail on the Metrics and Logs specifications for OpenTelemetry.

TRACING SDK SPECIFICATION

Here are some of the most important functions defined by the OpenTelemetry client SDKs:

- **Tracer Provider**: This is the SDK method responsible for creating trace objects. Tracer Provider is also used to configure span processors, sampling, unique ID generation, and span limits.
- **Sampling**: The SDK provides the ability to sample the number of traces being exported to the backend. In some situations, you may not want to collect every single piece of telemetry data being generated due to performance or cost reasons.

- **Span Processors**: Span processors are responsible for transforming the telemetry data into a format that can be exported to a backend system. A pipeline of multiple span processors that run in the order they are registered with the Trace provider is expected to be provided by any language SDK.

TRACING API SPECIFICATION

Here are some of the most important functions defined by the OpenTelemetry client API libraries

- **Trace**: The trace object is responsible for creating span objects. Traces cannot be created without having a Tracer Provider from the SDK.
- **Span**: Spans represent a single operation inside a trace. Spans can be nested to form a trace tree, which has a root span that covers the entire operation. Spans should have a name that makes it easy to recognize what is happening. Spans also require unique IDs and a trace ID so they can be connected to the other spans that comprise a trace.
- **SpanKind**: `SpanKind` allows developers to describe relationships between spans for things like synchronous calls to other services. By setting this information, it can help show where latency is happening.
- **SpanContext**: In some cases, you might have nested spans that you want to set a different context on compared to the overall trace.

CONTEXT API

The Context API is needed for tracking requests as they move across different services in your architecture. This includes information like the trace ID, span ID, and state for the current span. OpenTelemetry handles this under the hood by injecting context information into request headers and then extracts that information in the service that receives the request, a process known as "context propagation."

In most cases, you won't need to think about the context API much, but you do have the option of creating user-defined properties that can be attached to the context. Some examples would be things like the user ID for the request, server names, or datacenter region. This additional data can then be used later for analysis.

COLLECTING TRACES IN A PYTHON APPLICATION USING OPENTELEMETRY

In this section, you will learn how to collect traces from your application using OpenTelemetry.

BASIC TRACE

It's time to actually get into a bit of code and show how you can create a trace and spans in Python. While the syntax will be slightly different for other languages, because of the standardized specification, most of the naming conventions will carry over to other languages.

First, you need to install the Python OpenTelemetry SDK and API using Pip and the following command:

```
pip install opentelemetry-api
pip install opentelemetry-sdk
```

Now, you can import the methods and classes you need from the API and SDK to instrument your Python script:

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import
BatchSpanProcessor, ConsoleSpanExporter

trace.set_tracer_provider(TracerProvider())
trace.get_tracer_provider().add_span_processor(
    BatchSpanProcessor(ConsoleSpanExporter())
)
tracer = trace.get_tracer(__name__)
with tracer.start_as_current_span("span name"):
    print("doing some work here")
```

So what exactly is happening here? Let's go over some of the methods being used and what the code is doing step by step:

1. The Tracer Provider from the SDK is instantiated, which is required to create a trace because it holds the configuration for all the trace objects.
2. The trace object then uses that to set the span processor as `BatchSpanProcessor`, which is one type of implementation of the `SpanProcessor` specification from the SDK. As you might expect from the name, this processor batches together spans before sending them to an exporter.
3. `ConsoleSpanExporter` is used so that the output is sent to your terminal; this is useful for testing situations or if you don't want to set up somewhere to export your data yet.
4. The tracer object is assigned to the tracer variable and a span is created in the final two lines of code.

The output to the terminal looks like this:

```
{
    "name": "span name",
    "context": {
        "trace_id":
"0x84cda3e4bcd162c623c3fba49624c34e",
        "span_id": "0xc9324061456b7d30",
        "trace_state": "[]"
    },
    "kind": "SpanKind.INTERNAL",
    "parent_id": null,
    "start_time": "2021-11-19T16:31:13.872201Z",
    "end_time": "2021-11-19T16:31:13.872304Z",
    "status": {
        "status_code": "UNSET"
    },
    "attributes": {},
    "events": [],
    "links": [],
    "resource": {
        "telemetry.sdk.language": "python",
        "telemetry.sdk.name": "opentelemetry",
        "telemetry.sdk.version": "1.7.1",
        "service.name": "unknown_service"
    }
}
```

AUTOMATIC VS. MANUAL INSTRUMENTATION IN A FLASK APP

For a visual example of automatic vs. manual instrumentation, let's compare how the code of a Flask app looks when using both options.

The manually instrumented Flask app code looks like this:

```
from flask import Flask, request
from opentelemetry import trace
from opentelemetry.instrumentation.wsgi import
collect_request_attributes
from opentelemetry.propagate import extract
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    BatchSpanProcessor,
    ConsoleSpanExporter,
)

app = Flask(__name__)

trace.set_tracer_provider(TracerProvider())
tracer = trace.get_tracer_provider().get_tracer(__
name__)

trace.get_tracer_provider().add_span_processor(
    BatchSpanProcessor(ConsoleSpanExporter())
)

@app.route("/home")
def server_request():
    with tracer.start_as_current_span(
        "home",
        context=extract(request.headers),
        kind=trace.SpanKind.SERVER,
        attributes=collect_request_
attributes(request.environ),
    ):
        print(request.args.get("param"))
        return "home page"

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

The Flask app code using auto-instrumentation looks like this:

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

app = Flask(__name__)

FlaskInstrumentor().instrument_app(app)

@app.route("/home")
def server_request():
    print(request.args.get("param"))
    return "home page"

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

The above auto-instrumentation is provided by the OpenTelemetry Python library's contributor section package for Flask. Many other popular Python libraries and modules are supported like Django and popular databases like MongoDB or PostgreSQL.

The same applies to other languages where many of the most popular frameworks and libraries already have built-in support for OpenTelemetry. This will only increase over time due to the mutual benefit for open-source projects to add OpenTelemetry instrumentation and interoperability between projects.

From the code above, you can see how — in a large-scale application — using auto-instrumentation would save you from having to write a ton of repetitive code. On the other hand, you do lose some control, so in certain situations, it might make sense to customize your instrumentation.

MIGRATING LEGACY APPLICATIONS TO OPENTELEMETRY

For a greenfield project, using OpenTelemetry is an easy choice, but what about legacy applications? Refactoring is an option but might not be worth the time and has its own risks. In this case, an option could be to leave things as they are and instead rely on a tool like Telegraf, which can convert the telemetry data being generated by your application now from its current format and output it to OpenTelemetry.

WORKING WITH OPENTELMETRY DATA

So you are now set up and collecting your telemetry data using OpenTelemetry, but what do you do with it? There are a number of commercial tools available for handling this data, but here the focus will be on open-source projects that can be used to store and analyze your data.

TRACES

For tracing data, the two leading projects are Zipkin and Jaeger. Zipkin is an older project that was released by Twitter. Jaeger is newer but has been declared fully graduated from the Cloud Native Computing Foundation, joining projects like Kubernetes and Prometheus. This is a signal that, in the long term, Jaeger may be a stronger choice. Both provide similar features for visualizing and analyzing your traces, and both have OpenTelemetry exporter support. Zipkin and Jaeger both support multiple backends for the actual storage of your data, so you can choose whatever database you decide works best for your use case.

METRICS

When it comes to storing and querying metrics data, there are currently two leaders, InfluxDB and Prometheus. Both are open-source projects that specialize in handling time-series workloads and provide query languages, visualization tools, alerting features, and integrations with the majority of popular tools used for metric monitoring.

LOGS

What tool you use for logging will depend on how you want to work with them. One option is to essentially convert them into structured metrics and use a time-series database. Telegraf, Logstash, or FluentD are commonly used for processing raw logs before storage. For working with semi-structured or unstructured logs, ElasticSearch is a common choice.

The main takeaway from this section is to realize that you have a number of options to choose from thanks to using OpenTelemetry. Choosing which tools you utilize is a tradeoff between things like convenience, functionality, cost, and usability.

CONCLUSION

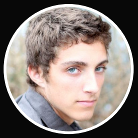
Properly implemented observability is a massive advantage for any software company. By having deeper insight into what is happening at each moment in your application, you are able to find bugs faster, make your app run more efficiently, and make your users happier. Trying to run a distributed application without observability is the equivalent of flying blind.

In the past, it may have been a challenge to instrument your app, but with the rise of OpenTelemetry, you can get all the benefits of observability without much hassle. Valuable engineering time doesn't have to be spent on implementation because OpenTelemetry — and the community surrounding it — have created auto-instrumentation packages for many of the most commonly used frameworks and libraries.

What's even better is that you no longer have to worry about being locked into a specific vendor or worry about having to refactor code if you migrate, because with the standardized exporters, you can simply change a bit of configuration and move to whatever backend tool works best for your use case.

WRITTEN BY CHARLES MAHLER,
TECHNICAL WRITER, INFLUXDATA

Charles Mahler is currently a Technical Writer at InfluxData. He has experience in full-stack software development and digital marketing.



Devada, Inc.
600 Park Offices Drive
Suite 300
Research Triangle Park, NC 27709

888.678.0399 | 919.678.0300