**OBSERVABILIDAD**

* **Componentes**
  1. *OpenTelemetry Collector*.- Recibe, procesa y exporta los datos de telemetría desde las aplicaciones hacia los backends de observabilidad (logs 🡪 *Loki*, métricas 🡪 *Prometheus* , trazas 🡪*Jaeger* ó *Zipkin*). Todo este flujo se conoce como Collector Pipeline y se define en archivos yml.
  2. *Prometheus*.- Recopila y almacena métricas con la marca de tiempo en la que se registró junto con pares-valor llamados etiquetas.
  3. *Loki*.- sistema de logs de alta disponibilidad y fácil de operar, trabaja con el agente *Promtail* que será el responsable de recopilar y enviar los logs a *Loki*.
  4. *Jaeger*.- herramienta para facilitar el seguimiento de extremo a extremo (una traza) de cada petición. Monitoreo de transacciones distribuidas, optimización de rendimiento y latencia, análisis de causa y dependencia.
  5. *Zipkin*.- análisis de traza o seguimiento. Recopila datos de temporización para solucionar problemas de latencia en arquitecturas de microservicios.
  6. Grafana.- plataforma de observabilidad que permite crear cuadros de mando y gráficos a partir de varias fuentes de datos.
* **Instrumentación de aplicaciones .NET**

Habilitar la capacidad de emitir datos de telemetría en la aplicación. Necesitamos instalar los siguientes paquetes Nuget:

1. *OpenTelemetry*
2. *OpenTelemetry.Api*
3. *OpenTelemetry.Exporte.Console*
4. *OpenTelemetry.Exporter.OpenTelemetryProtocol*
5. *OpenTelemtry.Extensions.Hosting*
6. *OpenTelemetry.Instrumentation.AspNetCore (recopila métricas de peticiones http)*
7. *OpenTelemetry.Instumentation.Http*
8. *OpenTelemetry.Instrumentation.Process (métricas de proceso)*
9. *OpenTelemetry.Instrumentation.Runtime (métricas de rendimiento)*

Para comenzar configuramos únicamente las trazas: (posteriormente haremos registros y métricas)

En nuestra clase Progam configuramos el uso de OpenTelemtry añadiendo:

En appsetting definimos la otlURL = “<http://localhost:4317> “

(puerto 4317 protocolo gRPc habitual en instrumentación manual,

puerto 4318 protocolo HTTP habitual en instrumentación automática)

*builder.Services.AddOpenTelemetry()*

*//Definimos el nombre del servicio como origen de los datos*

*.ConfigureResource(resource => resource.AddService(serviceName))*

***.WithTracing(tracing => tracing***

*.AddAspNetCoreInstrumentation()*

*.AddHttpClientInstrumentation()*

*.AddConsoleExporter()*

*//Indicamos la fuente de trazabilidad que utilizaremos (ActivitySource)*

*//private static readonly ActivitySource \_activitySource = new("Tracing.ApiMongoDB");*

*//Para agrupar y filtrar trazas*

*.AddSource($"Tracing.{serviceName}")*

*.AddOtlpExporter(otl =>{ otl.Endpoint = new Uri(otlURL);})*

*);*

Ahora se pueden añadir trazas en nuestro código utilizando ActivitySource donde, además, tenemos la posibilidad de añadir etiquetas (AddTag) y eventos (AddEvent). Esto lo haremos con StartActivity……

Una vez configurada nuestra aplicación es posible revisar el resultado en Jaeger, lo haremos creando un contenedor:

*docker run -d --name jaeger -p 16686:16686 -p 4317:4317 jaegertracing/all-in-one:latest*

Solo nos queda ejecutar nuestra aplicación y realizar peticiones, posteriormente si buscamos desde <http://localhost:16686> el servicio definido veremos el resultado.

Para configurar las métricas añadimos el siguiente código:

*.WithMetrics(metrics => metrics*

*.AddMeter(Instrumentor.ServiceName)*

*.ConfigureResource(resource => resource*

*.AddService(Instrumentor.ServiceName))*

*.AddRuntimeInstrumentation()*

*.AddAspNetCoreInstrumentation()*

*.AddProcessInstrumentation()*

*.AddHttpClientInstrumentation()*

*.AddEventCountersInstrumentation(c =>*

*{*

*c.AddEventSources(*

*"Microsoft.AspNetCore.Hosting",*

*"Microsoft-AspNetCore-Server-Kestrel",*

*"System.Net.Http",*

*"System.Net.Sockets");*

*})*

*.AddOtlpExporter(otl => { otl.Endpoint = new Uri(otlURL); })*

*);*

Y para capturar logs:

*builder.Logging.AddOpenTelemetry(options =>*

*{*

*options*

*.SetResourceBuilder(*

*ResourceBuilder.CreateDefault()*

*.AddService(Instrumentor.ServiceName))*

*.AddConsoleExporter();*

*options.AddOtlpExporter(otl => { otl.Endpoint = new Uri(otlURL); });*

*});*

Una buena idea puede ser configurar una clase donde centralizar nuestros objetos de instrumentación (ActivitySource, Meter y Logger) para monitorear la aplicación. Esto nos brinda un código limpio y mantenible y nos permite reutilizar código:

*public sealed class Instrumentor : IDisposable*

*{*

*public const string ServiceName = "ApiMongoDB";*

*public ActivitySource Tracer { get; }*

*public Meter Recorder { get; }*

*public Counter<long> IncomingRequestCounter { get; }*

*public Counter<long> PersonReadersCounter { get; }*

*private static readonly Action<ILogger, string, Exception> \_loggerError =*

*LoggerMessage.Define<string>(*

*LogLevel.Error,*

*new EventId(1, "ERROR"),*

*"{Message}");*

*private static readonly Action<ILogger, string, Exception> \_loggerInformation =*

*LoggerMessage.Define<string>(*

*LogLevel.Information,*

*new EventId(1, "INFORMATION"),*

*"{Message}");*

*public Instrumentor()*

*{*

*var version = typeof(Instrumentor).Assembly.GetName().Version?.ToString();*

*Tracer = new ActivitySource(ServiceName, version);*

*Recorder = new Meter(ServiceName, version);*

*IncomingRequestCounter = Recorder.CreateCounter<long>(*

*"ApiMongo.incoming.requests", "requests",*

*description: "The number of incoming requests to the backend API");*

*PersonReadersCounter = Recorder.CreateCounter<long>(*

*"ApiMongo.Person.Gets", "requests",*

*description: "The number of times a person was read from the database");*

*}*

*public void LogMessageError(ILogger logger, string message) =>*

*\_loggerError(logger, message, null);*

*public void LogMessageInformation(ILogger logger, string message) =>*

*\_loggerInformation(logger, message, null);*

*public void Dispose()*

*{*

*Tracer.Dispose();*

*Recorder.Dispose();*

*}*

*}*

Ejemplo completo en proyecto ApiMongoDB