

MEZIANTOU'S BLOG

Blog about Microsoft technologies (.NET, .NET Core, ASP.NET Core, WPF, UWP, TypeScript, etc.)

[HOME](#) [PROJECTS](#) [TALKS](#) [ARCHIVES](#) [CONTACT](#)



5 challenges to achieving
observability at scale

Monitoring a .NET application using OpenTelemetry

11/15/2021 Gérald Barré .NET

- [What is OpenTelemetry?](#)
- [Code instrumentation](#)
 - [Logging](#)
 - [Tracing](#)
 - [Metrics](#)
- [Exporting data](#)
 - [Starting the collector and back-ends](#)
 - [Configuring the application to export data to the collector](#)
- [Monitoring multiple services](#)
- [Enriching the request activity created by ASP.NET Core](#)
- [Additional resources](#)

Note: The samples in this post instrument an ASP.NET Core application, but you can instrument any kind of application using OpenTelemetry

What is OpenTelemetry?

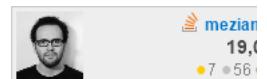
OpenTelemetry is a set of APIs, SDKs, tooling and integrations that are designed for the creation and management of telemetry data such as traces, metrics, and logs. The project provides a vendor-agnostic implementation that can be configured to send telemetry data to the backends of your choice. It supports a variety of popular open-source projects including Jaeger and Prometheus. Also, a lot of vendors support OpenTelemetry directly or using the OpenTelemetry Collector.

OpenTelemetry allows to monitor multiple services (distributed system) and correlates their events. You can correlate events:

- By the time of execution. Each event records the moment of time or the range of time the execution took place. This is a basic way to correlate events.
- By the execution context. Each event has a [TraceId](#) and [SpanId](#) associated with it, so you can correlate logs and traces that correspond to the same ids. Also, you can propagate the ids between services using standard mechanisms such as the [W3C tracecontext headers](#), so events from multiple services can be correlated.
- By the origin of the telemetry ([resource context](#)). It could be the service name, the service instance, or the service version.

GÉRALD BARRÉ

aka. meziantou



RECENT POSTS

[Uploading multiple files using InputFile in Blazor](#)

[Forcing HttpClient to use IPv4 IPv6 addresses](#)

[Converting code to the new Roslyn Source Generator](#)

[Performance benefits of sealed class in .NET](#)

[Validating a Blazor component on the first render](#)



OpenTelemetry defines 3 concepts when instrumenting an application:

- Logging ([specification](#)):

```
29 Jun 2020 09:59:59.388 POST HTTP POST /api/customers responded 400 in 536.343 ms
29 Jun 2020 09:59:57.964 GET HTTP GET /api/products/list responded 200 in 49.750 ms
29 Jun 2020 09:59:56.826 POST HTTP POST /api/customers responded 400 in 34.185 ms
29 Jun 2020 09:59:55.794 POST HTTP POST /api/customers responded 400 in 51.848 ms
29 Jun 2020 09:59:55.794 POST HTTP POST /api/customers responded 400 in 972.881 ms
29 Jun 2020 09:59:54.492 GET HTTP GET /api/products/list responded 200 in 548.502 ms
```

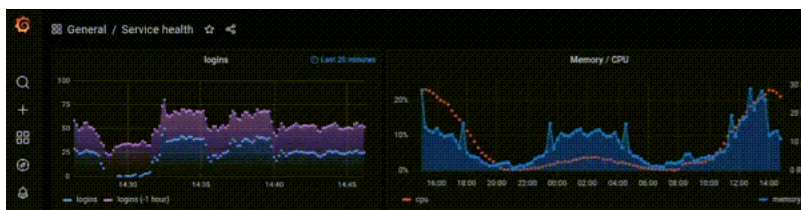
Example of logs is [Seq](#)

- Tracing ([specification](#)):



Example of tracing in [Jaeger](#)

- Metrics ([specification](#)):



Example of metrics in [Grafana](#)

Code instrumentation

Instrumenting the code for OpenTelemetry in .NET is a bit unique. Instead, you don't need to use an OpenTelemetry library. Instead you can use types provided in `System.Diagnostics` and `Microsoft.Extensions.Logging.Abstractions`. This means that most of the .NET code that use objects from these namespaces to instrument the code are already instrumented for OpenTelemetry. Also, there are lots of libraries that provides instrumentation for OpenTelemetry. You can use [OpenTelemetry.Instrumentation.*.packages](#) to collect data from common sources:

- [HttpClient](#)
- [ASP.NET Core](#)
- [SQL Server Client](#)
- [Redis client](#)
- [gRPC client](#)

LINKS

[Dev Tool List](#)

[Dev News](#)

[Async/await resources](#)

[.NET Multithreading resources](#)

[Visual Studio tips](#)

[Online tools](#)



- [AWS client](#)
- [Elasticsearch client](#)
- etc.

This means you can use OpenTelemetry to monitor your application and get insights even if you don't manually instrument your application. That's being said, you can still instrument your application manually if you want to.

Logging

OpenTelemetry relies on `Microsoft.Extensions.Logging.Abstractions` to handle logging. This library provides a set of interfaces and classes that enable you to create logging services that can be used to log messages from your application. It supports [structured logging](#), the arguments themselves are passed to the logging system, not just the formatted message template. This enables logging providers to store the parameter values as fields, which is useful to query your logs later. For example, using the following logger method, you could query all names containing a specific value requested within a specific time range.

C#



```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", (ILogger<Program> logger, string name) =>
{
    logger.LogInformation("Hello {Name}! It is {Time}", name, DateTime.UtcNow);
    return Results.Ok($"Hello {name}");
});
app.Run();
```

Tracing

OpenTelemetry relies on types in `System.Diagnostics.*` to support tracing. As those types have existed for a long time, this means that some parts of .NET and other libraries are already instrumented for OpenTelemetry. However, the naming is not the same as the one used by OpenTelemetry:

- `System.Diagnostics.ActivitySource` represents an [OpenTelemetry Tracer](#)
- `System.Diagnostics.Activity` represents an [OpenTelemetry Span](#)

You can add some info to the activity using `AddTag`. These data will be exported so you can see them in the tool you use (Zipkin, Jaeger, Azure Monitor, etc.). For instance, when doing an http request, you can add the domain and the status code as tags. Then, in your monitoring tool you can find which external services is not reliable enough.

You also add baggage using `AddBaggage`. Baggage will flow to child activities. This could be useful to flow a correlation id to all child activities, even the ones started on other services. Indeed, .NET sends baggages to other services automatically using W3C header ([source](#)).

C#



```
var activitySource = new ActivitySource("SampleActivitySource");

// ASP.NET Core starts an activity when handling a request
app.MapGet("/", async (string name) =>
{
    // The sampleActivity is automatically linked to the parent activity (the one from
    // ASP.NET Core in this case).
    // You can get the current activity using Activity.Current.
    using (var sampleActivity = activitySource.StartActivity("Sample", ActivityKind.Server))
    {
        // note that "sampleActivity" can be null here if nobody listen events generated
        // by the "SampleActivitySource" activity source.
        sampleActivity?.AddTag("Name", name);
    }
});
```



```

sampleActivity?.AddBaggage("SampleContext", name);

// Simulate a long running operation
await Task.Delay(1000);
}

return Results.Ok($"Hello {name}");
});

```

Metrics

As for tracing, Metrics API are incorporated directly into the .NET runtime itself. So, you can instrument your application by simply depending on `System.Diagnostics.*`. .NET supports 4 kind of metrics:

- Counter: Instrument that can be used to report monotonically increasing values. For example, you can increment the counter each time a request is processed to track the total number of requests. Most metric viewers display counters using a rate (requests/sec), by default, but can also display a cumulative total.
- ObservableCounter: Asynchronous instrument that reports monotonically increasing values. This is similar to a Counter, except the values are provided asynchronously.
- Histogram: Instrument that can be used to report arbitrary values that are likely to be statistically meaningful. It is intended for statistics such as histograms, summaries, and percentile.
- ObservableGauge: Asynchronous instrument that reports non-additive values when the instrument is being observed. An example of a non-additive value is the current number of tasks in an application.

C#

 copy

```

var meter = new Meter("MyApplication");

var counter = meter.CreateCounter<int>("Requests");
var histogram = meter.CreateHistogram<float>("RequestDuration", unit: "ms");
meter.CreateObservableGauge("ThreadCount", () => new[] { new Measurement<int>(ThreadPool.ThreadC


var httpClient = new HttpClient();
app.MapGet("/", async (string name) =>
{
    // Measure the number of requests
    counter.Add(1, KeyValuePair.Create<string, object?>("name", name));

    var stopwatch = Stopwatch.StartNew();
    await httpClient.GetStringAsync("https://www.meziantou.net");

    // Measure the duration in ms of requests and includes the host in the tags
    histogram.Record(stopwatch.ElapsedMilliseconds,
        tag: KeyValuePair.Create<string, object?>("Host", "www.meziantou.net"));

    return Results.Ok($"Hello {name}");
});

```

Using `Meter` also allows you to monitor the application using `dotnet counters` or `dotnet monitor`. You can install it using `dotnet tool install --global dotnet-counters` or [direct download link](#) . Then, you can use the following command to get the metrics:

Shell

 copy

```

dotnet tool install --global dotnet-counters
dotnet counters monitor --process-id 123 --counters MyApplication

```



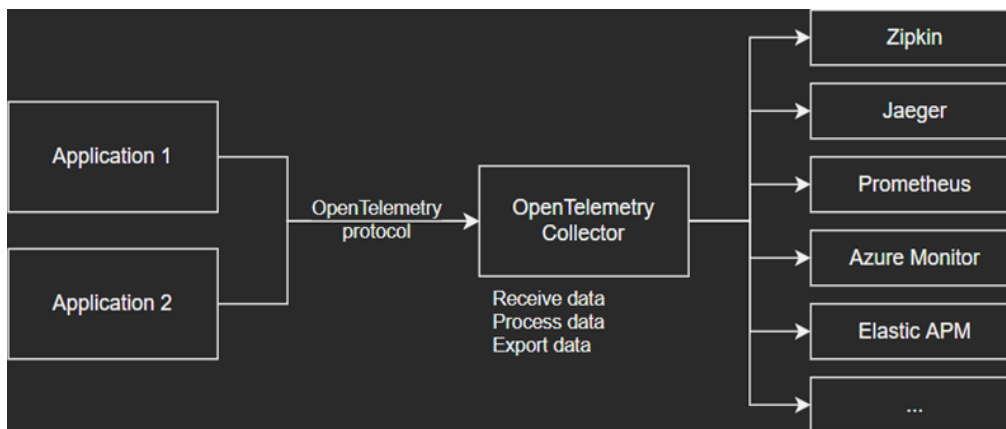
```
C:\WINDOWS\system32\cmd.exe x + v
Press p to pause, r to resume, q to quit.
Status: Running

[MyApplication]
RequestDuration (ms)
  Host=www.meziantou.net,Percentile=50      202
  Host=www.meziantou.net,Percentile=95      216
  Host=www.meziantou.net,Percentile=99      216
Requests (Count / 1 sec)                    5
ThreadCount                                4
```

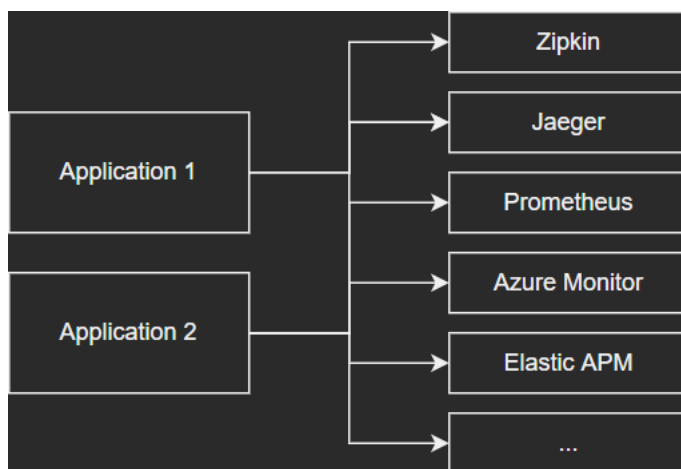
Exporting data

There are 2 ways to export data from OpenTelemetry:

- Using the [OpenTelemetry Collector](#)  (recommended)



- Exporting to each back-end directly



The recommended way is to use the collector to export data. It makes your application back-end agnostic and provides a consistent way to export data for all your applications. It also simplifies the integration in applications as you only need to export data to the collector using a single protocol. No need to write a



custom exporter for each back-end. Also, many vendors provide [their own exporters](#) for the OpenTelemetry Collector. So, you can use them even if these vendors don't provide an exporter for .NET.

If you want to export directly to the backends without using the OpenTelemetry Collector, you can use the NuGet packages [OpenTelemetry.Exporter.*](#).

There are multiple ways to deploy [OpenTelemetry Collector](#). You can check the documentation for all details. In the following section, we'll use docker-compose to start the server and a few back-ends.

Starting the collector and back-ends

The OpenTelemetry Collector is a generic service. You need to configure it to select how you want to receive, process and export data. The following example starts the collector and the back-ends. Also, it configures the collector to get data using the OpenTelemetry protocol and export data to Zipkin, Prometheus, and a file. Create the 3 following files in the same folder:

■ docker-compose.yaml

YAML

 copy

```
version: "2"
services:
  # back-ends
  zipkin-all-in-one:
    image: openzipkin/zipkin:latest
    ports:
      - "9411:9411"

  prometheus:
    container_name: prometheus
    image: prom/prometheus:latest
    volumes:
      - ./prometheus.yaml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  # OpenTelemetry Collector
  otel-collector:
    image: otel/opentelemetry-collector:latest
    command: ["--config=/etc/otel-collector-config.yaml"]
    volumes:
      - ./otel-collector-config.yaml:/etc/otel-collector-config.yaml
      - ./output:/etc/output:rw # Store the logs
    ports:
      - "8888:8888" # Prometheus metrics exposed by the collector
      - "8889:8889" # Prometheus exporter metrics
      - "4317:4317" # OTLP gRPC receiver
    depends_on:
      - zipkin-all-in-one
```

■ otel-collector-config.yaml

YAML

 copy

```
# Configure receivers
# We only need otlp protocol on grpc, but you can use http, zipkin, jaeger, aws, etc.
# https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/receiver
receivers:
  otlp:
    protocols:
      grpc:

# Configure exporters
```



```

exporters:
  # Export prometheus endpoint
  prometheus:
    endpoint: "0.0.0.0:8889"

  # log to the console
  logging:

  # Export to zipkin
  zipkin:
    endpoint: "http://zipkin-all-in-one:9411/api/v2/spans"
    format: proto

  # Export to a file
  file:
    path: /etc/output/logs.json

# Configure processors (batch, sampling, filtering, hashing sensitive data, etc.)
# https://opentelemetry.io/docs/collector/configuration/#processors
processors:
  batch:

# Configure pipelines. Pipeline defines a path the data follows in the Collector
# starting from reception, then further processing or modification and finally
# exiting the Collector via exporters.
# https://opentelemetry.io/docs/collector/configuration/#service
# https://github.com/open-telemetry/opentelemetry-collector/blob/main/docs/design.md#pipeli
service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [logging, zipkin]
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [logging, prometheus]
    logs:
      receivers: [otlp]
      processors: []
      exporters: [logging, file]

```

▪ prometheus.yaml

YAML



```

scrape_configs:
- job_name: 'otel-collector'
  scrape_interval: 10s
  static_configs:
  - targets: ['otel-collector:8889']
  - targets: ['otel-collector:8888']

```

After creating those 3 files, you can start the services using the following command:

Shell



```
docker-compose up
```

Configuring the application to export data to the collector

To export data, you need to enable OpenTelemetry and configure which events to export.



First you need to add a few packages in your csproj:

XML

 copy

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="OpenTelemetry.Extensions.Hosting" Version="1.0.0-rc8" />
    <PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore" Version="1.0.0-rc8" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Http" Version="1.0.0-rc8" />
    <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol" Version="1.2.0-beta" />
    <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol.Logs" Version="1.0.0-rc8" />
  </ItemGroup>

</Project>
```

Then, you need to add the OpenTelemetry listeners for logs, traces, and metrics:

C#

 copy

```
using System.Diagnostics.Metrics;
using System.Diagnostics;
using OpenTelemetry.Metrics;
using OpenTelemetry.Trace;
using OpenTelemetry;
using OpenTelemetry.Logs;

// This is required if the collector doesn't expose an https endpoint. By default, .NET
// only allows http2 (required for gRPC) to secure endpoints.
AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);

var builder = WebApplication.CreateBuilder(args);

// Configure metrics
builder.Services.AddOpenTelemetryMetrics(builder =>
{
    builder.AddHttpClientInstrumentation();
    builder.AddAspNetCoreInstrumentation();
    builder.AddMeter("MyApplicationMetrics");
    builder.AddOtlpExporter(options => options.Endpoint = new Uri("http://localhost:4317"));
});

// Configure tracing
builder.Services.AddOpenTelemetryTracing(builder =>
{
    builder.AddHttpClientInstrumentation();
    builder.AddAspNetCoreInstrumentation();
    builder.AddSource("MyApplicationActivitySource");
    builder.AddOtlpExporter(options => options.Endpoint = new Uri("http://localhost:4317"));
});

// Configure logging
builder.Logging.AddOpenTelemetry(builder =>
{
    builder.IncludeFormattedMessage = true;
    builder.IncludeScopes = true;
    builder.ParseStateValues = true;
});
```




```

builder.AddOtlpExporter(options => options.Endpoint = new Uri("http://localhost:4317"));
});

var app = builder.Build();

// Create a route (GET /) that will make an http call, increment a metric and log a trace
var activitySource = new ActivitySource("MyApplicationActivitySource");
var meter = new Meter("MyApplicationMetrics");
var requestCounter = meter.CreateCounter<int>("compute_requests");
var httpClient = new HttpClient();

// Routes are tracked by AddAspNetCoreInstrumentation
// note: You can add more data to the Activity by using HttpContext.Activity.SetTag("key", "value")
app.MapGet("/", async (ILogger<Program> logger) =>
{
    requestCounter.Add(1);

    using (var activity = activitySource.StartActivity("Get data"))
    {
        // Add data the the activity
        // You can see these data in Zipkin
        activity?.AddTag("sample", "value");

        // Http calls are tracked by AddHttpClientInstrumentation
        var str1 = await httpClient.GetStringAsync("https://example.com");
        var str2 = await httpClient.GetStringAsync("https://www.meziantou.net");

        logger.LogInformation("Response1 length: {Length}", str1.Length);
        logger.LogInformation("Response2 length: {Length}", str2.Length);
    }

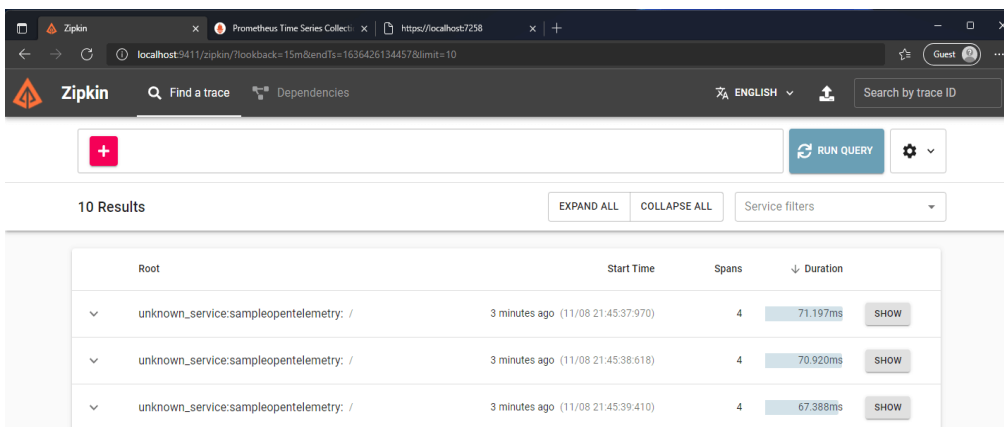
    return Results.Ok();
});

app.Run();

```

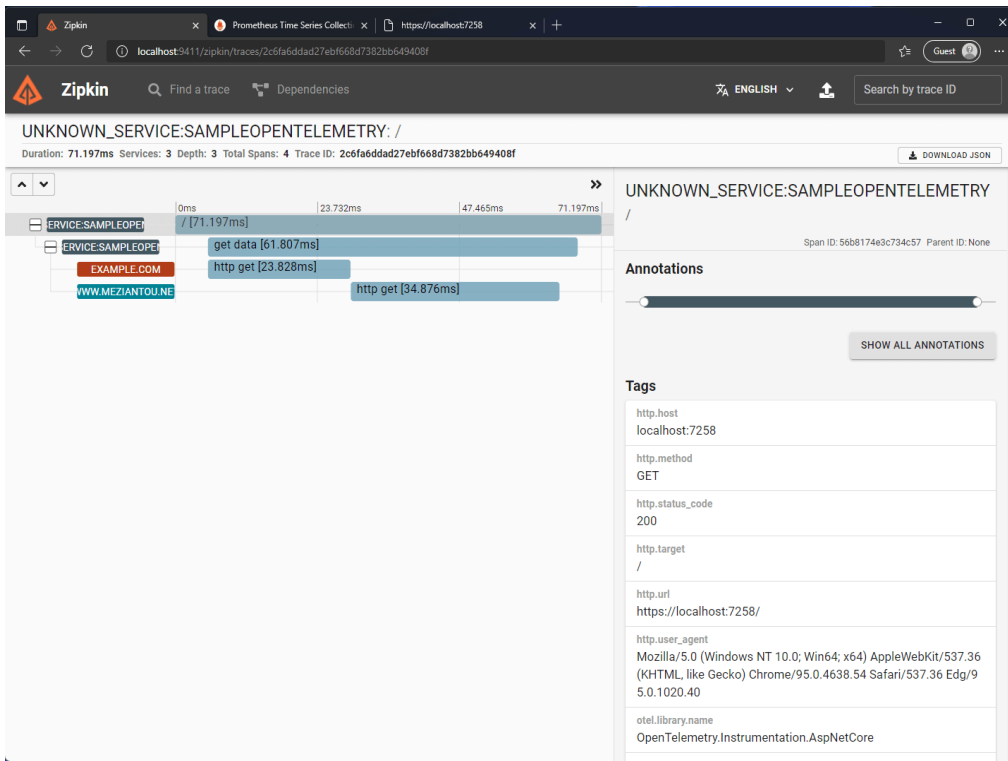
If you start the application using `dotnet run` and refresh the page a few times, you will see the following:

- The logs are available in `output/log.json`. The file contains one json payload by line. Each json contains all data from the log events.
- The traces are available in Zipkin <http://localhost:9411/>, including the tags created using `AddTag` :

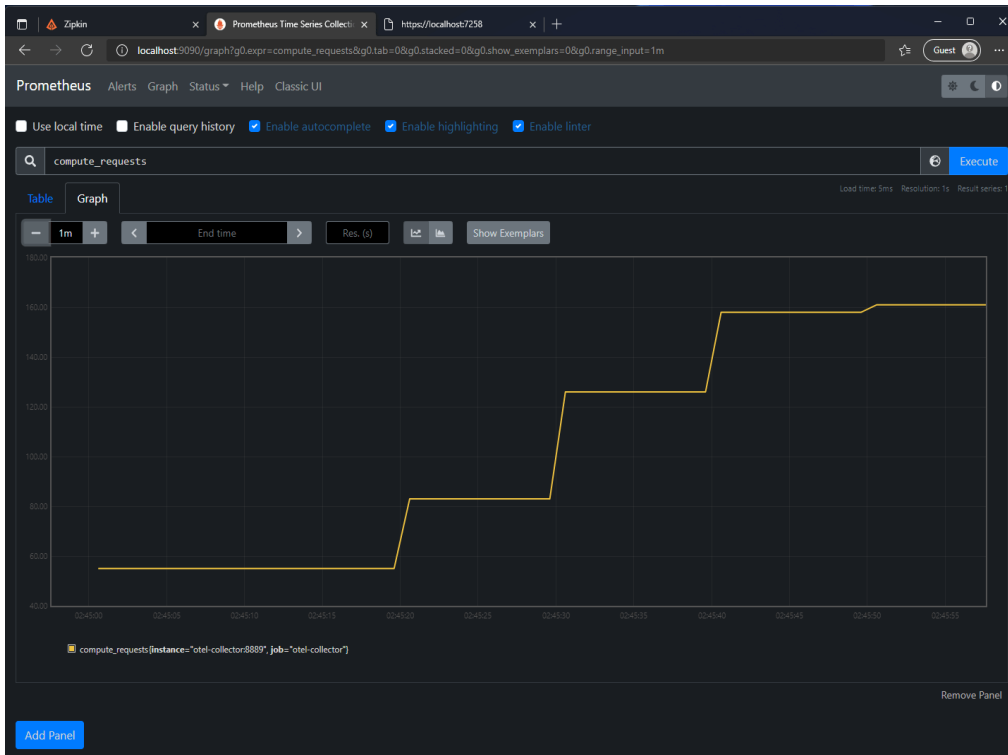


Root	Start Time	Spans	Duration
unknown_service:sampleopentelemetry: /	3 minutes ago (11/08 21:45:37.970)	4	71.197ms
unknown_service:sampleopentelemetry: /	3 minutes ago (11/08 21:45:38.618)	4	70.920ms
unknown_service:sampleopentelemetry: /	3 minutes ago (11/08 21:45:39.410)	4	67.388ms





- The metrics are available in Prometheus http://localhost:9090/graph?g0.expr=compute_requests&g0.tab=0&g0.stacked=0&g0.show_exemplars=0&g0.range_input=15m/

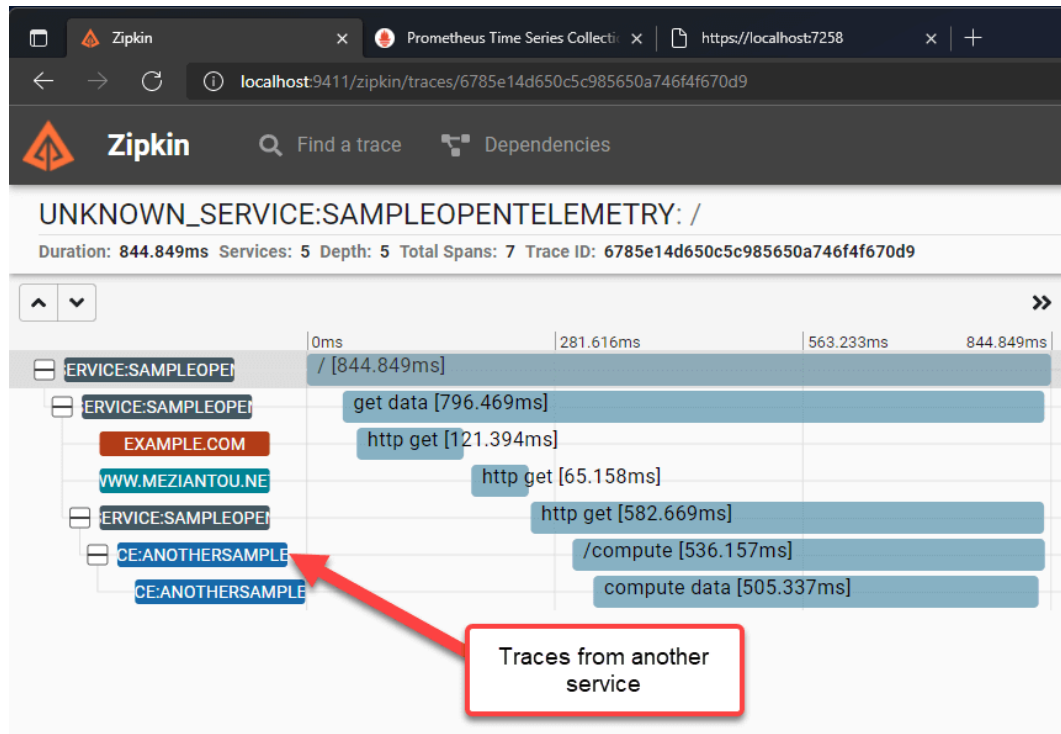
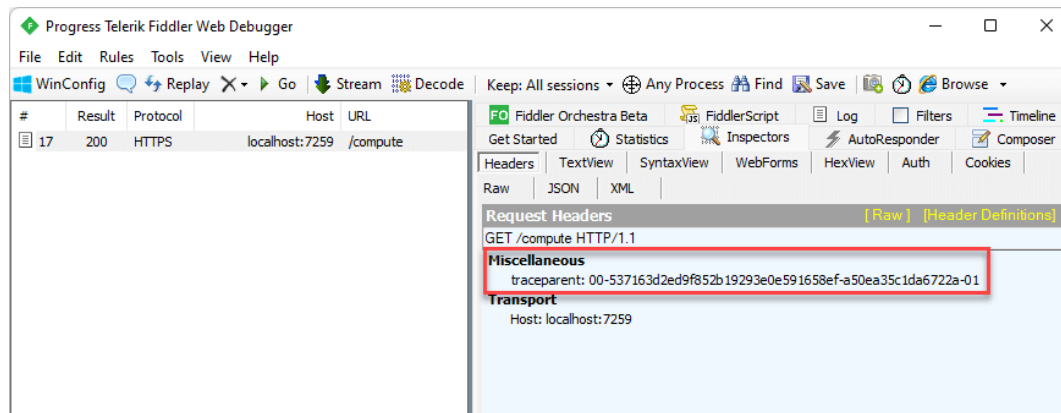


Your application is now monitored 😊

Monitoring multiple services



When you have multiple services, they often interact with each other using REST or gRPC. To track a request from end to end, you can use a correlation id. .NET automatically flows the current correlation id to the next service when using `HttpClient` (REST and gRPC) by using the `traceparent` header. This header is a [W3C standard](#) that allows tracing the request flow across services. This means that if you use the same back-end service to monitor all services, you can see the request flow across services:



You can customize the behavior of the `HttpClient` by using `SocketsHttpHandler.ActivityHeadersPropagator` :

```
C#  
  
using var handler = new SocketsHttpHandler()  
{  
    ActivityHeadersPropagator = DistributedContextPropagator.CreateDefaultPropagator(),  
};  
using var client = new HttpClient(handler);
```

You can check the default implementation on GitHub: [LegacyPropagator.cs](#)



Enriching the request activity created by ASP.NET Core

ASP.NET Core creates an activity for each request. You can enrich the activity by using the `Activity.SetTag` or `Activity.SetBaggage` methods. To do so, you need to access the Activity instance. This instance is available using the `IHttpActivityFeature` feature:

C#

 copy

```
app.MapGet("/", (HttpContext context) =>
{
    var activity = context.Features.Get<IHttpActivityFeature>()?.Activity;
    activity?.SetTag("foo", "bar");

    return Results.Ok();
});
```

If you are not in a Controller or a Minimal API delegate, you can use the `IHttpContextAccessor` to get the `HttpContext` :

C#








 copy

```
public class MyService
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public MyService(IHttpContextAccessor httpContextAccessor)
    {
        _httpContextAccessor = httpContextAccessor;
    }

    public void DoSomething()
    {
        var context = _httpContextAccessor.HttpContext;
        var activity = context.Features.Get<IHttpActivityFeature>()?.Activity;
        activity?.SetTag("foo", "bar");
    }
}
```

Additional resources

- [OpenTelemetry](#) 
- [OpenTelemetry .NET reaches v1.0](#) 
- [Metrics APIs Design](#) 
- [Libraries: Support for OpenTelemetry Metrics](#) 
- [Enable Azure Monitor OpenTelemetry Exporter for .NET, Node.js, and Python applications](#) 
- [Logging in .NET Core and ASP.NET Core](#) 
- [Semantic / Structured logging](#) 
- [Samples from this post](#)

Do you have a question or a suggestion about this post? [Contact me!](#)

Follow me:



Enjoy this blog?



**Earn an MIT
certificate**

Ad MIT Sloan

**Observability guide
for AWS**

Ad Elastic

**Leverage Enterprise
Data**

Ad Wyn Enterprise Solutions

**SQLmap SQL
Injection To
Basics**

Ad Cloud Academy

Copyright © 2022 Gérald Barré – Use of this site constitutes acceptance of our [Terms of use](#) and [Privacy policy](#).

