OpenTelemetry – Le novità di .Net 8

Andrea Merlin – Senior .Net Developer

Blog: https://amerlin.keantex.com

Twitter: @amerlin

Email: a.merlin@keantex.com

Intro

Telemetry and **monitoring** are two important techniques in software development to get more details about the workflow and resources used by the application during the execution

- Analyze issues in production where it could be difficult to debug the code.
- Get details about the interaction between components.
- Track duration time in processes to improve performance.
- Identify internal issues
- Collect information to analyze user behavior.

Intro

OpenTelemetry

There are many ways for monitoring and logging in .NET Applications.

We can use the following namespaces:

- Diagnostics.Trace
- Diagnostics.Debug
- Diagnostics.EventSource
- Diagnostics.Activity
- Diagnostics.EventLog (Only for Windows)

Intro

OpenTelemetry

- We can potentialize and extend all these features using libraries.

- In the following list you can find many options to use for logging and monitoring:
  - Serilog
  - Log4Net
  - Nlog
  - Elmah.io
  - Gelf4Net

Why Observability?

- To answer questions about the system
- Instrumentation produces data
- Querying data answers our questions
- Monitoring no longer can help us
- Microservices enviroment
- Instrumentation code is how we get telemetry
- Telemetry data can include traces, logs and/or metrics

What is OpenTelemetry

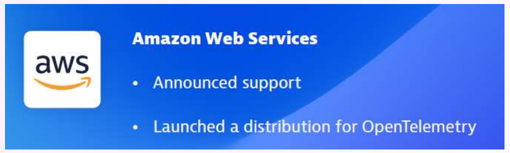
- OpenTracing
  - Provides APIS and instrumentation for distributed tracing
- OpenCensus
  - Provides APIS and instrumentation that allow you to collect application metrics and distributed tracing

OpenTelemetry

An effort to combine distributed tracing, metrics and logging into a single set of system components and language-specific libraries

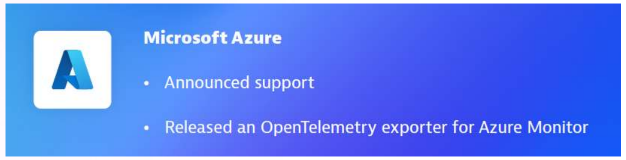
What is OpenTelemetry

- OpenTelemetry is an **open source** observability framework for cloud-native software that was created from the merger of the OpenTracing and OpenCensus projects.
- A Cloud Native Computing Foundation incubating project, OpenTelemetry is the **second most popular project behind Kubernetes**.
- OpenTelemetry provides a standard way to instrument, generate, collect, and **export telemetry data**, including metrics, logs, and traces.



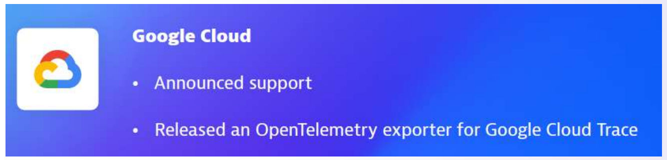
Amazon Web Services

- Announced support
- Launched a distribution for OpenTelemetry



Microsoft Azure

- Announced support
- Released an OpenTelemetry exporter for Azure Monitor

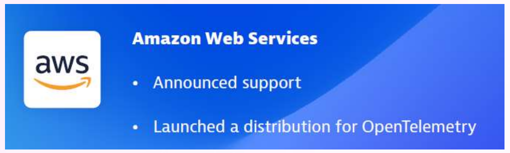


Google Cloud

- Announced support
- Released an OpenTelemetry exporter for Google Cloud Trace

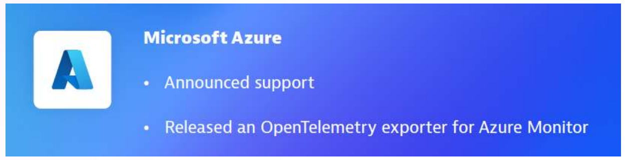
What is OpenTelemetry

- While it is a tool for capturing, transmitting, and parsing telemetry data, it does not provide back-end storage or analytics.
- A typical use case involves instrumenting applications with OpenTelemetry and then sending the resulting telemetry to a third-party observability solution for further analysis and visualization.



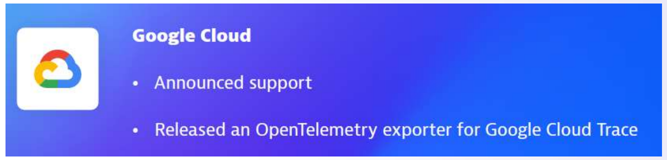
Amazon Web Services

- Announced support
- Launched a distribution for OpenTelemetry



Microsoft Azure

- Announced support
- Released an OpenTelemetry exporter for Azure Monitor



Google Cloud

- Announced support
- Released an OpenTelemetry exporter for Google Cloud Trace

Why OpenTelemetry?

- Tired of investigating zillions of logs
- Not willing to learn every dashboard of every team
- A trace us worth of thousand logs
- All services should participate in distributed tracing
- Services should be capable of collecting trace data automatically
- Telemetry backends can be changed without revising code
- OpenTelemetry protocol is used by APM servers

OpenTelemetry

- Official telemetry solution for all Microsoft frameworks
- Backwards compatible all the ways to .NET Standard 2.0
- Built into **System.Diagnostics.Activity and System.Diagnostics.Metrics**
- Wrapped via OpenTelemetry NuGet package
- High performance
- Easy integration
- Vendor agnostics

Before OpenTelemetry…

OpenTelemetry

- All vendors did their own thing
- Separate clients, SDKS, agents
- Wrapper around these SDK to form a common grammar
- Competing open standards

OPENTRACING

DATADOG

elastic

New Relic.

JAEGER

dynatrace

Before OpenTelemetry…

- Very difficult to migrate between APM solutions
- Performance overhead of APM varied wildly between implementations
- Impossibile for library / framework authors to reliably expose metrics / traces to users

OpenTelemetry architecture

Standard solution API

Before: Prior to OpenTelemetry, the process of capturing and transmitting application telemetry data was fragmented with a range of disparate open and vendor-provided solutions.

With OpenTelemetry: OpenTelemetry standardizes the telemetry API for metrics, logs, and traces. This open, standards-based approach enables broader adoption across a range of application development and operations teams. By providing a common approach to telemetry, it also enables collaboration between development and operations teams.

OpenTelemetry architecture

<u>Vendor agnostic</u>

Before: Traditional approaches to telemetry data collection required teams to manually instrument code, which locked users into specific tools. As a result, the cost of switching was high. So, if a customer adopted a solution that worked for monolithic apps but not cloud apps, moving to a more modern offering was challenging.

With OpenTelemetry: OpenTelemetry separates telemetry data collection from analytics. Because data is transmitted using OpenTelemetry's standards, customers can choose any back-end analytics platform, assuming it supports OpenTelemetry. This is particularly important as modern microservices-based applications have very different observability requirements than traditional monolithic architectures.

OpenTelemetry architecture

Pre-instrumented code

Before: Application tracing is critical to help users identify application issues and areas for improvement. Manually instrumenting code to export trace data can be challenging because it can require extensive testing cycles and code validation. With SaaS software, manual instrumentation may not even be possible.

With OpenTelemetry: OpenTelemetry provides a standard method for developers to expose tracing information from their software libraries and SaaS applications. As a result, users adopting these pre-instrumented libraries and applications can access OpenTelemetry telemetry data, which users can store, manage, and analyze in their observability platform.

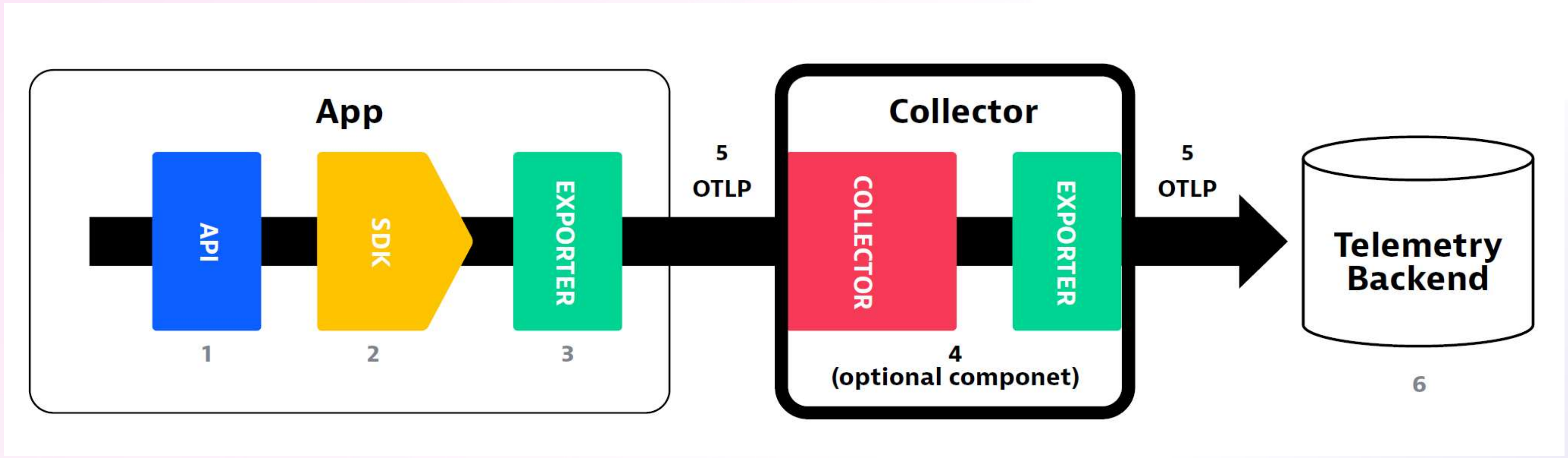
OpenTelemetry…weaknesses

- **Early stages**: since OpenTelemetry is still a relatively young project, some features may not be as mature or stable as other established tracing and monitoring tools.

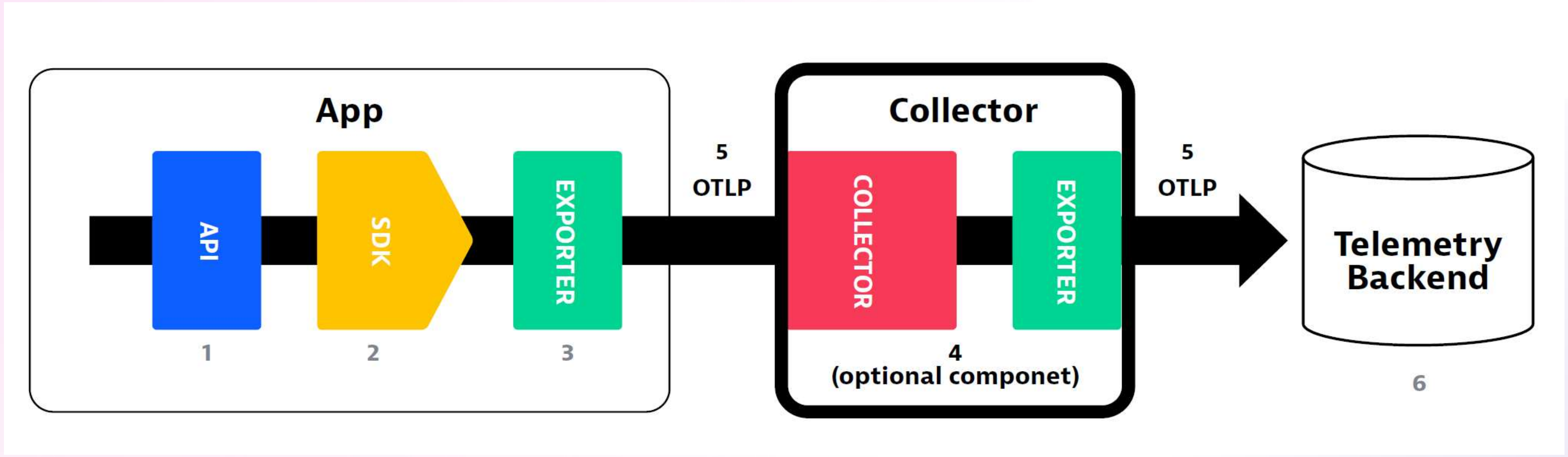
- **Limited auto-instrumentation**: While auto-instrumentation is available for some libraries, it’s less extensive than other platforms, like Java.

- **Learning curve**: Learning how to use OpenTelemetry effectively might take time, especially if you’re new to observability.

OpenTelemetry architecture



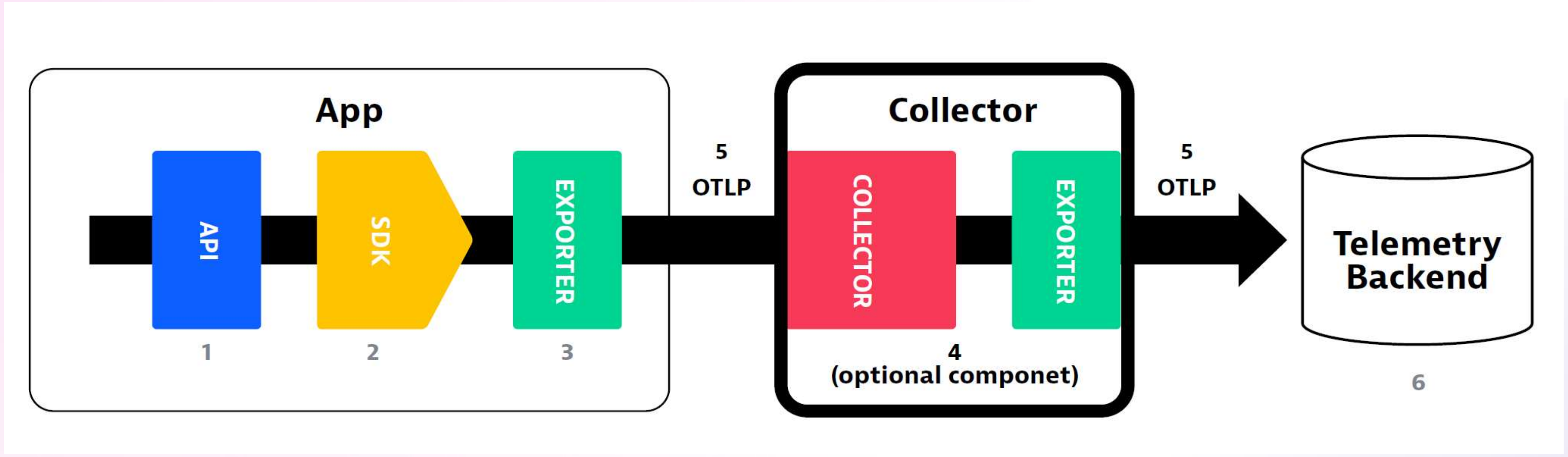
OpenTelemetry architecture



APIs are a core component of OpenTelemetry.

These are language-specific — Java, Python and .NET, for example — and provide the basic “plumbing” for instrumenting the code to generate telemetry data, such as metrics, logs, and traces.

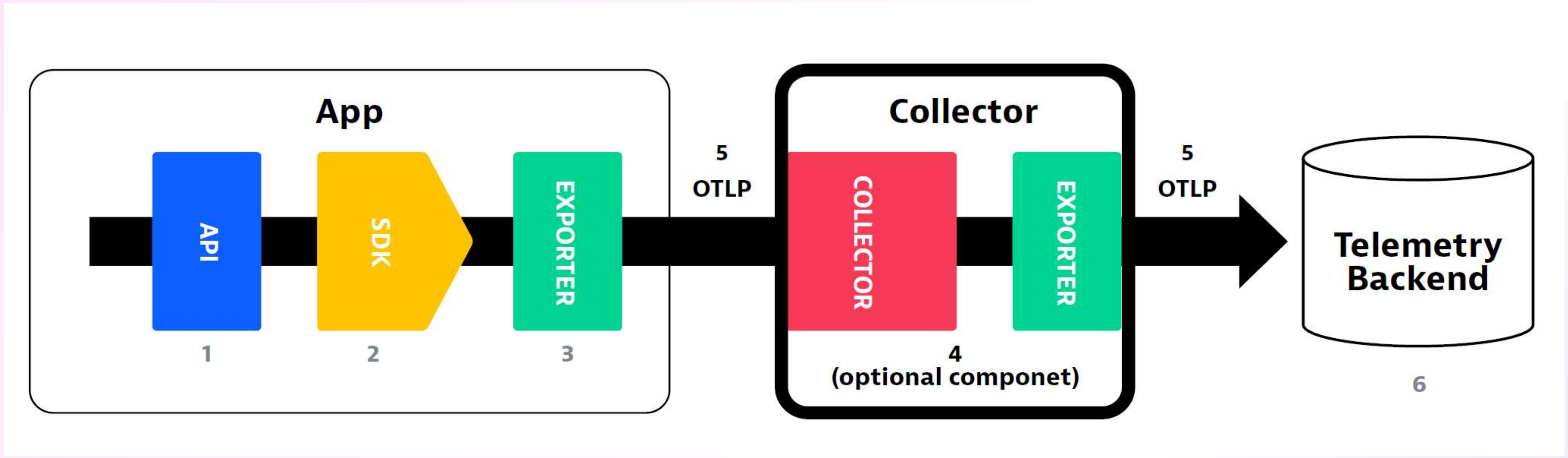
OpenTelemetry architecture



The SDKs are another language-specific component.

The implementation of the OpenTelemetry API serves as the bridge between the APIs and the exporter. This component allows for additional configuration, such as request filtering and transaction sampling.

OpenTelemetry architecture



Collector

The collector, while not technically required, is a useful component of the OpenTelemetry architecture.

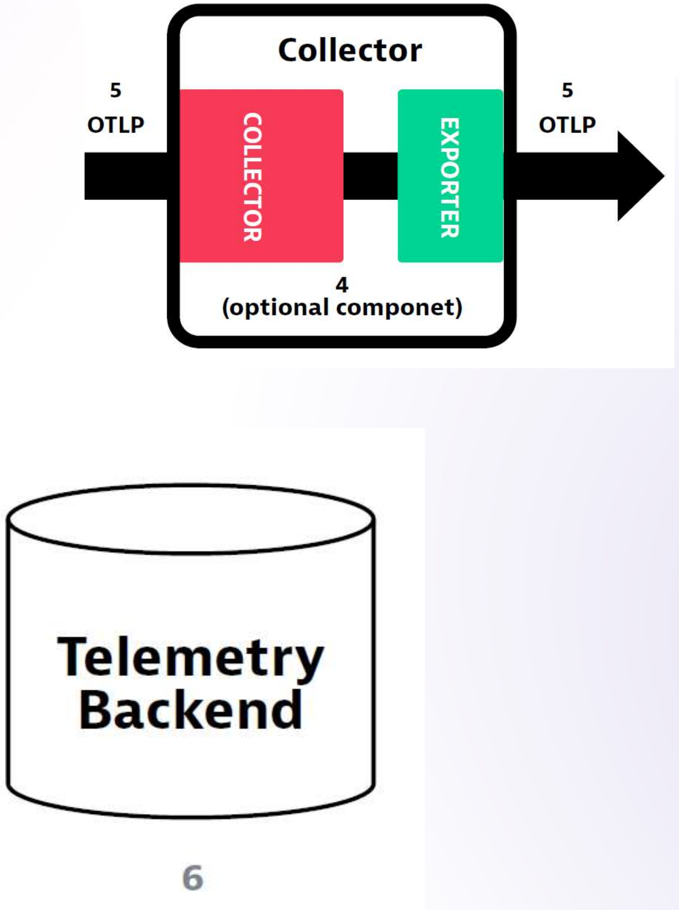
Allows greater flexibility for receiving, transforming, and sending the application telemetry to the back end(s)

OpenTelemetry architecture

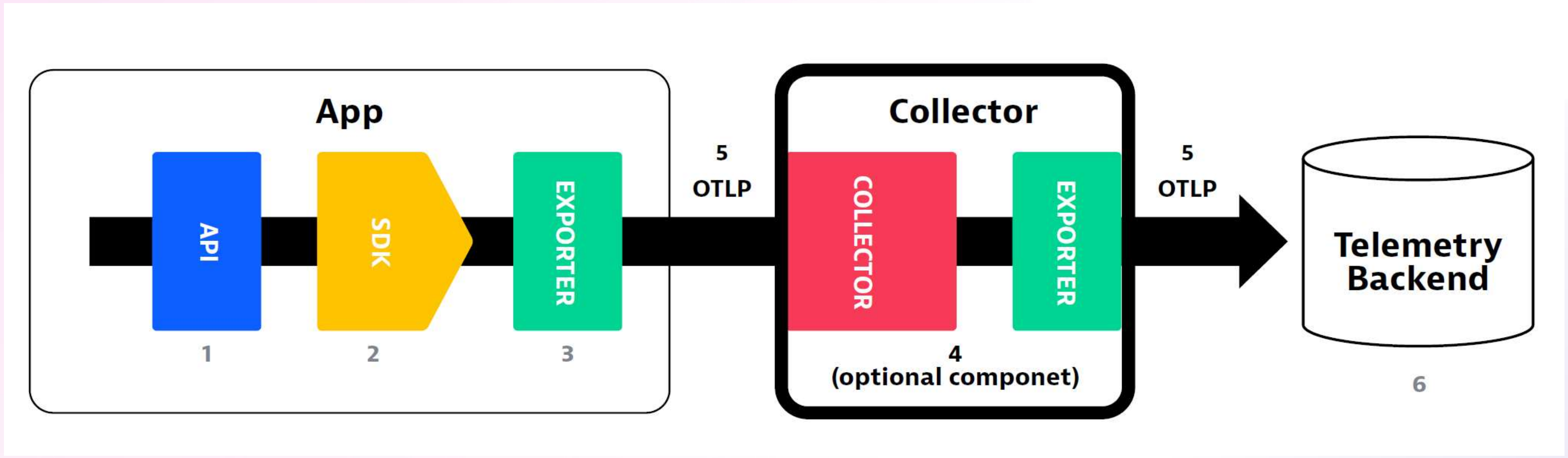
The collector has two deployment models:

- The first is an **agent** that resides on the same host as the application reporting data to it — binary, daemonset, or sidecar, for example. This collector can then send data to a server, operating system, database, or network directly or via another collector.

- The **second** is a **standalone process** (gateway) completely separate from the application(s) reporting telemetry data to it. It's responsible for exporting this data to a back-end observability tools



OpenTelemetry architecture

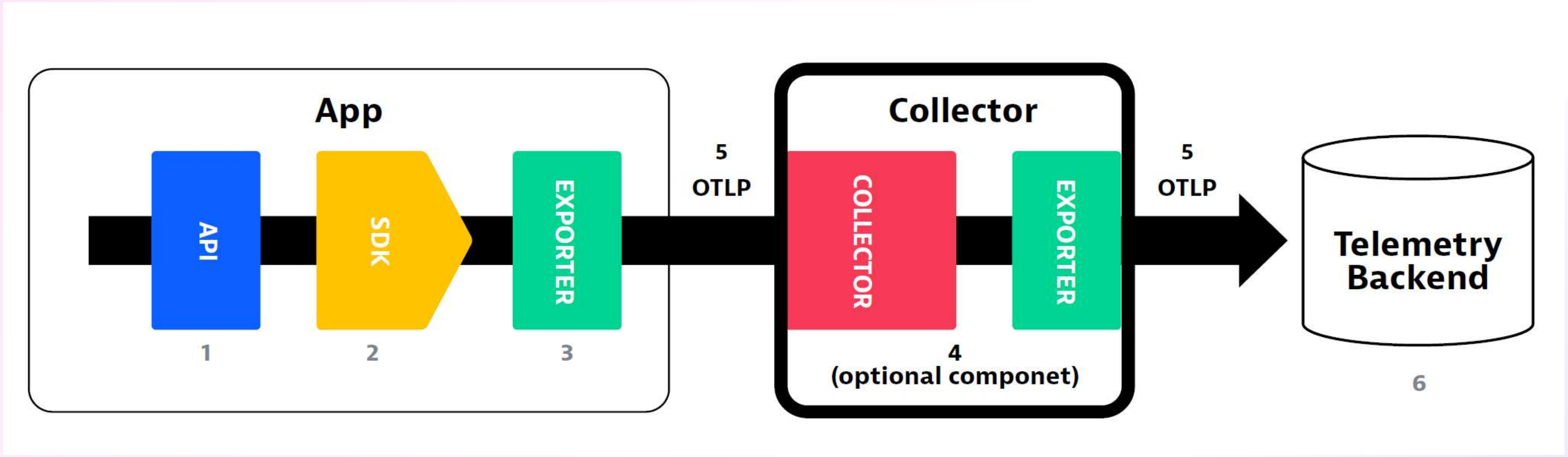


OpenTelemetry Protocol

OpenTelemetry Protocol (OTLP) is the OpenTelemetry native format that supports metrics, logs, and traces in a single data stream.

Exporters and collectors can be used to translate OTLP into the language of the back-end destination and then transport the data there.

OpenTelemetry architecture



Telemetry backend

OpenTelemetry Protocol (OTLP) is the OpenTelemetry native format that supports metrics, logs, and traces in a single data stream.

Exporters and collectors can be used to translate OTLP into the language of the back-end destination and then transport the data there.

OpenTelemetry architecture

<u>Instrumentation</u>

There are multiple ways to instrument code. The two most common approaches are the following:

- **Auto-instrumentation** An approach in which instrumentation is applied with no code changes using an agent

- **Manual instrumentation** An approach in which instrumentation requires code modification, such as source-code changes or adding new, pre-instrumented libraries

OpenTelemetry - Auto-Instrumentation

- **Auto-instrumentation** is a valuable feature of OpenTelemetry that enables the automatic collection of traces and metrics from your application without requiring manual code changes

- This means you don’t need to instrument your code to collect traces and metrics manually.

This can save you time and reduce the possibility of errors introduced through manual instrumentation

OpenTelemetry

OpenTelemetry - Auto-Instrumentation

- **OpenTelemetry.Instrumentation.AspNetCore**: For instrumenting ASP.NET Core applications.

- **OpenTelemetry.Instrumentation.Http**: For instrumenting outgoing HTTP requests made using HttpClient.

- **OpenTelemetry.Instrumentation.SqlClient**: For instrumenting SQL Server database calls made with System.Data.SqlClient or Microsoft.Data.SqlClient.

OpenTelemetry - Auto-Instrumentation

To enable auto-instrumentation, you must install the relevant NuGet packages, and :

```
using var tracerProvider = Sdk.CreateTracerProviderBuilder()
    .AddAspNetCoreInstrumentation()
    .AddHttpClientInstrumentation()
    .AddSqlClientInstrumentation()
    // ...
    .Build();
```

OpenTelemetry - Auto-Instrumentation

Auto-instrumentation can be configured to suit your needs.

For example, we can filter out specific requests, add custom tags to spans, or modify span names.

In this example, we filter out requests to the **/health** endpoint and add a custom tag to the spans.

OpenTelemetry - Auto-Instrumentation

```
using var tracerProvider = Sdk.CreateTracerProviderBuilder()
    .AddAspNetCoreInstrumentation(options =>
    {
        options.Filter = (httpContext) => httpContext.Request.Path != "/health";
        options.Enrich = (activity, eventName, obj) =>
        {
            if (eventName == "OnStartActivity")
            {
                activity.SetTag("custom-tag", "custom-value");
            }
        };
    })
    // ...
    .Build();
```

OpenTelemetry – Manual instrumentation

- While auto-instrumentation is convenient, it has some limitations. For example, it may only cover some libraries or frameworks you use and may not provide the granularity you need for some parts of your application.

- In such cases, you must implement custom instrumentation using the OpenTelemetry API.

- Remember that combining auto-instrumentation and custom instrumentation can help you achieve the desired level of observability for your .NET applications.

OpenTelemetry and Application Performance…

- Application Performance Monitoring is the tracking key software performance metrics using telemetry data
- Application Performance Monitoring can be used to validate compliance with the Service Level Objectives (SLO)
- Azure Monitor Tools
- Azure Aspire

OpenTelemetry core elements

- **Logs**: detailed debugging information emitted by processes
- **Traces**: trace requests to system
- **Metrics**: aggregated summary statistics
- **Baggage**: tags

OpenTelemetry VS .NET Terminology

OpenTelemetry	.NET
Tracer	ActivitySource
TelemetrySpan	Activity
SpanContext	ActivityContext

OpenTelemetry tracing

ActivitySource / Tracer : a source of tracing activity. Normal to have many per application

Activity / TelemetrySpan: represents a single logical unit of work recorded during a trace

ActivityContext / SpanContext: the propagation context used to correlate spans

Attributes: searchable properties of a span

Events: log events and other items recorded during an individual span

OpenTelemetry metrics

Meter : ActivitySource equivalent, but for metrics

Counter: monotonic counter used for measuring rates and frequencies

UpdownCounter: Similar to a counter, it can also decrease, allowing you to track values that may go up or down.

ValueRecorder: Used to record individual values and calculate statistics, like min, max, sum, and average.

ObservableGauge: async gauge that measures arbitrary values over time

Histogram: used to bucket counters into distributions

OpenTelemetry metrics

Setting Up Metrics Collection

To set up metrics collection in your .NET application, you'll need to follow these steps:

- Install the **OpenTelemetry.Metrics** NuGet package.
- Create a meter and define the metrics you want to collect.
- Configure the metrics export pipeline with your preferred exporter.

OpenTelemetry metrics

Creating a Meter and Defining Metrics

A meter is a factory for creating metric instruments.

```
using OpenTelemetry.Metrics;

var meter = new Meter("MyApp", "1.0.0");

var requestCounter = meter.CreateCounter<long>("requests_total", "Total
number of requests");
```

OpenTelemetry metrics…

Recording data

```
public class MyController : ControllerBase
{
    private static readonly Counter<long> RequestCounter = GlobalMeterProvider
        .GetMeter("MyApp", "1.0.0")
        .CreateCounter<long>("requests_total", "Total number of requests");

    public IActionResult Get()
    {
        RequestCounter.Add(1);
        // Your logic here
        return Ok();
    }
}
```

OpenTelemetry metrics…

Configuring the Metrics Export Pipeline

```
using OpenTelemetry.Exporter.Prometheus;
using OpenTelemetry.Metrics;

var metricsHost = "localhost";
var metricsPort = 9184;

var prometheusExporter = new PrometheusExporterOptions
{
   Url = $"http://{metricsHost}:{metricsPort}/metrics/",
};

Sdk.CreateMeterProviderBuilder()
   .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("MyApp"))
   .AddSource("MyApp")
   .AddPrometheusExporter(prometheusExporter)
   .Build();
```

OpenTelemetry tracing…

- **Span**
  - Single unit of work in a system
  - Typically encapsulates: operation name, start-finish timestamp, the parent span identifier, the span identifier and context items

- **Trace**
  - Defined implitily by its span. A trace can ben thought as a direct acyclic graphs of spans where the edges between spans are defined as parent/child relashionships

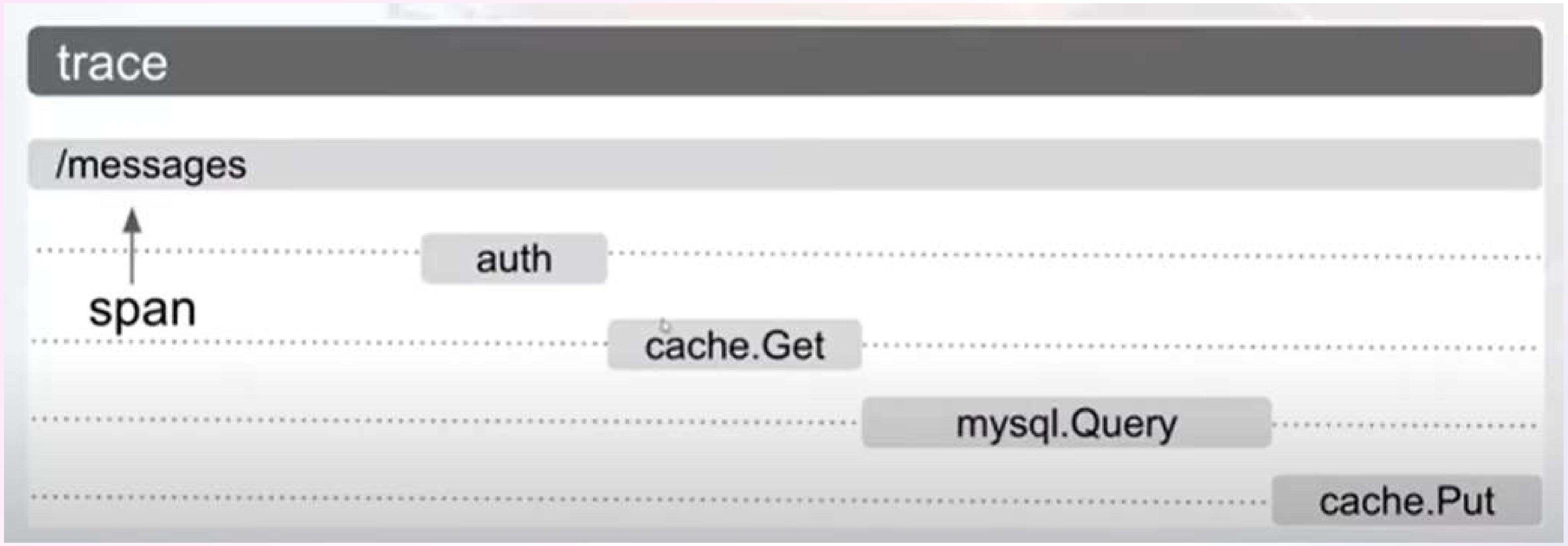
- **DistributedContext**
  - Contains tracing identifiers, tags and options that are propagated from parent to child spans

OpenTelemetry tracing…

Span: A span represents an action or operation that occurs within our system. This can include an HTTP request or a database operation that takes place over a specific duration of time. Spans often have a parent-child relationship with other spans.

Trace: Traces are essentially “call stacks” for distributed services, representing a tree of spans connected in a parent-child relationship. Traces outline the progression of requests across various services and components in our application (such as databases, data sources, queues, etc.)

OpenTelemetry tracing…



Manual and automatic span creation

- To create a new span: StartActivity method
- Span for configured instrumentations will be created automatically:
    - HTTP clients request
    - Incoming ASP.NET service calls
    - gRPC calls
    - SQL calls

Messaging frameworks may automate create of root spans

- NServiceBus: NServiceBus.Extensions.Diagnostics.OpenTelemetry
- Akka.NET: Phobos

Leaving trace span creation to messaging framework may result in too detailed collection ot traces and be unwanted in a distributed trace scenarios that involve multiple services

Add OpenTelemetry

- Add OpenTelemetry NuGet Package
- Add NuGet packages for each exporter you will use, e.g.:
  - **OpenTelemetry.Exporter.Console**
  - **OpenTelemetry.Exporter.Prometheus**
- Add NuGet Packages for each instrumentation, e.g.:
  - OpenTelemetry.Instrumentation.Http
  - OpenTelemetry.Instrumentation.SqlClient
- Activate exporters and instrumentations in boostrapper

Search OpenTelemetry exporter

- Check main OpenTelemetry repo at Github
  - Github.com/open-telemetry/opentelemetry-dotnet
  - Github.com/open-telemetry/opentelemetry-collector-contrib
- Check vendor's repository
  - **Github.com/mysql-net/mysqlconnector**
  - Github.com/datadog/dd-opentelemetry-exporter-XXX
- Write your own! ☺

OpenTelemetry Best Practices

- **Leverage auto-instrumentation**: Use the OpenTelemetry auto-instrumentation packages to simplify collecting data from your application. This approach can save you time and reduce the risk of misconfiguration.
- **Use semantic conventions**: Following the OpenTelemetry semantic conventions for span and attribute names helps ensure consistency across different services and makes it easier to analyze the data.
- **Avoid high cardinality attributes**: High cardinality attributes, like user IDs or timestamps, can lead to many unique spans, making it difficult to analyze data and increasing storage and processing costs.

OpenTelemetry Best Practices

- **Implement custom instrumentation**: For parts of your application not covered by auto-instrumentation, implement custom instrumentation using the OpenTelemetry API.

- **Monitor OpenTelemetry performance**: Keep an eye on the performance of your OpenTelemetry implementation, as it can impact your application's performance. In addition, use the OpenTelemetry metrics package to collect data on OpenTelemetry itself.

- **Leverage opentelemetry-dotnet-contrib**: The opentelemetry-dotnet-contrib repository contains additional instrumentation packages and exporters for .NET. Check it out to see if it offers additional value for your project..

OpenTelemetry package

- OpenTelemetry
- OpenTelemetry.Extensions.Hosting
- OpenTelemetry.Exporter.Zipkin (or your preferred exporter)

- OpenTelemetry.Instrumentation.AspNetCore
- OpenTelemetry.Instrumentation.HTTP
- OpenTelemetry.Instrumentation.SqlClient (if using SQL Server)

- OpenTelemetry.Metrics

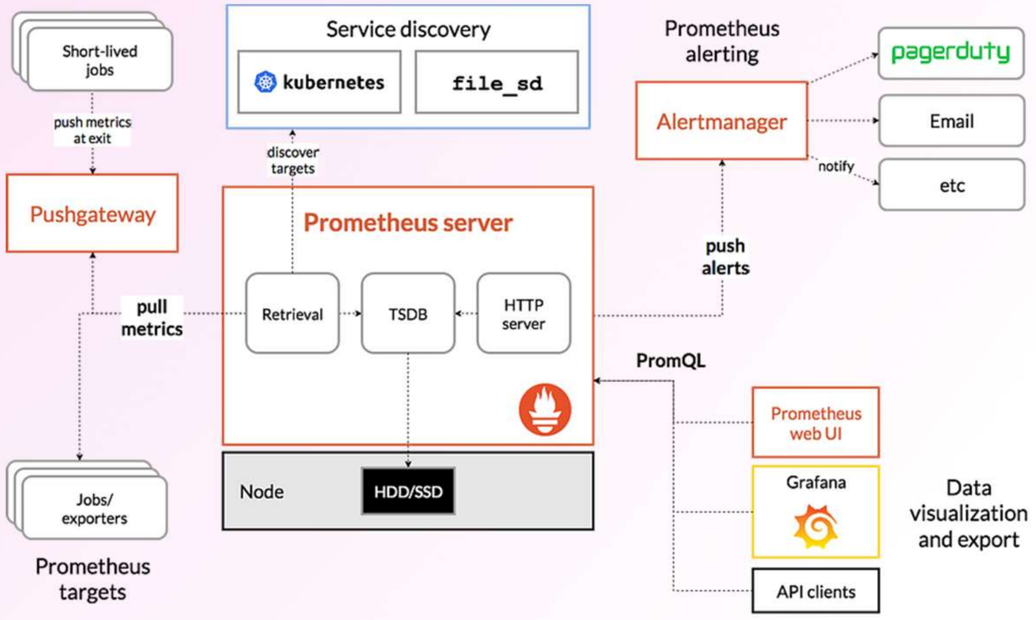
Prometheus

Prometheus is an open-source data monitoring system, that can collect telemetry data from our software system and helps us in monitoring this data.

At it’s core prometheus has the following things:

- Data collection agents aka jobs which periodically scrape raw metrics data from our application.
- A time series database for storing the scraped data.
- A HTTP Server for exposing this data.

Prometheus



Prometheus Architecture

Prometheus has it's own query language called PromQL for querying the metric data to extract useful insights

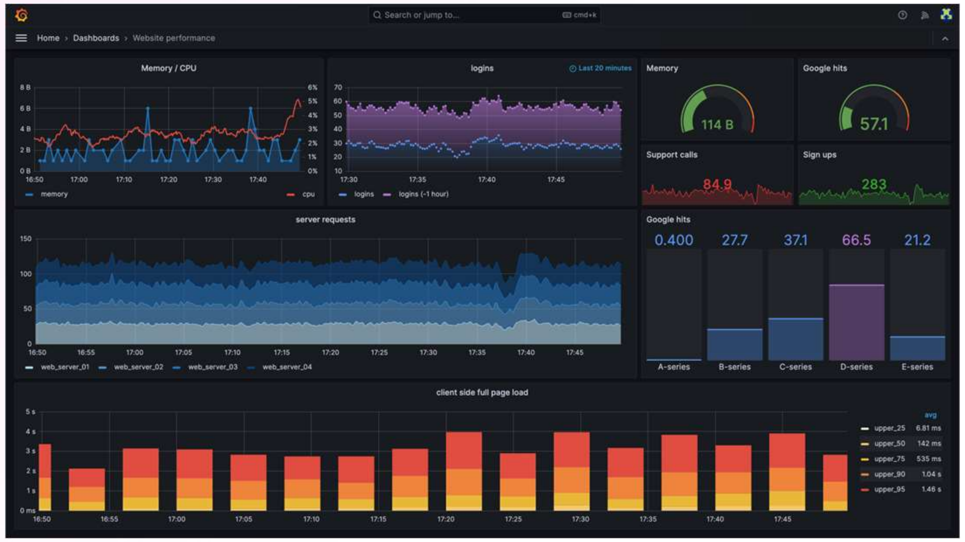
As powerful as PromQL syntax, to be honest it's not very friendly to be dealt by not so tech savvy folks.

Moreover we can't leverage the full power of these metrics unless we can visualize them.

Grafana

Grafana is an open source data visualization framework, that lets us build powerful dashboards to query data from various data source like Prometheus.

Apart from data visualization it also a highly configurable monitoring and alerting system

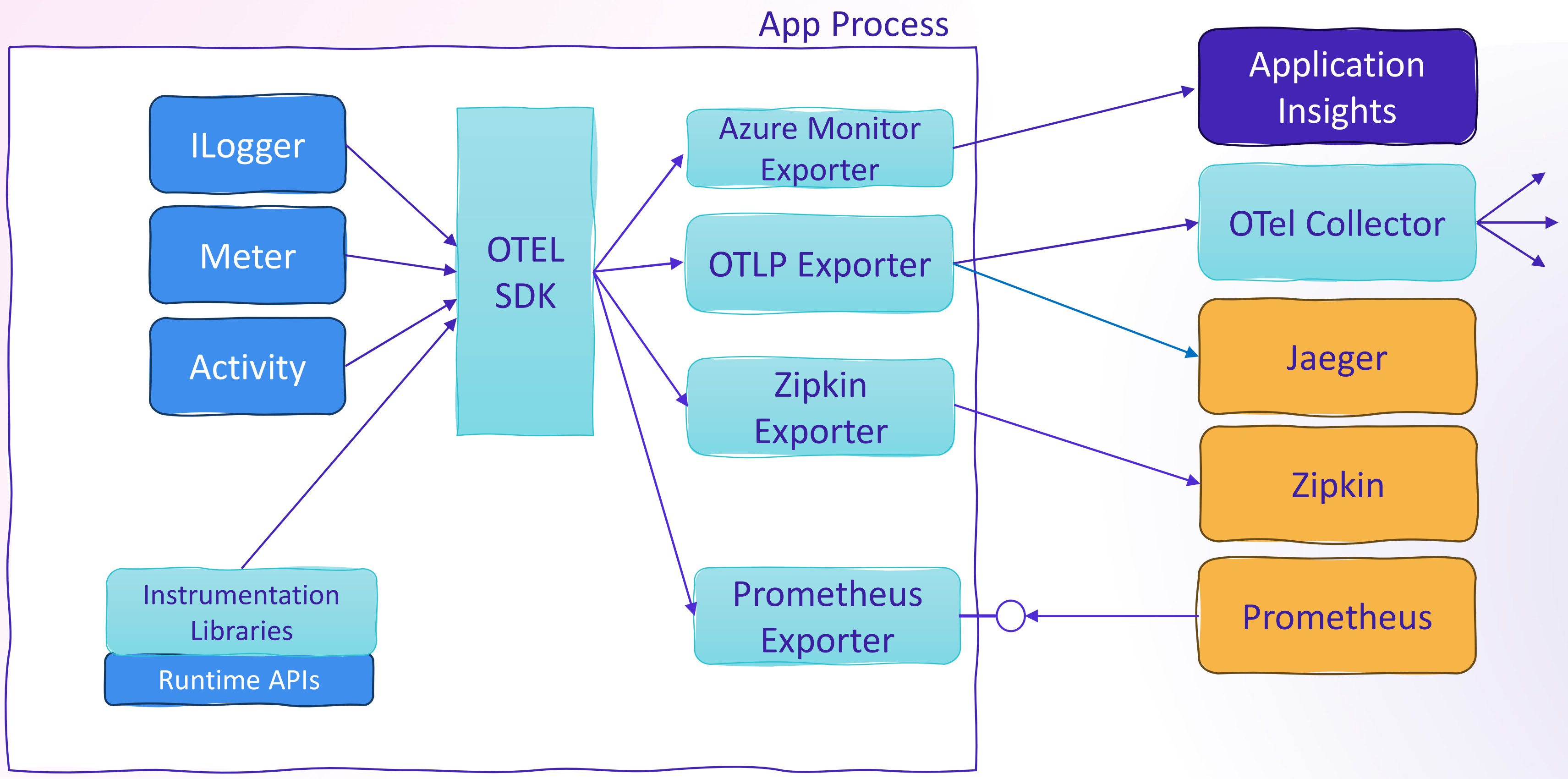


OpenTelemetry Demo

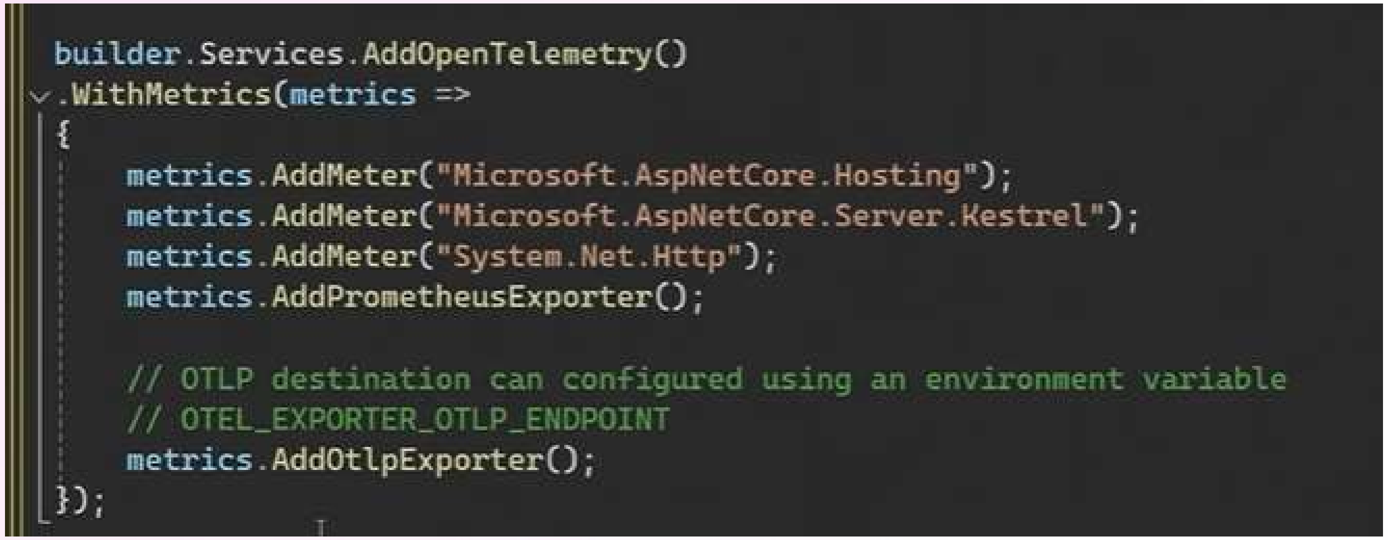
- Bootstrapping applications
- Adding exporters
- Adding instrumentations
- Adding telemetry sources
- Collecting traces in a single applications
- Collecting traces in a distributed scenario

Demo1
Demo2
Demo3
Demo4

.NET Observability Layer Diagram



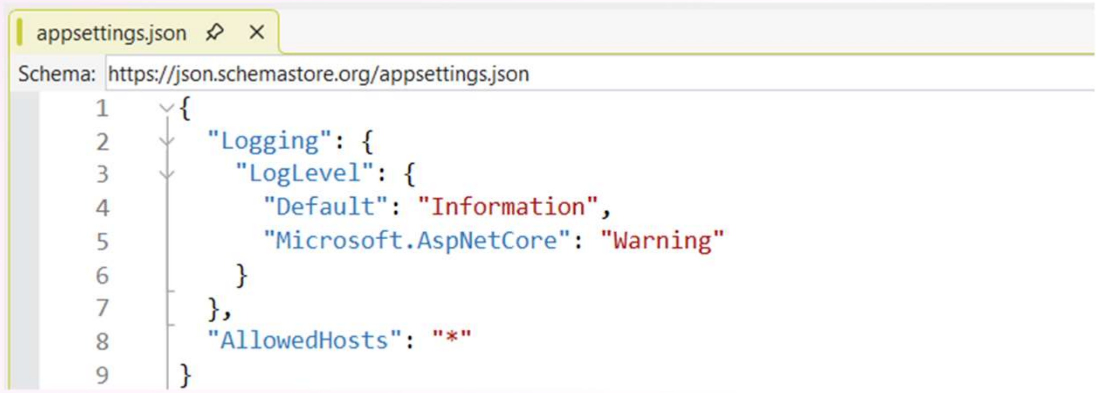
.NET Observability Layer Diagram



```
builder.Services.AddOpenTelemetry()
.WithMetrics(metrics =>
{
    metrics.AddMeter("Microsoft.AspNetCore.Hosting");
    metrics.AddMeter("Microsoft.AspNetCore.Server.Kestrel");
    metrics.AddMeter("System.Net.Http");
    metrics.AddPrometheusExporter();

    // OTLP destination can configured using an environment variable
    // OTEL_EXPORTER_OTLP_ENDPOINT
    metrics.AddOtlpExporter();
});
```

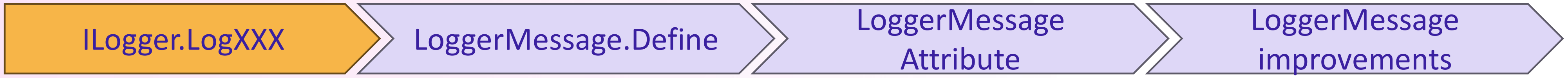
Logging with ILogger

- ILogger is an abstraction over multiple log sources
  - Usually fetched via dependency injection (DI)
  - Usually using ILogger<T> where T provides the name of the log source
- Sources are configured in code & configuration
  - Automatic in ASP.NET applications
  - Microsoft.Extensions.Logging for other Apps
- Multiple providers for exporting logs including console, OTel



```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Logging API Progression



ILogger.LogInformation, LogWarning, LogError, LogTrace, LogDebug

```
myLog.LogInformation("{count} items have been added to the cart", cart.items.Length);
```

Note: Parameters are separate from the message string

- Simple to use, but sub-optimal in high frequency applications
- Will check if logger is active before composing message, but…
  - Parameter expressions have to be calculated, ref values are boxed
- No stable IDs

Logging API Progression

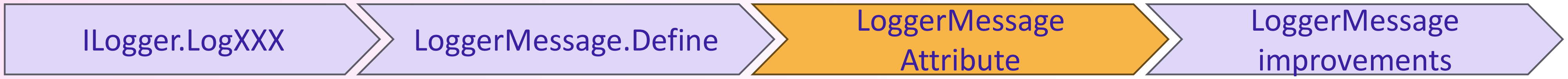
ILogger.LogXXX → **LoggerMessage.Define** → LoggerMessage Attribute → LoggerMessage improvements

- Use LoggerMessage.Define to create an action delegate that can then be called

```
private static readonly Action<ILogger, int> s_itemAddedToCart = LoggerMessage.Define(
    LogLevel.Information,
    new EventId(13, nameof(ItemAddedToCart)),
    “{count} items have been added to the shopping cart!");
```

- Action delegate parameters are strongly typed – no boxing
- Message is parsed once for parameters
- Event has a static Id

Logging API Progression



- LoggerMessageAttribute uses codegen to create logging delegate

```
[LoggerMessage( EventId = 13,
        Level = LogLevel.Critical,
        Message = "{count} items have been added to the shopping cart!")]
public static partial void ItemAddedToCart(ILogger logger, int count);
```

- Generates similar code to LoggerMessage.Define
- Cleaner syntax
- Automatically handles exception parameters
- Must be defined in a partial class
  - Options for how to access ILogger, pass in the log level

Logging API Progression

ILogger.LogXXX → LoggerMessage.Define → LoggerMessage Attribute → LoggerMessage parameters

- [LogProperties] flattens complex properties into individual parameters

```
record UserInfo(string Name, Guid Id);

[LoggerMessage(EventId = 13,
    Level = LogLevel.Critical,
    Message = "{count} items have been added to the shopping cart!")]
public static partial void ItemAddedToCart(ILogger logger, int count, [LogProperties] user);
```

Name & Id will be written as individual parameters

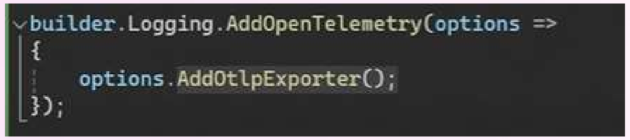
- Logs all public properties not marked with [LogPropertyIgnore]
- [TagProvider] can point to method for selecting which properties of complex objects should be logged

```
public static void LogUserInfoTags(ITagCollector collector, UserInfo? Item)
{
    collector.Add("Custom_tag_name", Item?.Id);
}
```

- IEnumerable and dictionaries will automatically be logged

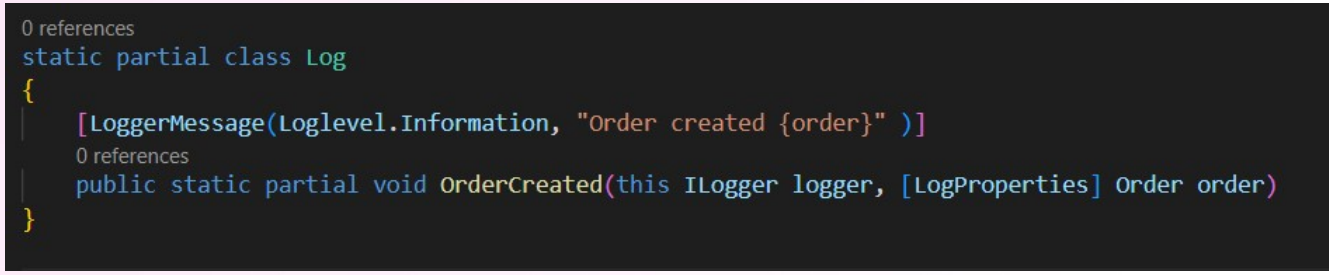
Packages: Microsoft.Extensions.Telemetry, Microsoft.Extensions.Telemetry.Abstractions

Logging API Progression

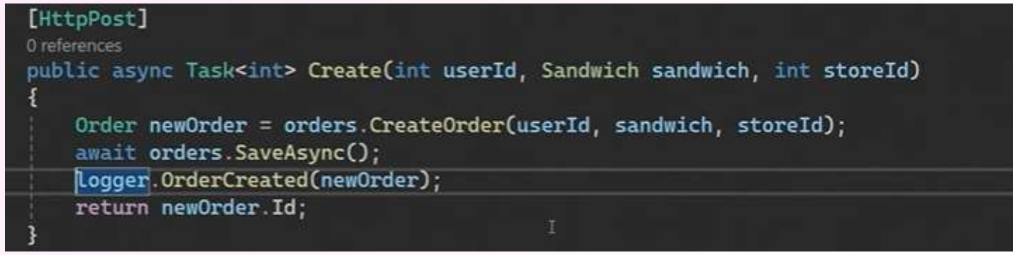


```
builder.Logging.AddOpenTelemetry(options =>
{
    options.AddOtlpExporter();
});
```

Microsoft.Extensions.Telemetry.Abstractions



```
0 references
static partial class Log
{
    [LoggerMessage(Loglevel.Information, "Order created {order}" )]
    0 references
    public static partial void OrderCreated(this ILogger logger, [LogProperties] Order order)
}
```



```
[HttpPost]
0 references
public async Task<int> Create(int userId, Sandwich sandwich, int storeId)
{
    Order newOrder = orders.CreateOrder(userId, sandwich, storeId);
    await orders.SaveAsync();
    logger.OrderCreated(newOrder);
    return newOrder.Id;
}
```

Metrics

- System.Diagnostics.Metrics API supports:
  - Instruments: Counter, Gauge, Histogram, observable variants
  - Key/Value tags add dimensions to metrics
- Observed through OpenTelemetry

- New to .NET 8
  - Built-in metrics for ASP.NET Core & HttpClient
  - IMeterFactory
  - Testing Fake for Meter
    - Microsoft.Extensions.Diagnostics.Metrics.Testing.MetricCollector

Built-in Metrics in .NET 8

- Microsoft.AspNetCore.Server.Kestrel
  - kestrel.active_connections
  - kestrel.connection.duration
  - kestrel.rejected_connections
  - kestrel.queued_connections
  - kestrel.queued_requests
  - kestrel.upgraded_connections
  - kestrel.tls_handshake.duration
  - kestrel.active_tls_handshakes
- Microsoft.AspNetCore.Http.Connections
  - signalr.server.connection.duration
  - signalr.server.active_connections
- Microsoft.AspNetCore.Hosting
  - http.server.request.duration
  - http.server.active_requests
- Microsoft.AspNetCore.Routing
  - aspnetcore.routing.match_attempts
- Microsoft.AspNetCore.Diagnostics
  - aspnetcore.diagnostics.exceptions
- Microsoft.AspNetCore.RateLimiting
  - aspnetcore.rate_limiting.active_request_leases
  - aspnetcore.rate_limiting.request_lease.duration
  - aspnetcore.rate_limiting.queued_requests
  - aspnetcore.rate_limiting.request.time_in_queue
  - aspnetcore.rate_limiting.requests
- System.Net.NameResolution
  - dns.lookup.duration
- System.Net.Http
  - http.client.open_connections
  - http.client.connection.duration
  - http.client.request.duration
  - http.client.request.time_in_queue
  - http.client.active_requests

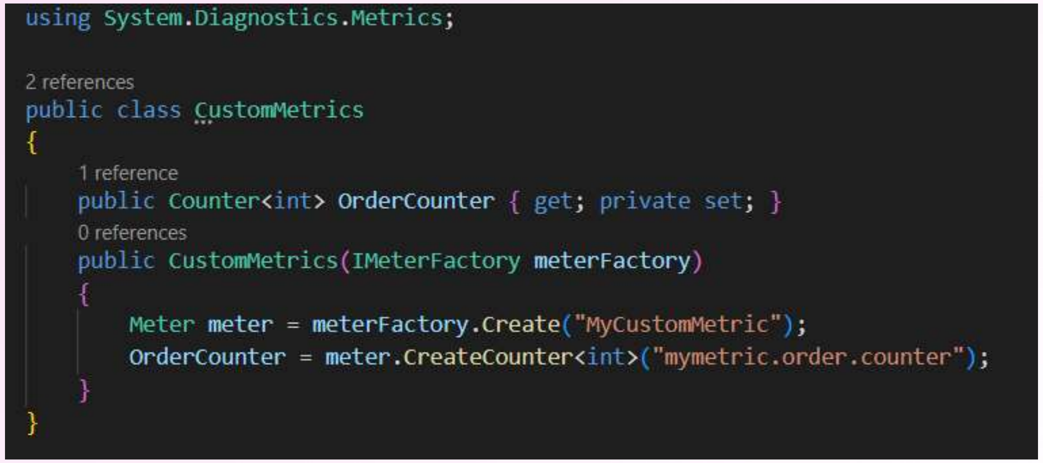
IMeterFactory

- IMeterFactory Adds DI to the way an app can define and use Meters

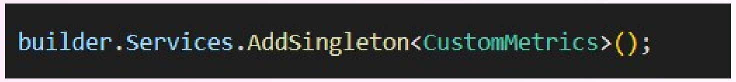
```
class Orders
{
    private readonly ILogger<Orders> _logger;

    private readonly Counter<int> _ordersCounter;

    public Orders(ILogger<Orders> logger, IMeterFactory meterFactory)
    {
        _logger = logger;
        var _meter = meterFactory.Create("MyApp.Orders");

        _ordersCounter = _meter.CreateCounter<int>("myapp.orders.count", "processed orders");
    }
}
```

- MetricCollector enables unit testing
- Can define static tags/dimensions for Meters & Instruments at creation time

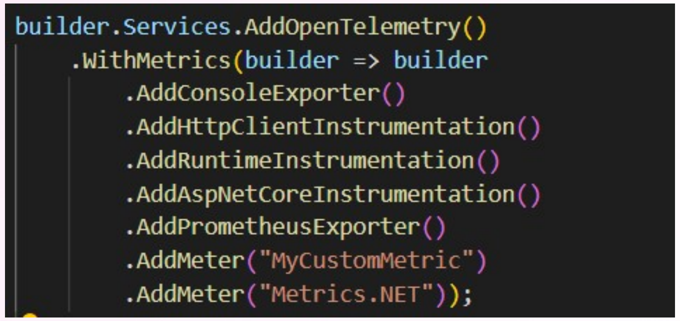
IMeterFactory



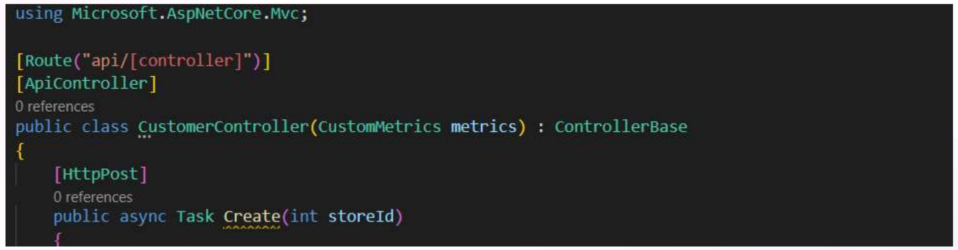
```
using System.Diagnostics.Metrics;

2 references
public class CustomMetrics
{
    1 reference
    public Counter<int> OrderCounter { get; private set; }
    0 references
    public CustomMetrics(IMeterFactory meterFactory)
    {
        Meter meter = meterFactory.Create("MyCustomMetric");
        OrderCounter = meter.CreateCounter<int>("mymetric.order.counter");
    }
}
```



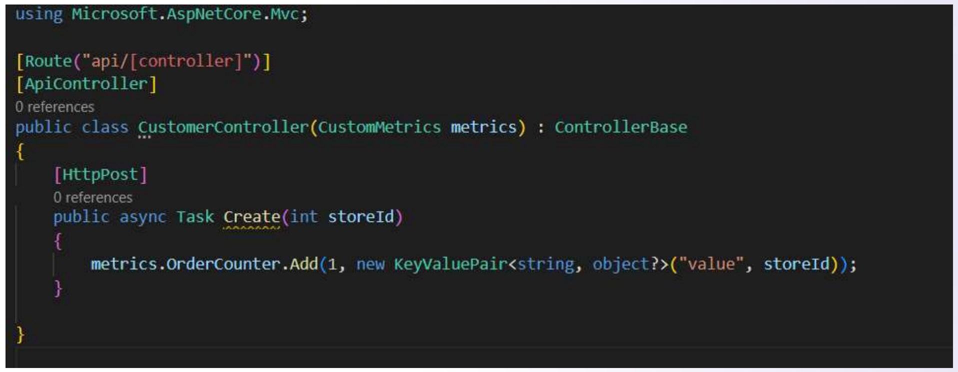
```
builder.Services.AddSingleton<CustomMetrics>();
```



```
builder.Services.AddOpenTelemetry()
    .WithMetrics(builder => builder
        .AddConsoleExporter()
        .AddHttpClientInstrumentation()
        .AddRuntimeInstrumentation()
        .AddAspNetCoreInstrumentation()
        .AddPrometheusExporter()
        .AddMeter("MyCustomMetric")
        .AddMeter("Metrics.NET"));
```



```
using Microsoft.AspNetCore.Mvc;

[Route("api/[controller]")]
[ApiController]
0 references
public class CustomerController(CustomMetrics metrics) : ControllerBase
{
    [HttpPost]
    0 references
    public async Task Create(int storeId)
    {
```



```
using Microsoft.AspNetCore.Mvc;

[Route("api/[controller]")]
[ApiController]
0 references
public class CustomerController(CustomMetrics metrics) : ControllerBase
{
    [HttpPost]
    0 references
    public async Task Create(int storeId)
    {
        metrics.OrderCounter.Add(1, new KeyValuePair<string, object?>("value", storeId));
    }

}
```

Resources

Download .NET 8
aka.ms/get-dotnet-8

.NET OpenTelemetry Repo
https://github.com/open-telemetry/opentelemetry-dotnet

Open Telemetry on Nuget
https://www.nuget.org/profiles/OpenTelemetry

Grafana Dashboard
https://github.com/dotnet/aspire/tree/main/src/Grafana

Resources

Grazie!!!

Questionario

https://dotnetliguria.azurewebsites.net/