OpenTelemetry Workshop

Instrumenting, collecting and exporting telemetry data

Version: 1.0.0

Table of Contents

1 Observability Concepts
2 OpenTelemetry
3 Instrumentation
4 OTel Collector
5 OTel Backends

Licence

Agenda

- OpenTelemetry (OTel) theory
- Using OTel in an application
- The OTel Collector
- Sending data to telemetry backend

1 Observability Concepts

Monitoring and Observability

Monitoring is the *practice* of collecting, aggregating and processing data about a system.

- Black-box monitoring. Testing externally visible behavior as users would see it
- White-box monitoring. Observing the data/telemetry exposed by the system's internals

Observability is a *measure* of how well we can understand a system (measures the *ability* to *observe* the system).

The data generally comes in the form of traces, metrics, and logs.

Metrics

Metrics are numerical values that can be used to determine a service or component's behavior. Examples:

```
service_open_connections
filesystem_avail_bytes
network_receive_bytes_total
config_last_reload_success
```

Metrics can originate from a variety of sources (hosts, services, external sources, etc.).

Metrics can be used to observe and compare behavior over time.

Logs

Logs are text-based events that protocol activities over time.

```
Jan 10 14:30:01 pc CRON[121602]: pam_unix(cron:session):
  session opened for user root(uid=0) by (uid=0)
Jan 10 14:30:01 pc CRON[121602]: pam_unix(cron:session):
  session closed for user root
```

They come in a variety of formats and are either structured (i.e. JSON) or unstructured.

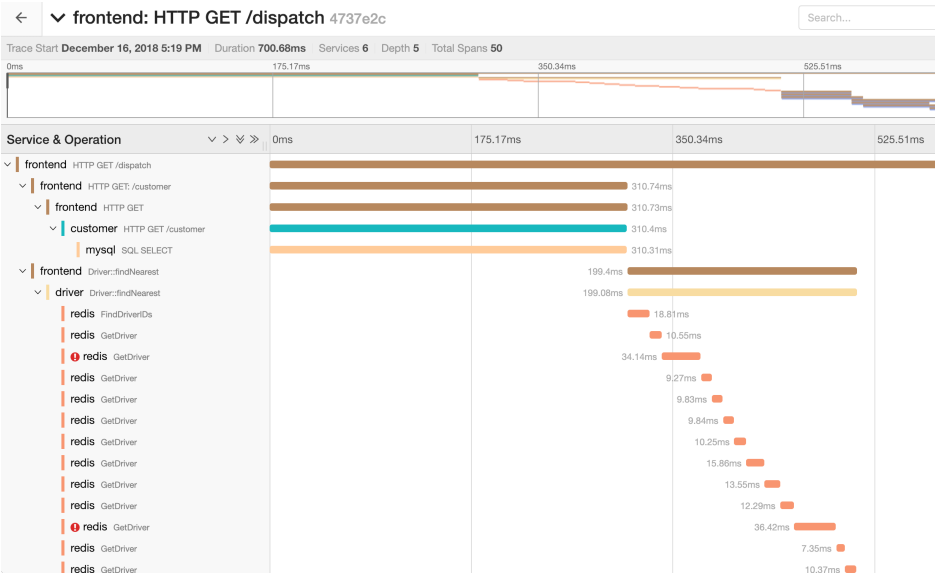
Structured logs aim to create machine-readable data.

Traces

Traces are records of requests flowing through different parts of an application, which can record the duration and data for each subsystem.

Traces represent the execution path of a request, each span in a trace represents a unit of work during this journey (e.g. API call or database query).

Jaeger Trace Example

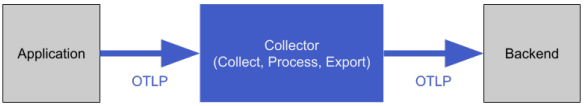


2 OpenTelemetry

What is OpenTelemetry?

OpenTelemetry (OTel) is a collection of APIs, SDKs, and tools.

Designed to instrument, generate, collect, and export telemetry data aka. signals (metrics, logs, and traces).



OpenTelemetry tries to standardize software telemetry data.

What is OpenTelemetry not?

OTel is not an observability product.

OTel is not a backend for the data.

OTel is not a visualization tool.

The storage and visualization of telemetry is intentionally left to other tools.

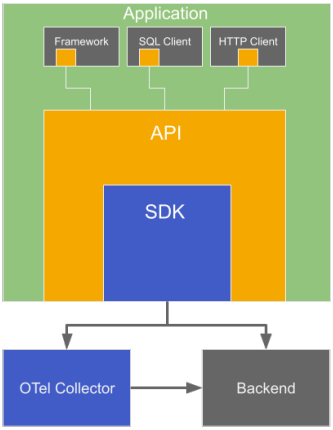
Specifications and Components

OpenTelemetry is developed on a signal by signal basis. Each of which consists of four components:

- API, technical specifications and documentation
- SDK, tools for a specific platform
- Collector
- OpenTelemetry Protocol (OTLP)

Components follow a development life cycle: Draft, Experimental, Stable, Deprecated, Removed.

API, SDK and the Collector



More on the Collector later. First we focus on the API and SDK.

3 Instrumentation

3.1 Metrics

The components that we use in our code:

- A `Meter Provider` is a factory for Meters
- A `Meter` creates `Metric Instruments`, these instruments capture the actual measurements
- `Metric Exporters` send metric data to a consumer

Manual Metric Instrumentation

Dependencies for instrumentation:

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

Simple example for an instrument:

```
mymeter = metrics.get_meter(name="dice.meter")

roll_counter = mymeter.create_counter(
    "dice.rolls",
    unit="count",
    description="The number of rolls by roll value"
)
```

See: Lab 1.1 and Lab 1.2

Synchronous Instruments

These are invoked together with operations they are measuring.

- `Counter`, supports non-negative increments
- `UpDownCounter`, supports increments and decrements
- `Histogram` can be used to report arbitrary values that are likely to be statistically meaningful

See: Lab 1.3

Asynchronous Instruments

These are periodically invoke a callback function to collect measurements.

- `ObservableCounter`, reports monotonically increasing values when the instrument is being observed
- `ObservableUpDownCounter`, additive values when the instrument is being observed (also negative)
- `Asynchronous Gauge`, reports non-additive values when the instrument is being observed (e.g. a temperature)

See: Lab 1.4

Views and Aggregations

Aggregations are the means by which metric events are processed. Example: SumAggregation, LastValueAggregation, ExplicitBucketHistogramAggregation

The OpenTelemetry API provides a default aggregation for each instrument, which can be overridden by a View.

```
view = View(
  instrument_name="myhist*",
  aggregation=ExplicitBucketHistogramAggregation(boundaries=[1,2,3,4])
)

provider = MeterProvider(metric_readers=[reader], views=[view])
```

Exporters

Exporters are responsible for sending the collected data elsewhere:

- ConsoleMetricExporter: write debug messages to the console
- OTLPMetricExporter: push metrics to any device that understands the OpenTelemetry protocol
- Prometheus Metric Exporter: pull-based exporter that Prometheus clients can scrape
- Prometheus Remote Write Exporter: push-based exporter for the Prometheus remote write protocol

```
exporter = OTLPMetricExporter(insecure=True)
reader = PeriodicExportingMetricReader(exporter)
provider = MeterProvider(metric_readers=[reader])
```

More on OTLP later.

3.2 Traces

The components that we use in our code:

- A `Trace Provider` is a factory for Tracers
- A `Tracer` creates `Spans` containing more information about what is happening for a given operation
- `Trace Exporters` send trace data to a consumer

Manual Trace Instrumentation

Simple example for a Python tracer:

```
tracer = trace.get_tracer("mytracer")

with tracer.start_as_current_span("foo"):
    do_stuff()
```

Similar in Golang:

```
ctx, span := tracer.Start(r.Context(), "roll")
defer span.End()

roll := 1 + rand.Intn(6)

rollValueAttr := attribute.Int("roll.value", roll)
span.SetAttributes(rollValueAttr)
```

See: Lab 2.1

Span

A span represents a unit of work or operation. Spans are the building blocks of Traces.

- Name
- Parent span ID (empty for root spans)
- Span Links, associate one span with one or more spans, implying a causal relationship.
- Span Context
- Span Events, like a structured log message
- Span Status, possible values: Unset, Error, Ok
- Attributes
- Start and End Timestamps

Span Events

Events are like structured log message (or annotation) on a Span, typically used to denote a singular point in time during the Span's duration.

For example, consider two scenarios in a browser:

1. Tracking a page load
2. Denoting when a page becomes interactive

The first scenario has a clear start and end, thus a Span.

The second is a singular point in time, thus a Span Event.

See: Lab 2.2 and Lab 2.3

3.3 Logs

OpenTelemetry does not define an API or SDK to create logs, only a data model.

OpenTelemetry's support for logs is designed to be compatible with what you already have.

The `Log Bridge API` can be used by logging library authors to integrate with OpenTelemetry.

It should not be called by you the user.

Auto Instrumentation

Depending on the logging library you are using:

```
pip install opentelemetry-exporter-{exporter}
pip install opentelemetry-instrumentation-{instrumentation}

# Example:
pip install opentelemetry-instrumentation-logging
```

See: https://opentelemetry.io/ecosystem/registry/

Manual Log Instrumentation

Just an example, we don't need to do this manually:

```
import logging
from opentelemetry.instrumentation.logging import (
DEFAULT_LOGGING_FORMAT, LoggingInstrumentor )

LoggingInstrumentor().instrument(
  set_logging_format=True,
  log_level=logging.DEBUG
)

logging.debug('Debug me')
logging.info('Info only')
logging.warning('Dangerzone!')
```

More on zero-code instrumentation later.

See: Lab 3.1

Log Record

The data model:

- Resource: describes the source of the log
- Timestamp: time when the event occurred
- SeverityText: the severity text (also known as log level)
- SeverityNumber: numerical value of the severity
- Body: the body of the log record
- Attributes: additional information about the event

Existing log formats should be mapped to this data model.

See: https://opentelemetry.io/docs/specs/otel/logs/data-model/

Zero-Code Instrumentation

The `opentelemetry-distro` package installs the API, SDK, and the opentelemetry-bootstrap and opentelemetry-instrument tools:

```
pip install opentelemetry-distro opentelemetry-exporter-otlp
opentelemetry-bootstrap -a install

opentelemetry-instrument --service_name myapp \
--traces_exporter console \
--logs_exporter console \
--metrics_exporter console \
python myapp.py
```

Zero-code instrumentation depends on the programming language.

See: Lab 3.2

Profiling

The OTel project is working on a new signal: Profiles.

Profiling is a method to dynamically inspect the behavior and performance of application code at runtime.

Tracing focuses on the macro level (request flow).

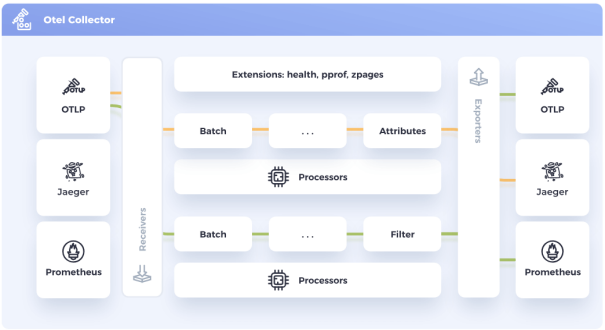
Profiling focuses on the micro level: CPU usage, memory consumption and execution time hot spots.

See: https://opentelemetry.io/blog/2024/state-profiling/

4 OTel Collector

The Collector

The OpenTelemetry Collector is a vendor-agnostic proxy that can receive, process, and export telemetry data.



It also supports processing and filtering telemetry data before it gets exported.

When to use the collector

For most language specific instrumentation libraries you have exporters for popular backends and OTLP. Why use another tool?

The Collector allows the service to offload data quickly and the collector can take care of the rest.

Handling retries, batching, encryption or even filtering sensitive data.

Installation

There are various ways to install the Collector:

- Packages from the Collector Release repository
- Container Images on `docker.io/otel/opentelemetry-collector-contrib`
- OpenTelemetry Helm Charts or the Kubernetes Operator

https://github.com/open-telemetry/opentelemetry-collector-releases

Configuration

The structure of any Collector configuration consists of:

- Receivers
- Processors
- Exporters
- Connectors

Configuration Structure

```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
processors:
  batch:

exporters:
  debug:
    verbosity: detailed

service:
  pipelines:
    logs:
      receivers: [otlp]
      processors: [batch]
      exporters: [debug]
```

See: Lab 4.1

Integration

`opentelemetry-instrument` is a tool that automatically instruments code via available instrumentation libraries:

```
pip install opentelemetry-distro opentelemetry-exporter-otlp
opentelemetry-bootstrap -a install

export OTEL_PYTHON_LOGGING_AUTO_INSTRUMENTATION_ENABLED=true

opentelemetry-instrument --traces_exporter=console \
--metrics_exporter=console \
--logs_exporter=console --service_name=dice flask run
```

Hint: Differs depending on the language.
https://opentelemetry.io/docs/concepts/instrumentation/zero-code/

See: Lab 4.2

Receivers

Receivers typically listen on a network port and receive telemetry data.

They can also actively obtain data, like scrapers.

Usually one receiver is configured to send received data to one pipeline.

```
receivers:
  filelog:
    include: [ /var/log/myservice/*.json ]
    operators:
      - type: json_parser
        timestamp:
          parse_from: attributes.time
          layout: '%Y-%m-%d %H:%M:%S'
```

See: https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/receiver

Processor

Processors take the data collected by receivers and modify or transform it before sending it to the exporters.

This might include filtering, dropping, renaming, or recalculating telemetry.

```
processors:
  logs:
    log_record:
      - 'IsMatch(body, ".*password.*")'
  memory_limiter:
    limit_mib: 4000
  batch:
service:
  pipelines:
    logs:
      receivers: [otlp]
      processors: [memory_limiter, logs, batch]
      exporters: [otlp]
```

See: https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/processor

Distributions

The OpenTelemetry project provides compiled distributions of the collector.

But you easily build your own with the `ocb` tool (OpenTelemetry Collector Builder).

When you need custom functionality like authenticator extensions, receivers, processors, exporters or connectors.

https://opentelemetry.io/docs/collector/distributions/

OTLP

The OpenTelemetry Protocol (OTLP) defines the encoding of telemetry data and the protocol used to exchange data between the client and the server.

This specification defines how OTLP is implemented over gRPC and HTTP.

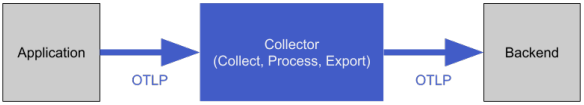
More and more vendors offer data ingestion via OTLP natively.

See: https://opentelemetry.io/docs/specs/otlp/

5 OTel Backends

OpenTelemetry Consumers

There are many solutions that consume OTLP natively. Some are open source some are not.



How data is stored, aggregated, visualized and other features depend on the vendor:

https://opentelemetry.io/ecosystem/vendors/

See: Lab 5.1

Collector Deployment Patterns

The OpenTelemetry Collector consists of a single binary which you can use in different ways, for different use cases.

See: https://github.com/jpkrohling/opentelemetry-collector-deployment-patterns/

The OTel Team also runs many performance benchmarks on the components:

https://opentelemetry.io/blog/2023/perf-testing/

Conclusion

The Good, the Bad and the Ugly

- Is it maybe just the lowest common denominator for telemetry?
- What is it? Is it a semantic standard? Is a protocol? It is a facade? It is a library?
- There are so many concepts you have to master
- The OTel standard is mature, but the open source implementation might be somewhat bulky or still work-in-progress

OpenTelemetry Certified Associate

The CNCF and Linux Foundation Education just launched the OpenTelemetry Certified Associate (OTCA) certification (Posted on November 15, 2024):

https://www.cncf.io/blog/2024/11/15/gain-insights-into-cloud-native-applications-with-the-opentelemetry-certified-associate-otca/