Unlocking Observability with OpenTelemetry

A Journey into Distributed Tracing and Monitoring

</June 2023>

zartis

Agenda

01 Introduction to Observability

02 OpenTelemetry overview

03 Telemetry data collection

04 Exporting and visualization

05 OpenTelemetry Ecosystem and Community

06 Demo

07 Q&A

1

Introduction to Observability

What is observability? Why is it important?

Observability in software refers to the ability to **understand the internal state of complex systems** by collecting and analyzing its outputs: **logs, metrics, and traces**.

Observability is important because it enables efficient **debugging**, efficient **troubleshooting** and proactive **monitoring**, among other interesting capabilities, in the context of complex software systems.

2

OpenTelemetry overview

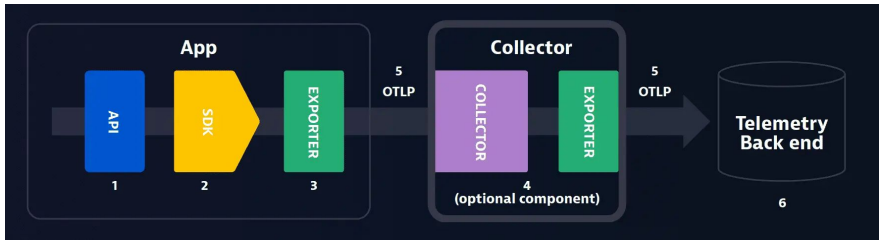
OpenTelemetry Origins

< 2016 Traditionally, Observability was achieved by **combining different tools** (loggers and log aggregators, monitoring and tracing solutions) or by using **vendor specific products** (New Relic, Dynatrace, Data Dog...). Each tool and vendor with their own standards.

2016 Cloud community, noticing the need for standardization and flexibility, created **two Open Source projects**: OpenTracing (CNCF) and OpenCensus (Google).

2019 Both projects merged into a single Open Source project called OpenTelemetry (OTel) adding additional features.

OpenTelemetry architecture



OpenTelemetry API: Definition of data types and operations for generating and correlating tracing, metrics, and logging data. Language specific.

OpenTelemetry SDKs: Uses the OpenTelemetry API to generate telemetry data within your technological stack and language of choice. Bridge between API and Exporter.

OpenTelemetry Exporters: Defines where and how to send collected telemetry data.

OpenTelemetry Collectors: Vendor-agnostic proxy that can receive, process, and export telemetry data. It supports receiving telemetry data in multiple formats. This is an optional component.

OpenTelemetry Backends: Not included within OpenTelemetry.

Instrumentation with OpenTelemetry

Software Instrumentation: Process of adding code/infrastructure to a system or application to collect data for analysis, monitoring, or debugging purposes.

```java
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.api.trace.TracerProvider;
import io.opentelemetry.api.trace.attributes.SemanticAttributes;
import io.opentelemetry.context.Scope;
import io.opentelemetry.sdk.OpenTelemetrySdk;
import io.opentelemetry.sdk.trace.SdkTracerProvider;

public class InstrumentationExample {

    public static void main(String[] args) {
        // Create a new OpenTelemetry tracer provider
        TracerProvider tracerProvider = SdkTracerProvider.builder().build();
        OpenTelemetry openTelemetry = OpenTelemetrySdk.builder().setTracerProvider(tracerProvider).build();

        // Get the tracer
        Tracer tracer = openTelemetry.getTracer("instrumentation-example");

        // Start a new span
        try (Scope scope = tracer.spanBuilder("example-span").startScopedSpan()) {
            Span span = Span.current();

            // Perform some operation
            System.out.println("Performing some operation...");

            // Add attributes to the span
            span.setAttribute(SemanticAttributes.HTTP_METHOD, "GET");
            span.setAttribute(SemanticAttributes.HTTP_STATUS_CODE, 200);

            // Simulate some work
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        } // Span automatically ends when the scope is closed

        System.out.println("Instrumentation example complete.");
    }
}
```

```python
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import SimpleSpanProcessor
from opentelemetry.exporter import jaeger

# Configure the Jaeger exporter
jaeger_exporter = jaeger.JaegerSpanExporter(service_name="instrumentation-example")
trace.set_tracer_provider(TracerProvider())
tracer = trace.get_tracer(__name__)
trace.get_tracer_provider().add_span_processor(SimpleSpanProcessor(jaeger_exporter))

# Start a new span
with tracer.start_as_current_span("example-span") as span:
    # Perform some operation
    print("Performing some operation...")

    # Add attributes to the span
    span.set_attribute("custom-attribute", "example-value")

    # Simulate some work
    import time
    time.sleep(2)

print("Instrumentation example complete.")
```

```csharp
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;
using System.Diagnostics.Metrics;

const string serviceVersion = "1.0.0";

var serviceName = AppDomain.CurrentDomain.FriendlyName;
var meter = new Meter(serviceName);

var builder = WebApplication.CreateBuilder(args);
var appResourceBuilder = ResourceBuilder.CreateDefault().AddService(serviceName: serviceName, serviceVersion: serviceVersion);

builder.Services.AddControllers();

//Open telemetry
builder.Services.AddOpenTelemetry()
    .WithTracing(tracerProviderBuilder =>
    {
        tracerProviderBuilder
            .AddSource(serviceName)
            .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService(serviceName: serviceName, serviceVersion: serviceVersion))
            .AddHttpClientInstrumentation()
            .AddAspNetCoreInstrumentation()
            .AddSqlClientInstrumentation()
            .AddEntityFrameworkCoreInstrumentation()
            .AddConsoleExporter();
    })
    .WithMetrics(metricProviderBuilder =>
    {
        metricProviderBuilder
            .AddConsoleExporter()
            .AddMeter(meter.Name)
            .SetResourceBuilder(appResourceBuilder)
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation();
    });

var app = builder.Build();

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

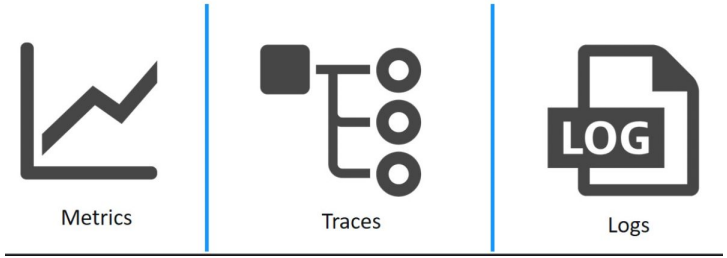
3

Telemetry data collection

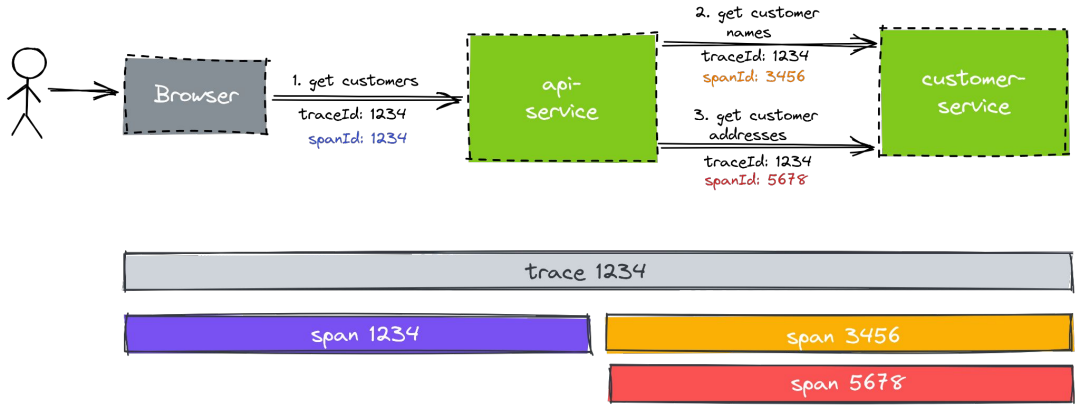
Types of telemetry data

The types of data that OpenTelemetry handles are three:

- **Metrics**: quantitative measurement that provides insights into the performance and behavior of a system or application.
- **Traces**: sequential record of events and interactions that occur during the processing of a specific request or transaction within an application or distributed system.
- **Logs**: record of significant events or messages generated by a system, aiding in understanding system behavior and troubleshooting.



How telemetry data is correlated together



Two key concepts, Trace identifier and Span identifier.

- **Trace id**: Globally unique identifier that represents an entire trace across multiple services or components.
- **Span id**: Local unique identifier assigned to specific operations or activities within the system, such as method invocations, network requests, or database queries.

4

Exporting and visualization

Exporting telemetry data

OpenTelemetry SDKs provides a **wide range of exporters**, from standard OTel protocol (vendor agnostic) exporter to vendor specific exporters that allows you to send collected data to observability platforms (e.g. Grafana, Elastic Observability, Data Dog…)

Exporter **configuration** allows to define authentication credentials, batching, compression and other details that can be handy.

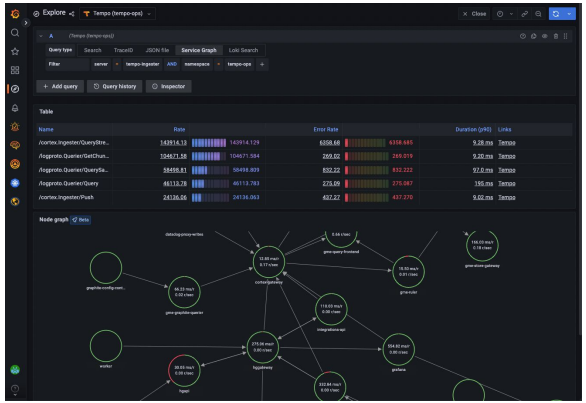
OpenTelemetry is really **extensible** so you can even define your own exporters!

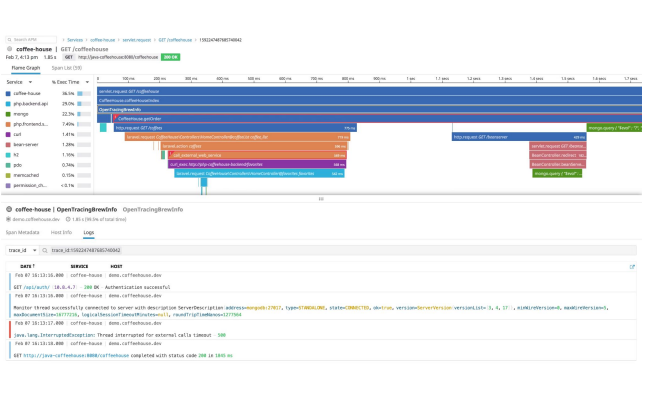
```
using OpenTelemetry;
using OpenTelemetry.Exporter.Jaeger;
using OpenTelemetry.Trace;

0 references
class Program
{
    0 references
    static void Main()
    {
        // Configure the Jaeger exporter
        var jaegerExporter = new JaegerExporter(
            new JaegerExporterOptions
            {
                ServiceName = "my-service",
                AgentHost = "localhost",
                AgentPort = 6831,
            });

        // Create a tracer provider with the Jaeger exporter
        var tracerProvider = Sdk.CreateTracerProviderBuilder()
            .AddSource("your-application-source")
            .SetSampler(new AlwaysOnSampler())
            .AddExporter(jaegerExporter)
            .Build();

        // Application code here...
    }
}
```

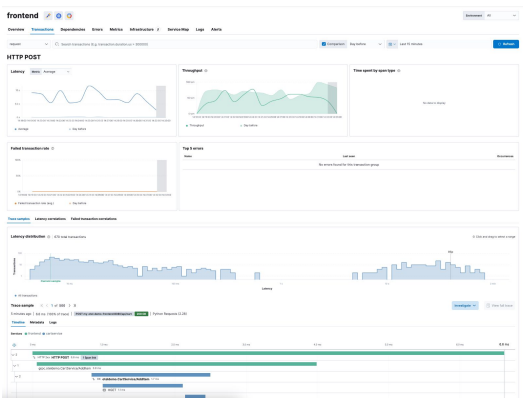
Visualizing telemetry data

OpenTelemetry **does not provide storage and visualization** for the collected telemetry data. For OpenTelemetry visualization, you need to use a backend that can ingest the collected data and provide an interface to visualize it.

There are numerous and well-known backends that can be used to visualize collected telemetry data: Grafana, Datadog, Elastic Observability.... it is even possible to create your own visualizer.







5

OpenTelemetry Ecosystem and Community

Current status of OpenTelemetry

Project

CNCF incubating project:
https://www.cncf.io/projects/opentelemetry/

Active development (more than 180 official collaborators, 62 repos, updated daily)

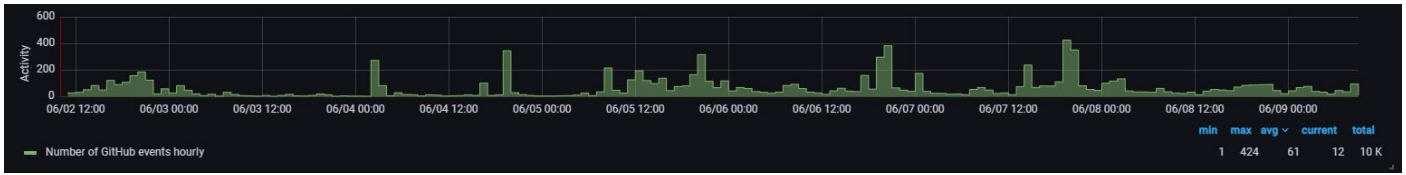
Current status: https://opentelemetry.io/status/

Additional links

Website: https://opentelemetry.io/community/

GiHub organization:
https://github.com/open-telemetry

Twitter: https://twitter.com/opentelemetry



6

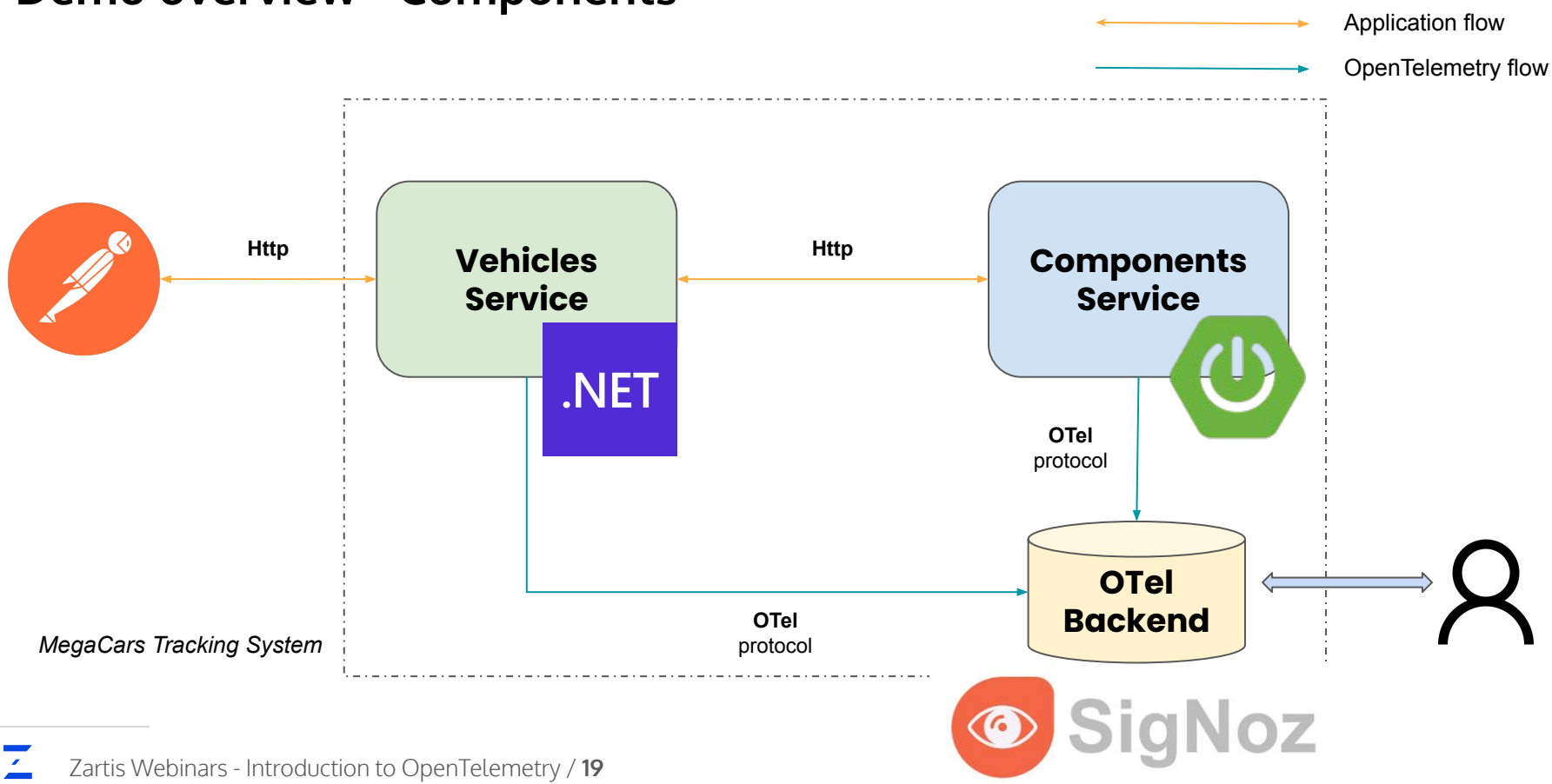
Demo

Demo overview - Business case

"MegaCars" is a visionary automobile company founded in the vibrant city of Innovia in 2020, a technological hub in the United States. Their distinctiveness lies in their emphasis on sustainable practices, as they specialize in manufacturing electric vehicles that prioritize eco-conscious design and manufacturing processes while delivering reliable performance.

As part of their **manufacturing process** they are using a software system that **tracks the cars they produce and the components** (engines, batteries…) **associated to each car**. Since this system is critical for the manufacturing process, **they want to monitor it** in order to make sure all works as expected.

Demo overview - Components



Code can be found here: https://github.com/zartis-digital/opentelemetry-webinar

7

Q&A

Q&A