Getting Started With OpenTelemetry

Observability and Monitoring for Modern Applications

JOANA CARVALHO
PERFORMANCE ENGINEER, POSTMAN

CONTENTS

The mass migration to the cloud brings new architecture paradigms and challenges. The many components involved make monitoring and correlating signals from all elements difficult. OpenTelemetry comes to aid with a vendor-agnostic telemetry specification that allows developers in any stack to gather telemetry data. OpenTelemetry aims to be the standard for implementing and enabling effective observability. This Refcard introduces its core architecture components, key concepts and features, and how to set them up for tracing and exporting telemetry data.

OVERVIEW OF OPENTELEMETRY

Observability, or "o11y" for friends, empowers teams to ask questions about their system and business and receive clear answers driven by the signals collected. Telemetry signals — logs, metrics, traces, events, and metadata — work together to correlate individual systems' health with the business' overall health, giving developers and operations teams a greater understanding.

A common misconception is that observability replaces monitoring; quite the contrary — observability amplifies its potential.

- **Monitoring** is a process that collects and analyzes telemetry data for specific metrics and acts according to the objectives defined (e.g., alerts, notifies).
- **Observability** is the ability to ask questions about the holistic state of a system through the signals it generates.

Monitoring takes you a long way, but if the telemetry is ineffective, insufficient, or inaccurate, it will not take you to the level you want — observability.

In 2016, OpenTracing became a Cloud Native Computing Foundation (CNCF) project, and in 2018, Google open sourced OpenCensus. These standards complemented each other, aiming to make observability easy and widely adopted. However, having the community divided to maintain both projects would lead to poor adoption, contribution, and support. To avoid this, in 2019, it was announced that both projects would converge into OpenTelemetry and join the CNCF.

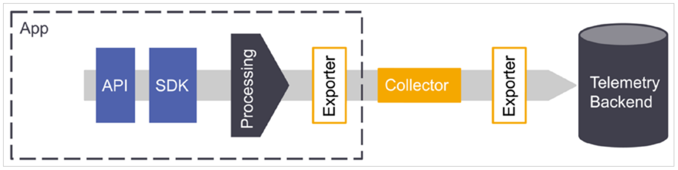
OpenTelemetry (OTel) quickly became the de facto standard for flexible full-stack observability in cloud-native applications. Its vendor-neutral standards, libraries, integrations, APIs, and software development kits (SDKs) give developers an independent specification for telemetry.

As with any open-source software, the maturity level of each component will depend on the language and the interest taken by that particular community — the more popular the language or the framework, the more support and maturity it'll reach.

OPENTELEMETRY ARCHITECTURE

OpenTelemetry provides a library framework that receives, processes, and exports telemetry, which requires a back end to receive and store the data. In the following diagram, you can see how all elements work together, and we'll go into more detail about each one.

Figure 1: OpenTelemetry pipeline



Source: Schema based on "OpenTelemetry: beyond getting started"

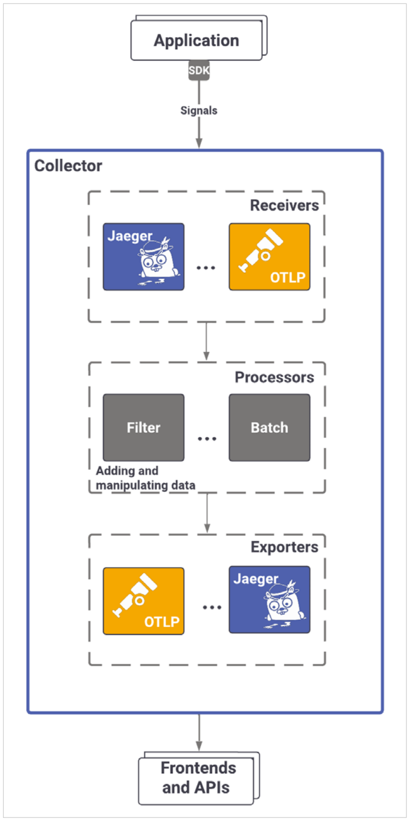
APIS AND SDKS

OpenTelemetry APIs define how applications speak to one another and are used to instrument an application or service. They are generally available for developers to use across popular programming languages (e.g., Ruby, Java, Python). Because they are part of the OpenTelemetry standard, they will work with any OpenTelemetry-compatible back-end system, eliminating the need to re-instrument in the future. The SDK is also language specific, providing the bridge between APIs and the exporter. It can sample traces and aggregate metrics.

COLLECTOR

The OpenTelemetry Collector is like a bakery: Regardless of how the raw ingredients are processed, you can still shape your bread in whatever way you fancy. This means you don't need to alter your code to send data into whatever back end you use for storage and visualization.

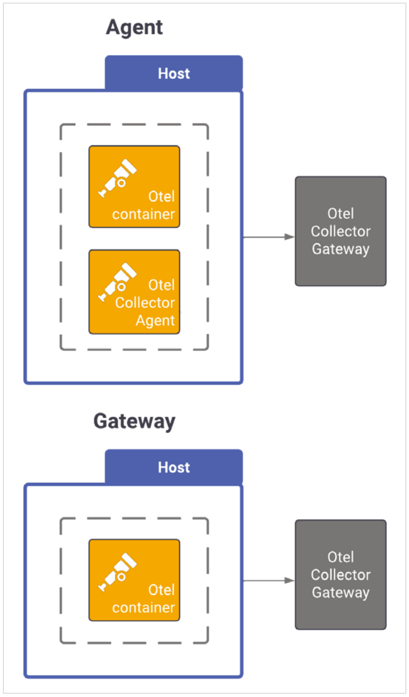
Figure 2: Inside the OTel Collector pipeline



The Collector's job is to process, filter, aggregate, and batch telemetry, giving developers greater flexibility for receiving, shaping, and sending data to multiple back ends. It works with two primary deployment models:

- As an **Agent** that lives within the application or in the same host as the application, acting as a source of data for the host (by default, OpenTelemetry assumes a local collector is available)
- As a **Gateway** working as a data pipeline that receives, exports, and processes telemetry

Figure 3: Collector Agent and Gateway setup



The Collector consists of three components: receivers, processors, and exporters. *Receivers* (e.g., Jaeger, Prometheus) are in charge of pushing or pulling the applications' signals by listening for calls on particular ports on the Collector. They work with both gRPC and HTTP protocols. A complete list of receivers for specific scenarios or frameworks can be found on GitHub.

Processors sit between receivers and exporters; they enable us to shape the data by filtering, formatting, and enriching it before it goes through the exporter to a back end. Common use cases include data sanitization to remove sensitive or private information, exporting metrics from spans, or deciding which signals are saved to the back end. There are many supported processors available, or you can develop your own. They work sequentially, so the configuration order is important. Although processors are not required, some might be recommended based on the data source.

Exporters can push or pull data into one or multiple configured back ends or destinations (e.g., Kafka, OTLP). They work by transforming the data into a different format if needed and sending it to the endpoint defined. An exporter creates a layer of separation between instrumentation and the back-end configuration so users can switch back ends without re-instrumenting the code. It supports either the HTTP or gRPC protocol. Popular exporters include Jaeger, Prometheus, and Zipkin, along with a vast list of other options.

To configure the three Collector components, we must specify the parts that will compose our pipeline. We can do so by writing the configuration in a YAML format and stating what elements will be configured in the Collector using the service section, as shown in the example below:

```
# otel-collector-config.yml
receivers:
 otlp:
   protocols:
     http:
       endpoint: 0.0.0.0:4318
     grpc:
       endpoint: 0.0.0.0:4317
processors:
 batch:
   timeout: 1s
exporters:
 logging:
   loglevel: info
 extensions:
 health_check:
 pprof:
   endpoint: :1888
 zpages:
   endpoint: :55679
service:
 extensions: [pprof, zpages, health_check]
 pipelines:
   traces:
     receivers: [otlp]
     processors: [batch]
     exporters: [logging]
```

As you can see on the configuration file above, we set the OTLP receiver to add HTTP and gRPC endpoints in the collector. To process our data, we use a batch processor that will compress and segment it; it's configurable for batching by time and size. Using the batch processor is highly recommended, as it reduces the number of outgoing connections.

We also define two exporters: logging that will print into the console and Jaeger, where we'll send the traces. The service section is where we set up how all the previous elements come together in the pipeline. We only refer to traces in this example, but we could also have metrics or logs.

OPENTELEMETRY PROTOCOL

The OpenTelemetry Protocol (OTLP) is one of the reasons for OpenTelemetry's success. It's an agnostic protocol specification that defines the encoding for data and the transport protocol for sending traces, metrics, and logs. It can send data from the SDK to the Collector and from the Collector to the chosen back end. Using the Collector elements, we can abstract from third-party frameworks by configuring the proper receiver.

OPENTELEMETRY KEY CONCEPTS

All observability journeys must begin with instrumenting an app to emit signals from services as they execute. OpenTelemetry gives you several components that'll help you add proper instrumentation to services and have each operation execution result in one or multiple spans, metrics, or logs.

INSTRUMENTATION

There are mainly two ways to instrument apps using OpenTelemetry: manual and auto-instrumentation. These become available by adding the OpenTelemetry SDK to your project. Auto-instrumentation makes it possible to collect application-level telemetry without manual changes to the code — it allows tracing a transaction's path as it navigates different components, including:

- Application frameworks
- Communication protocols
- Data stores

Manual instrumentation lets you decide how and where to add observability code to your project. Four instrumentation libraries are available:

- **Core** contains all language instrumentation libraries available.
- **Instrumentation** adds to the Core library by adding extra language-specific capabilities.
- **Contrib** includes additional helpful libraries and standalone utilities that don't fit the scope of the previous two.
- **Distribution** adds vendor-specific customization.

Not all languages will separate their instrumentation libraries as above. Some can live within the same repository, while others are split into additional ones.

LANGUAGES AND SUPPORT STATUS

OpenTelemetry is a collection of tools, APIs, and SDKs available in multiple languages. Several dedicated groups are working to maintain all these components and their language implementations. Some are dedicated to a vertical, working on signals, while others support the implementations and extensions for languages.

Development speed depends on multiple factors like team size and availability, which lead to projects being in various stages of maturity — Draft, Experimental, Stable, and Deprecated. Stable means a project is production-ready and is receiving long-term support. Table 1 on the next page shows the current maturity status of OpenTelemetry elements for some of the languages it supports.

Table 1: OpenTelemetry code instrumentation state

LANGUAGE	TRACING	METRICS	LOGGING
Java	Stable	Stable	Experimental
.NET	Stable	Stable	ILogger: Stable OTLP log exporter: Experimental
Go	Stable	Experimental	Not yet implemented
JS	Stable	Development	Roadmap
Python	Stable	Experimental	Experimental

TELEMETRY SOURCES

Telemetry data, or signals, are any output collected from the system, and when analyzed together, this output provides a view of the relationships and dependencies of the distributed system. Currently, OpenTelemetry supports three categories of telemetry: logs, traces, and metrics. Hopefully, it will extend support for more signals, like profiling or user data.

METRICS

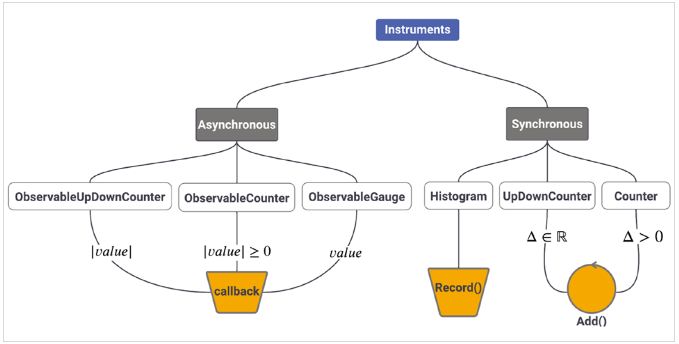
A metric is a numerical representation of a value calculated or aggregated for a service captured at runtime (e.g., size of a message broker, number of errors per second, process memory utilization, error rate). The moment one of these measurements is captured is known as a metric event — it comprises not only the measurement but also the time of capture and associated metadata.

Application and request metrics are essential indicators of availability and performance. The OpenTelemetry Metrics API processes the raw measurements, summarizing them to give developers visibility into their services' operational metrics.

In the Metrics API, you have six available instruments that are associated with a specific meter at creation time. These can be *synchronous* or *asynchronous*; synchronous instruments are invoked inline with the application code execution, while asynchronous instruments allow the user to register a callback function responsible for reporting the measurements.

In this diagram, you can see the operations that the instruments call, as well as the type of value that is captured:

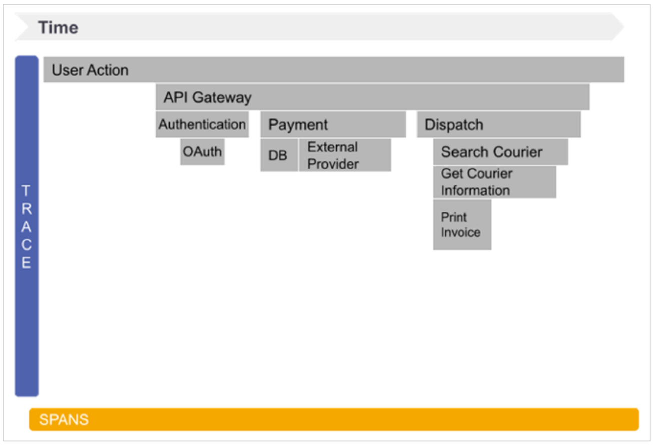
Figure 4: Metrics API available Instruments



TRACES

A trace represents the flow of a single transaction or request as it goes through the system. They provide a holistic view of the chain of events triggered by requests and are defined by a tree of nested spans — one for each unit of work they represent and a parent span. In .NET, you might use the OTel Tracing API or the .NET System.Diagnostics.Activity API that is also supported. Be aware that in the .NET library, the terms used differ from the Tracing API.

Figure 5: Representation of a trace with a tree of spans



To better understand the objects and actions you will add to your code, let's look at the main concepts that the OpenTelemetry SDK and API will implement for traces:

- `TracerProvider` is a factory for `Tracers`; it's initialized once and lives for the duration of the application's lifecycle. It's the first step in tracing with OpenTelemetry. In some SDKs, a global Tracer Provider already exists (.NET).
- `Tracer` creates spans containing supplementary information about what is happening for a given operation. It is created from Tracer Providers, and in some SDKs, a global Tracer already exists (Python, .NET).
- Trace Exporter sends traces to a back end; it can be standard output like the OpenTelemetry Collector or any open-source or vendor back end of your choice.
- Trace Context

This is an example of a trace with three spans; the information inside the spans is shown in the next section:

```
{
 "data": [
   {
     "traceID": "81289be65e00618d84366dfe2f7fc1a2",
     "spans": [
       {
         "traceID":
"81289be65e00618d84366dfe2f7fc1a2",
         "spanID": "e03e8cca690f81c1",
         "operationName": "read_json_from_file",
```

CODE CONTINUES ON NEXT PAGE

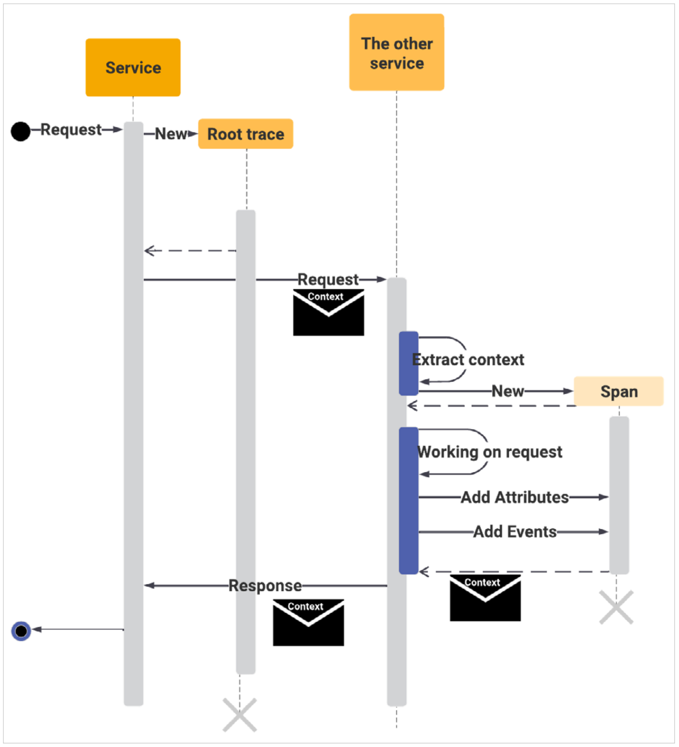
```
        }
        // ...
        {
          "traceID":
"81289be65e00618d84366dfe2f7fc1a2",
          "spanID": "75473187e1bc7579",
          "operationName": "word-by-language"
          // ...
        },
        {
          "traceID":
"81289be65e00618d84366dfe2f7fc1a2",
          "spanID": "45c0d587ebdddf60",
          "operationName": "/words"
          // ...
        }
      ],
      "processes": {
        "p1": {
          "serviceName": "untranslatable-python",
          "tags": []
        }
      },
      "warnings": null
    }
  ],
  "total": 0,
  "limit": 0,
  "offset": 0,
  "errors": null
}
```

SPANS

In OpenTelemetry, a span includes the following information:

- Name
- Start and End Timestamps
- Span Context is an object that can't be changed after creation, containing:
  - Its own ID
  - Trace ID – a unique 16-byte array that identifies the trace that the span is part of, and all spans contained in that trace share this ID.
  - Trace Flags – present in all traces and, through binary encoded data, provide more details on the trace.
  - Trace State – a key-value list carrying vendor-specific trace information, so multiple tracing systems can participate in a trace.
- Span Attributes are key-value pairs added to a span to help analyze the trace data. The extra information will help you better understand and search for specific traces.
- Span Events are typically used to mark a singular point in time during the span's duration, similar to adding an annotation on a span.
- Span Links associate one span with one or more, implying some relationship; they are optional but an excellent way of associating trace spans.
- Span Status

Figure 6: Span lifecycle



To help you contextualize and add extra information about what happens during the work that's tracked by a span, OpenTelemetry provides Attributes and Span Events. This is an example of a span with two events:

```
{
 "traceID": "81289be65e00618d84366dfe2f7fc1a2",
 "spanID": "e03e8cca690f81c1",
 "operationName": "read_json_from_file",
 "references": [
   {
     "refType": "CHILD_OF",
     "traceID": "81289be65e00618d84366dfe2f7fc1a2",
     "spanID": "75473187e1bc7579"
   }
 ],
 "startTime": 1659199980429164,
 "duration": 143,
 "tags": [
   {
     "key": "otel.library.name",
     "type": "string",
     "value": "data.file_reader"
   },
```

CODE CONTINUES ON NEXT PAGE

```
    {
      "key": "span.kind",
      "type": "string",
      "value": "internal"
    },
    {
      "key": "internal.span.format",
      "type": "string",
      "value": "proto"
    }
  ],
  "logs": [
    {
      "timestamp": 1659199980429173,
      "fields": [
        {
          "key": "event",
          "type": "string",
          "value": "Opening data file."
        }
      ]
    },
    {
      "timestamp": 1659199980429301,
      "fields": [
        {
          "key": "event",
          "type": "string",
          "value": "Finished reading data file."
        }
      ]
    }
  ],
  "processID": "p1"
}
```

In OpenTelemetry, the Tracer creates the spans. It's an object that tracks the currently active span while allowing you to create new spans. As spans start and complete, the Tracer dispatches them to the back end you configured on the Collector.

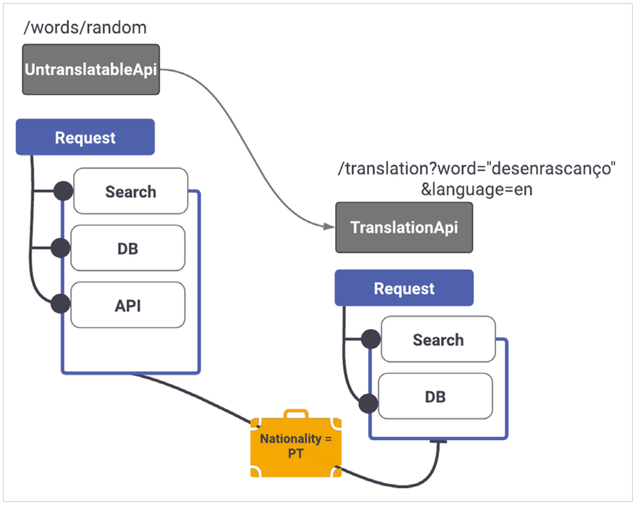
LOGS

A log is recorded as lines of text that describe a timestamped event and can be output in plain text, structured text (like JSON), or binary code. They result from a code execution block and are convenient for troubleshooting systems less prone to instrumentation (e.g., databases, load balancers). OpenTelemetry assumes that any data that doesn't belong to a trace or metric must be a log.

BAGGAGE

As the name hints, Baggage refers to contextual information passed on between spans. It's represented in OpenTelemetry by a key-value store that lives in a Trace Context, making those values available to all spans created within that trace. OpenTelemetry uses Context Propagation to pass around Baggage and exists in all libraries, so you don't have to implement it yourself. Baggage is designed to be language agnostic so that it can travel through stacks. Transporting downstream values that are only available higher in the stack makes it easier to filter when searching in your back end.

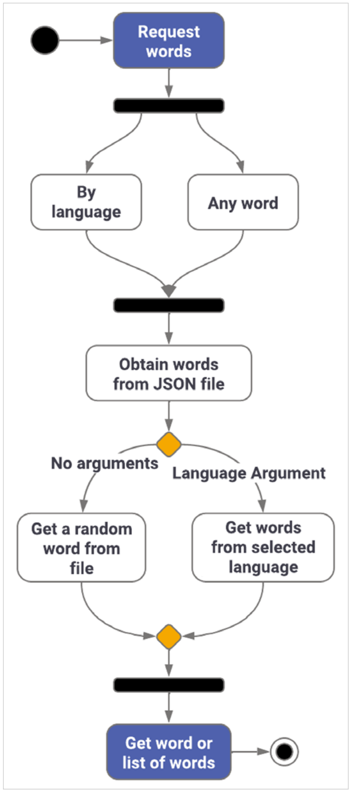
Figure 7: Baggage passing between two services



GETTING STARTED WITH OPENTELEMETRY

Now, with more context about the main concepts, architecture, and components of OpenTelemetry, we are ready to start tracing. We will instrument two APIs; one will be built in .NET and the other in Python. They will be designed to have the same endpoints and the same purpose. It will return untranslatable words that exist only in one language at random or by language. In this diagram, you can follow each API's simple flows:

Figure 8: Untranslatable API flow chart


Request words
By language
Any word
Obtain words from JSON file
No arguments
Language Argument
Get a random word from file
Get words from selected language
Get word or list of words

Different programming language paradigms present us with different challenges, so I've selected to show examples in two languages, Python and .NET — not to highlight the challenges but to demonstrate the consistency of OpenTelemetry across stacks. Please note that all .NET examples are for ASP.NET Core, and the configuration might differ for the .NET Framework.

CONFIGURATION

First, imagine that we already have a project created for whatever language we will use with the basic structure. You will install (Python) or add the necessary libraries (.NET) to your project by running the following commands. For Python, if using *setuptools*, you can add this library as an installation requirement.

Python:

```
$ pip install opentelemetry-distro
```

.NET:

```
$ dotnet add package OpenTelemetry --prerelease

$ dotnet add package OpenTelemetry.Instrumentation.
AspNetCore --prerelease

$ dotnet add package OpenTelemetry.Extensions.
Hosting --prerelease
```

These commands will also add the SDK and API for OpenTelemetry as a dependency.

COLLECT TRACES USING OPENTELEMETRY

In OTel, we can perform tracing operations on a *Tracer*. We can obtain it by using `GetTracer()` in the global Tracer Provider, returning an object that can be used for tracing operations. However, when using auto-instrumentation and depending on the language, that might not be necessary.

ADD A SIMPLE TRACE WITH AUTOMATIC INSTRUMENTATION

Not all frameworks offer automatic instrumentation, but OTel advises using it for those that do. Not only does it save lines of code, but it also provides a baseline for telemetry with little work. It works by attaching an agent to the running application and extracting tracing data. When considering auto-instrumentation, remember that it's not as flexible as manual instrumentation and only captures basic signals.

Let us look at code implementations. Below, we have the basic setup for auto-instrumenting our API.

Python:

```
# app.py
from flask import Flask, Response
app = Flask(__name__)

@app.route("/")
```

CODE CONTINUES IN NEXT COLUMN

```
@app.route("/home")
@app.route("/index")
def index():
   return Response("Welcome to Untranslatable!",
status=200)
# Add more actions here

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

.NET:

```
// Program.cs
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;

var serviceName = "untranslatable-dotnet";
var serviceVersion = "1.0.0";

var builder = WebApplication.CreateBuilder(args);
var resource = ResourceBuilder.CreateDefault().
AddService(serviceName);

builder.Services.
AddOpenTelemetryTracing(tracerProviderBuilder =>
   tracerProviderBuilder
   .SetResourceBuilder(resource)
   .AddSource(serviceName)
   .SetResourceBuilder(
       ResourceBuilder.CreateDefault()
           .AddService(serviceName: serviceName,
serviceVersion: serviceVersion))
  .AddAspNetCoreInstrumentation()
   .AddConsoleExporter()
).AddSingleton(TracerProvider.Default.
GetTracer(serviceName));

var app = builder.Build();

//… Rest of the setup and actions here
```

In Python, we don't need to add anything to the code to extract basic metrics, but I'd recommend using the FlaskInstrumentor that adds flask-specific features support. You can add `FlaskInstrumentor().instrument_app(app)` after instantiating Flask and add extra configurations as needed.

In .NET, we need to configure necessary OpenTelemetry settings as the exporter, instrumentation library, and constants. Like in Python, adding the `OpenTelemetry.Instrumentation.AspNetCore` package will provide extra features specific to the framework, adding to the base instrumentation library.

The instrumentation library for ASP.NET Core will automatically create spans and traces from inbound HTTP requests.

To run our applications with automatic instrumentation and start collecting and exporting telemetry, run the commands below.

Python:

```
$ python3 -m venv .
$ source ./bin/activate
$ pip install .
$ opentelemetry-bootstrap -a install
$ opentelemetry-instrument \
    --traces_exporter console \
    --metrics_exporter console \
    flask run
```

.NET:

```
$ dotnet run Untranslatable.Api.csproj
```

These commands will start the instrument agent and set up the specific instrumentation libraries. Now, a trace will be printed to the console whenever we send a request.

We can see the examples of output below.

Python:

```
{
 "name": "/words",
 "context": {
   "trace_id": "0x55072f6cc00531a489613e782942f75a",
   "span_id": "0x28135f1ccf37d85f",
   "trace_state": "[]"
 },
 "kind": "SpanKind.SERVER",
 "parent_id": null,
 "start_time": "2022-07-28T10:14:38.951442Z",
 "end_time": "2022-07-28T10:14:38.952775Z",
 "status": {
   "status_code": "UNSET"
 },
 "attributes": {
 "http.method": "GET",
   "http.server_name": "127.0.0.1",
   "http.scheme": "http",
   "net.host.port": 8000,
   "http.host": "127.0.0.1:8000",
   "http.target": "/words?language='es'",
  "net.peer.ip": "127.0.0.1",
   "http.user_agent": "python-requests/2.28.1",
   "net.peer.port": 58618,
   "http.flavor": "1.1",
   "http.route": "/words",
   "http.status_code": 200
 },
 "events": [],
```

CODE CONTINUES IN NEXT COLUMN

```
 "links": [],
 "resource": {
   "telemetry.sdk.language": "python",
   "telemetry.sdk.name": "opentelemetry",
   "telemetry.sdk.version": "1.12.0rc2",
   "telemetry.auto.version": "0.32b0",
   "service.name": "unknown_service"
 }
}
```

.NET:

```
Activity.TraceId:
e5e958c3cf3cfb4819605c102cdcfeba
Activity.SpanId:              b75dd2c4abb36412
Activity.TraceFlags:          Recorded
Activity.ActivitySourceName: OpenTelemetry.
Instrumentation.AspNetCore
Activity.DisplayName: words
Activity.Kind:        Server
Activity.StartTime:   2022-07-28T10:10:42.9292690Z
Activity.Duration:    00:00:00.0004600
Activity.Tags:
      http.host: localhost:7104
      http.method: GET
      http.target: /words
      http.url: http://localhost:7104/
words?language=pt
      http.user_agent: python-requests/2.28.1
      http.route: words
      http.status_code: 200
   StatusCode : UNSET
Resource associated with Activity:
      service.name: untranslatable-dotnet
      service.instance.id: d715f73f-3147-4708-aec6-
98bd75a3ad77
```

Notice that in Python, the traces have many empty unknown values by not adding any configuration.

ADD MANUAL INSTRUMENTATION

Manual instrumentation means adding extra code to the application to start and finish spans, define payload, and add counters or events. You can use client libraries and SDKs available for many programming languages. Manual instrumentation and automatic instrumentation should walk hand in hand as they complement each other.

Instrumenting your application with intention will augment the automated instrumentation and provide better and deeper observability. The implementation on the following page will add traces to the APIs' methods.

Python:

```python
# app.py
# the libraries you already had
from opentelemetry import trace
from opentelemetry.sdk.resources import SERVICE_
NAME, Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import
BatchSpanProcessor, ConsoleSpanExporter
from opentelemetry.instrumentation.flask import
FlaskInstrumentor

resource = Resource(attributes={SERVICE_NAME:
"untranslatable-python"})

tracer_provider = TracerProvider(resource=resource)
trace.set_tracer_provider(tracer_provider)
tracer = trace.get_tracer(__name__)
trace.get_tracer_provider().add_span_processor(
   BatchSpanProcessor(ConsoleSpanExporter())
)

app = Flask(__name__)
FlaskInstrumentor().instrument_app(app)

@tracer.start_as_current_span("welcome-message")
@app.route("/")
def index():
   return Response("Welcome to Untranslatable!",
status=200)

@app.route("/words/random", methods=["GET"])
def word_random():
   with tracer.start_as_current_span("random-word"):
       data = read_json_from_file()
       words = json.dumps(data)
       random_word = random.choice(words)

   return Response(random_word,
mimetype="application/json", status=200)

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

.NET differs from other languages that support OpenTelemetry. The System.Diagnostics API implements tracing, reusing existing objects like `ActivitySource and Activity` to comply with OpenTelemetry under the hood. For consistency, I've used the OpenTelemetry Tracing Shim so that you can learn to use OpenTelemetry concepts.

If you want to see an implementation using `Activities`, you can check this repo.

.NET:

```csharp
// UntranslatableController.cs
using System.Linq;
using System.Threading;
using Microsoft.AspNetCore.Mvc;
using OpenTelemetry.Trace;
using Untranslatable.Api.Controllers.Extensions;
using Untranslatable.Api.Models;
using Untranslatable.Data;
using Untranslatable.Shared.Monitoring;

namespace Untranslatable.Api.Controllers
{
   [ApiController]
   [Route("words")]
   [Produces("application/json")]
   public class WordsController : ControllerBase
   {
       private readonly IWordsRepository
wordsRepository;
       private readonly Tracer tracer;

       public WordsController(Tracer tracer,
IWordsRepository wordsRepository)
       {
           this.wordsRepository = wordsRepository;
           this.tracer = tracer;
       }

       [HttpGet]
       public ActionResult<UntranslatableWordDto>
Get([FromQuery] string language = null,
CancellationToken cancellationToken = default)
       {
           using var span = this.tracer?.
StartActiveSpan("GetWordByLanguage");

           Metrics.Endpoints.WordsCounter.Add(1);
           using (Metrics.Endpoints.WordsTime.
StartTimer())
           {
               var allWords = Enumerable.
Empty<UntranslatableWord>();
               using (var childSpan1 = tracer.
StartActiveSpan("GetByLanguageFromRepository"))
               {
                   childSpan1.AddEvent("Started
loading words from file...");
                   allWords = wordsRepository.
GetByLanguage(language, cancellationToken);
                   childSpan1.AddEvent("Finished
loading words from file...");
               }
```

CODE CONTINUES ON NEXT PAGE

```
               using (tracer.
StartActiveSpan("WordsToArray"))
               {
                   var result = allWords.Select(w =>
w.ToDto()).ToArray();
                   return Ok(result);
               }
           }
       }

       [HttpGet]
       [Route("random")]
       public ActionResult<UntranslatableWordDto>
GetRandom(CancellationToken cancellationToken =
default)
       {
           using var span = this.tracer?.
StartActiveSpan("GetRandomWord");

           Metrics.Endpoints.WordRandom.Add(1);
           using (Metrics.Endpoints.WordRandomTime.
StartTimer())
           {
               span.AddEvent("GetRandomWord");
               var word = wordsRepository.
GetRandom(cancellationToken);
               span.AddEvent("Done select Random
Word");

               return Ok(word.ToDto());
           }
       }
   }
}
```

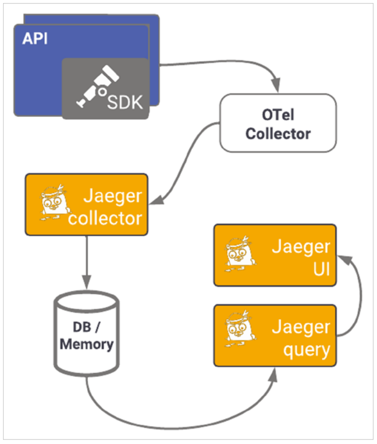
STORE AND VISUALIZE DATA

Jaeger is a popular open-source distributed tracing tool initially built by teams at Uber and then open sourced once it became part of the CNCF family. It's a back-end application for tracing that allows developers to view, search, and analyze traces. One of its most powerful functionalities is visualizing request traces through services in a system domain, enabling engineers to quickly pinpoint failures in complex architectures.

Jaeger provides instrumentation libraries built on OpenTracing standards. Using the specific exporter for Jaeger can offer a quick win on observing your application. Here we will use the OTel exporter and OpenTelemetry's Jaeger exporter to send OTel traces to a Jaeger back-end service.

We've seen how the OTel collector works and is set up; Figure 9 in the next column shows what using Jaeger's specific exporter pipeline looks like.

Figure 9: OTel Collector pipeline



To start visualizing data, you need to set up Jaeger first. You can opt for other setups, but I'll use the all-in-one image to install the collector, query, and Jaeger UI in one container, using memory as default storage (not for production environments).

This `docker-compose` file sets up all components, the network, the ports needed, and the OTel Collector. The ports used in this example are the default ports for each service. Run `docker-compose up` to start the containers.

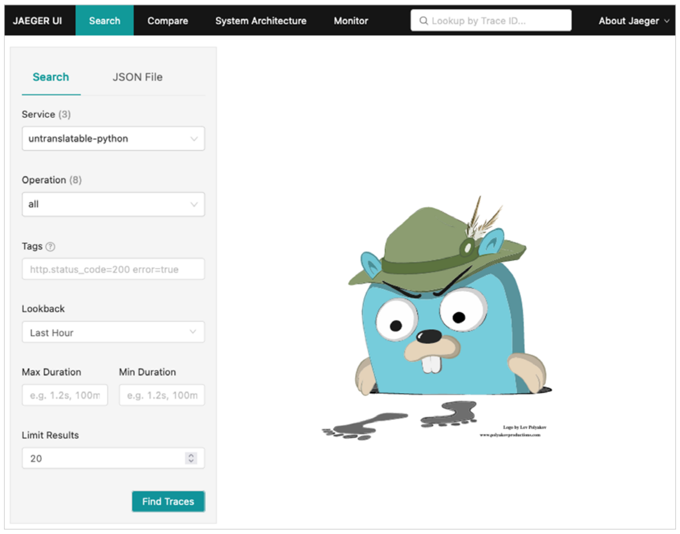
```
version: "3.5"
services:
 jaeger:
   networks:
     - backend
   image: jaegertracing/all-in-one:latest
   ports:
     - "16686:16686"
     - "14268"
     - "14250"
 otel_collector:
   networks:
     - backend
   image: otel/opentelemetry-collector:latest
   volumes:
     - "/YOUR/FOLDER/otel-collector-config.yml:/etc/
otelcol/otel-collector-config.yml"
   command: --config /etc/otelcol/otel-collector-
config.yml
   environment:
     - OTEL_EXPORTER_JAEGER_GRPC_INSECURE:true
   ports:
     - "1888:1888"
     - "13133:13133"
     - "4317:4317"
     - "4318:4318"
     - "55670:55679"
   depends_on:
     - jaeger
networks:
 backend:
```

You should now have two containers running, one for Jaeger and another for the OTel collector:

```
NAMES                 STATUS
otel_collector-1      Up 23 minutes
jaeger-1              Up 23 minutes
```

If you navigate to `http://localhost:16686`, you should see Jaeger's UI. Here you'll be able to explore the traces generated by your instrumentation:

Figure 10: Jaeger's user interface



Service (3): untranslatable-python
Operation (8): all
Tags: http.status_code=200 error=true
Lookback: Last Hour
Max Duration: e.g. 1.2s, 100ms
Min Duration: e.g. 1.2s, 100ms
Limit Results: 20

In the top left drop-down menu is the service we created. Services are added to that list when we export traces to Jaeger. As I've mentioned, there are two ways to export telemetry to Jaeger, using the OTLP or directly to Jaeger using one of the supported protocols. We've already seen how to configure the OTLP collector, so now all we have to configure is the Collector to export to Jaeger:

```
receivers:
 otlp:
   protocols:
     http:
       endpoint: 0.0.0.0:4318
     grpc:
       endpoint: 0.0.0.0:4317
processors:
 batch:
   timeout: 1s
exporters:
 logging:
   loglevel: info
 jaeger:
   endpoint: jaeger:14250
   tls:
     insecure: true
extensions:
 health_check:
 pprof:
   endpoint: :1888
 zpages:
   endpoint: :55679
service:
 extensions: [pprof, zpages, health_check]
 pipelines:
   traces:
     receivers: [otlp]
     processors: [batch]
     exporters: [logging, jaeger]
```

CODE CONTINUES IN NEXT COLUMN

However, if setting up a collector seems daunting, or if you want to start small using OpenTelemetry, sending data directly to a back end can offer results reasonably fast without the Collector. Let's start by installing OpenTelemetry's Jaeger exporter.

Python:

```
$ pip install opentelemetry-exporter-jaeger
```

.NET:

```
$ dotnet add package OpenTelemetry.Exporter.Jaeger
```

Again, in Python, we install the package on our host or the virtual environment, whereas for .NET, we add it directly as a project dependency. For Python, the package comes with both gRPC and Thrift protocols.

Python:

```
# app.cs
# ... other imports
from opentelemetry import trace
from opentelemetry.exporter.jaeger.thrift import
JaegerExporter
from opentelemetry.sdk.trace.export import
BatchSpanProcessor
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.resources import SERVICE_
NAME, Resource

resource = Resource(attributes={SERVICE_NAME:
"untranslatable-python"})

jaeger_exporter = JaegerExporter(
   agent_host_name="localhost",
   agent_port=6831,

   collector_endpoint="http://localhost:14268/api/
traces?format=jaeger.thrift",
)

tracer_provider = TracerProvider(resource=resource)
jaeger_processor = BatchSpanProcessor(jaeger_exporter)
tracer_provider.add_span_processor(jaeger_processor)
```

CODE CONTINUES ON NEXT PAGE

```
trace.set_tracer_provider(tracer_provider)
tracer = trace.get_tracer(__name__)
#... rest of initializations and actions
```

After installing the package, you can set the exporter in the `TracerProvider`, which will be configured when tracing starts. Now we will do the same for .NET.

After adding the NuGet package to the project, we will configure the exporter. Here we will enable instrumentation using an extension method — `AddAspNetCoreInstrumentation` — on `IServiceCollection` and binding the Jaeger exporter.

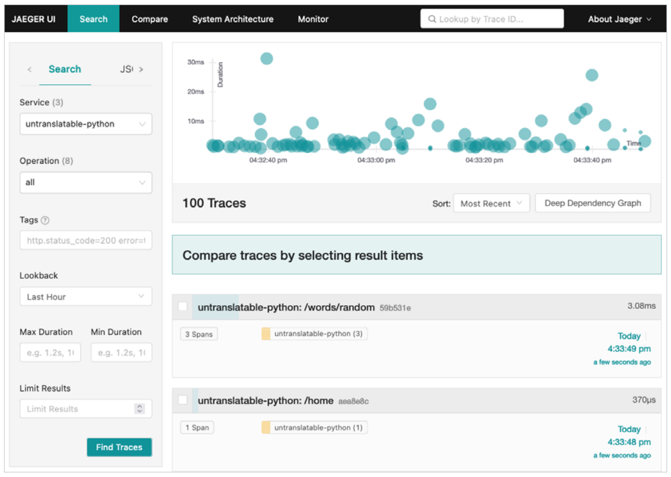
.NET:

```
// Program.cs
// ... other imports and initializations
var serviceName = "untranslatable-dotnet";
var serviceVersion = "1.0.0";

var resource = ResourceBuilder.CreateDefault().
AddService(serviceName);
builder.Services.AddOpenTelemetryTracing(b => b
   // ...  rest of setup code
   .AddJaegerExporter(o =>
   {
              o.AgentHost = "localhost";
       o.AgentPort = 6831;
       o.Endpoint = new Uri("http://localhost:14268/
api/traces?format=jaeger.thrift");
   })
).AddSingleton(TracerProvider.Default.
GetTracer(serviceName));
// ...  rest of initializations and actions
```

Now run your APIs and make some requests. Go to Jaeger's UI, and you should be able to see traces generated by those requests by selecting the service name you specified and the operation you traced. Below, you can see all traces captured within a time window and a trace's detail and the associated spans:

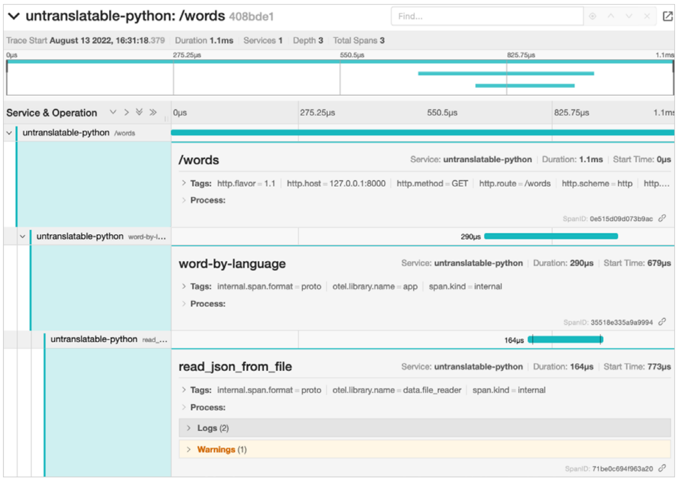
Figure 11: All traces



untranslatable-python: /words/random 59b531e 3.08ms
3 Spans | untranslatable-python (3) | Today 4:33:49 pm, a few seconds ago

untranslatable-python: /home aea8e8c 370μs
1 Span | untranslatable-python (1) | Today 4:33:48 pm, a few seconds ago

Figure 12: Trace details


untranslatable-python: /words 408bde1
Find...
Trace Start August 13 2022, 16:31:18.379
Duration 1.1ms
Services 1
Depth 3
Total Spans 3
0μs
275.25μs
550.5μs
825.75μs
1.1ms
Service & Operation
untranslatable-python /words
untranslatable-python word-by-l...
untranslatable-python read_...
290μs
164μs
Logs (2)
Warnings (1)

/words — Service: untranslatable-python | Duration: 1.1ms | Start Time: 0μs
Tags: http.flavor = 1.1 | http.host = 127.0.0.1:8000 | http.method = GET | http.route = /words | http.scheme = http | http....
Process:
SpanID: 0e515d09d073b9ac

word-by-language — Service: untranslatable-python | Duration: 290μs | Start Time: 679μs
Tags: internal.span.format = proto | otel.library.name = app | span.kind = internal
Process:
SpanID: 35518e335a9a9994

read_json_from_file — Service: untranslatable-python | Duration: 164μs | Start Time: 773μs
Tags: internal.span.format = proto | otel.library.name = data.file_reader | span.kind = internal
Process:
SpanID: 71be0c694f963a20

For complex systems and architectures, distributed tracing is invaluable. You can quickly start exporting directly to Jaeger's back end and adding OpenTelemetry auto-instrumentation to get the telemetry data. With Jaeger, it's easier to find where the problem occurred than through logs, allowing you to monitor transactions, perform root cause analysis, optimize performance and latency, and visualize service dependencies.

COMMON PITFALLS OF MIGRATING LEGACY APPLICATIONS TO OPENTELEMETRY

Your services are probably already emitting telemetry data bound to some observability back end. Changing your observability architecture can be very painful:

- Re-instrumenting is time consuming
- Data will change
- Telemetry data must continue to flow, not allowing blind stops in the system
- Traces have to remain linked

To migrate sequentially and as seamlessly as possible, you can use the OpenTelemetry Collector as a proxy between your services and the back ends you use. The Collector can replace most telemetry services, removing the need for separate services for processing and transmitting signals, making them redundant. Its telemetry pipeline's flexibility lets you configure any compatible back end or service while keeping your code agnostic.

Suppose you want to start migrating all your applications slowly. In that case, the Collector can translate any input into the output you need; you can move an application to OpenTelemetry and send data to the same back end.

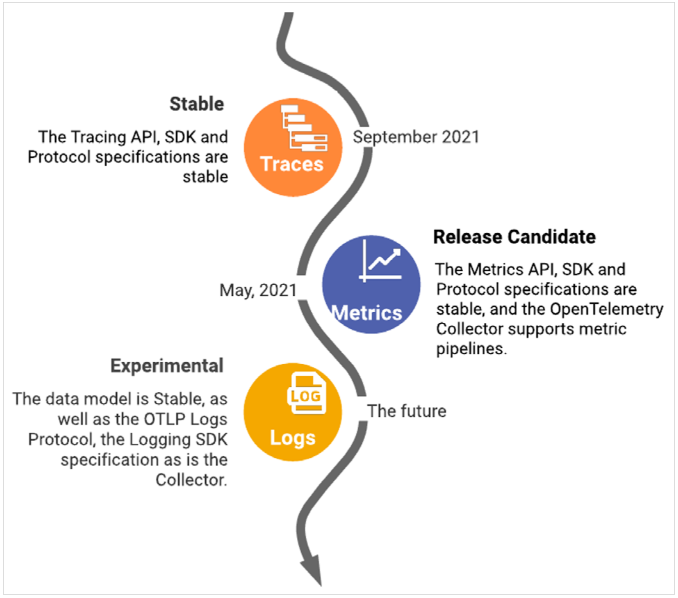
Note that when changing instrumentation libraries, the output produced changes, so you might have to adjust your dashboards and alerting systems.

CONCLUSION

Correlation does not equal causation — we must interpret the meaning of every correlation. But who has the time? OpenTelemetry aims to simplify data collection to focus on data analysis and processing while creating a standard to abstract from the previous proprietary and in-house solutions. Collecting and reviewing the data takes time; automating this process is a massive win for observability.

As of writing this Refcard, below is the status of OpenTelemetry Signals:

Figure 13: Timeline of OpenTelemetry Signals status


Stable
The Tracing API, SDK and Protocol specifications are stable
Traces
September 2021
Release Candidate
The Metrics API, SDK and Protocol specifications are stable, and the OpenTelemetry Collector supports metric pipelines.
May, 2021
Metrics
Experimental
The data model is Stable, as well as the OTLP Logs Protocol, the Logging SDK specification as is the Collector.
LOG
Logs
The future

With [93 pull requests per week] and over [450 companies] backing up and maintaining the project, it provides access to an extensive set of telemetry collection frameworks.

Having the community's backing means that you'll have to wait for a shorter period from the need identification to the supply. By not having product or profit concerns, the community can respond promptly and offer support for new technologies without waiting for vendors' support.

By standardizing how frameworks and applications collect and send observability data, OpenTelemetry helps solve the challenges created by the different stacks and back ends, giving teams a vendor-neutral, portable, and pluggable solution that is easily configured with open-source and commercial solutions alike.

Additional resources:

- OpenTelemetry Documentation – https://opentelemetry.io/docs
- OpenTelemetry DevStats Dashboard – https://opentelemetry.devstats.cncf.io/d/8/dashboards
- OpenTelemetry implementation status per language – https://github.com/open-telemetry/opentelemetry-specification/blob/main/spec-compliance-matrix.md
- OpenTelemetry Twitter – https://twitter.com/opentelemetry
- "Full-Stack Observability Essentials" Refcard – https://dzone.com/refcardz/full-stack-observability-essentials
- "Distributed Tracing in ASP.NET Core With Jaeger and Tye, Part 1: Distributed Tracing" – https://dzone.com/articles/distributed-tracing-in-aspnet-core-with-jaeger-and
- Distributed Tracing Overview – https://www.logicmonitor.com/support/tracing/getting-started-with-tracing
- "Getting Started With Log Management" Refcard – https://dzone.com/refcardz/log-management

WRITTEN BY JOANA CARVALHO,
PERFORMANCE ENGINEER, POSTMAN

Joana has been a performance engineer for the last 10 years. She analyzed root causes from user interaction to bare metal, performance tuning, and new technology evaluation. Her goal is to create solutions to empower the development teams to own performance investigation, visualization, and reporting so that they can, in a self-sufficient manner, own the quality of their services. Currently working at Postman, she mainly implements performance profiling, evaluation, analysis, and tuning.

DZone
600 Park Offices Drive, Suite 300
Research Triangle Park, NC 27709
888.678.0399 | 919.678.0300

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.