Integrating OpenTelemetry & Security in eShop
Assignment 02.02 - 2024/2025

Roberto Rolão de Castro
107133

1. Introduction

1.1 Objectives of the Assignment

The main objective of this project was to successfully integrate *OpenTelemetry* and *Security* into an eShop Application. This involded implementing tracing and metrics collections to monitor the system's performance and detect potential problems early.

For this I implemented **eight** metrics and traces specifically for the *Place an Order* feature. Additionally, sensitive data, such as user information or payment details, were masked in logs to ensure compliance with security best practices. A ***Grafana*** dashboard was also implemented to provide enhanced visualization of these metrics, offering valuable insights into system performance and operational health.

1.2 Useful Links

- GitHub Repository: `https://github.com/RobertoCastro391/eShop_AS_02`

2. Metrics and Tracing Configuration

To successfully export metrics to ***Prometheus*** and traces to ***Jaeger*** several configuration changes were necessary. The files modified were:

- ***eShop.ServiceDefaults/Extensions.cs*** - Enables OpenTelemetry instrumentation and Prometheus metrics scraping in the application.

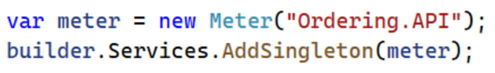
- ***otel-collector-config.yml*** - Defines the OpenTelemetry Collector configuration.

- ***prometheus.yml*** - Configures Prometheus to scrape metrics from the OpenTelemetry Collector.

- ***docker-compose.yml*** - Defines the deployment setup for OpenTelemetry Collector, Prometheus, and Jaeger.

To being sure that both metrics and traces are exported to *OpenTelemetry*, each microservice's *Program.cs* was modified as follow (*Github File Link*):

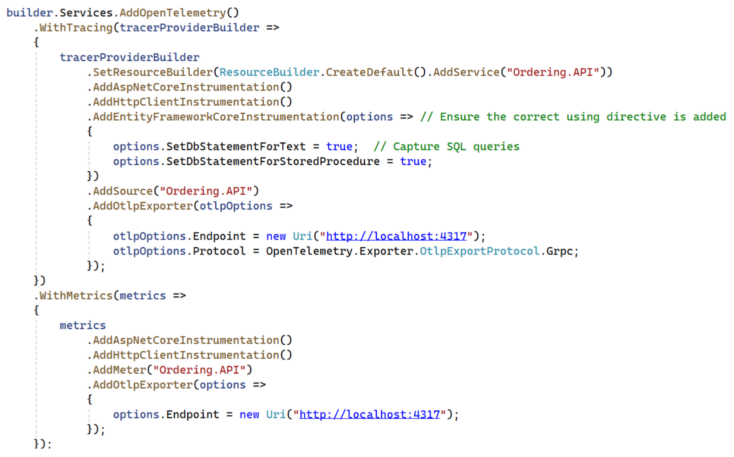
- Creating a Meter Instance for Metrics Collection:



```
var meter = new Meter("Ordering.API");
builder.Services.AddSingleton(meter);
```

Figure 2.1: Meter Instance Creation in *Ordering.API*

- Configuration of *OpenTelemetry* Instrumentation, enable Traces and Metrics:



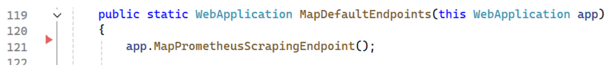
```
builder.Services.AddOpenTelemetry()
    .WithTracing(tracerProviderBuilder =>
    {
        tracerProviderBuilder
            .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("Ordering.API"))
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddEntityFrameworkCoreInstrumentation(options => // Ensure the correct using directive is added
            {
                options.SetDbStatementForText = true;  // Capture SQL queries
                options.SetDbStatementForStoredProcedure = true;
            })
            .AddSource("Ordering.API")
            .AddOtlpExporter(otlpOptions =>
            {
                otlpOptions.Endpoint = new Uri("http://localhost:4317");
                otlpOptions.Protocol = OpenTelemetry.Exporter.OtlpExportProtocol.Grpc;
            });
    })
    .WithMetrics(metrics =>
    {
        metrics
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddMeter("Ordering.API")
            .AddOtlpExporter(options =>
            {
                options.Endpoint = new Uri("http://localhost:4317");
            });
    });
```

Figure 2.2: *OpenTelemetry* configuration in *Ordering.API*

2.1 eShop.ServiceDefaults/Extensions.cs

To activate *OpenTelemetry* instrumentation, tracing, and *Prometheus* metric scraping, modification were made in this *file* (*Github File Link*):

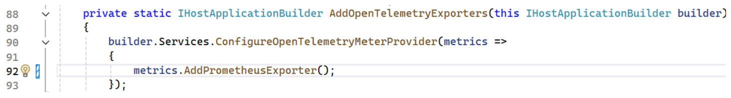
- Uncommented the *Prometheus* scraping endpoint:



```
public static WebApplication MapDefaultEndpoints(this WebApplication app)
{
    app.MapPrometheusScrapingEndpoint();
```

Figure 2.3: Enable *Prometheus* scraping endpoint

- Added *Prometheus* exporter configuration in *AddOpenTelemetryExporters* method:



```
private static IHostApplicationBuilder AddOpenTelemetryExporters(this IHostApplicationBuilder builder)
{
    builder.Services.ConfigureOpenTelemetryMeterProvider(metrics =>
    {
        metrics.AddPrometheusExporter();
    });
```

Figure 2.4: Enable *Prometheus* exporter

2.2 otel-collector-config.yml

This file configues *OpenTelemetry Collector* to receive telemetry data from microservices and export it to the appropriate backends (*Github File Link*).

2.3 prometheus.yml

Prometheus was configured to scrape the *OpenTelemetry Collector* for metrics (*Github File Link*).

2.4 docker-compose.yml

To make it easier to set up *OpenTelemetry Collector*, *Prometheus*, *Jaeger*, and *Grafana* a *docker-compose.yml* was created (*Github File Link*).

2.5 Observability Flow

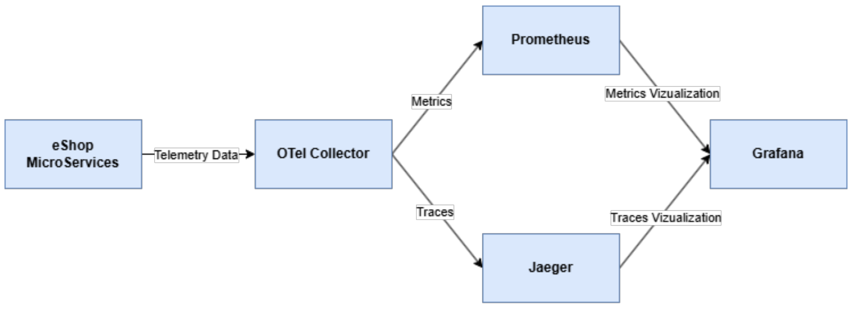


Figure 2.5: Observability Architecture Flow

3. Metrics and Traces Implemented

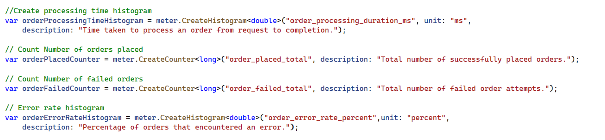
3.1 Metrics

For the *Place an Order* feature I decided to implement meaningful metrics in *Ordering.API* and in *Basket.API*.

3.1.1 *Ordering.API*

- **order_placed_total**: Counts the number of successfully placed orders.
- **order_failed_total**: Tracks the number of failed order attempts.
- **order_processing_duration_ms**: Measures the time taken to process an order from request to completion.
- **average_order_value**: Average monetary value of the placed orders.
- **monetary_value_orders**: Total monetary value of the placed orders.
- **order_error_rate**: Percentage of orders that encountered any errors.
- **payment_processing_time_ms**: Tracks the time taken to process payment for an order.

The first three metrics were implemented in *OrdersApi.cs* (*Github File Link*):



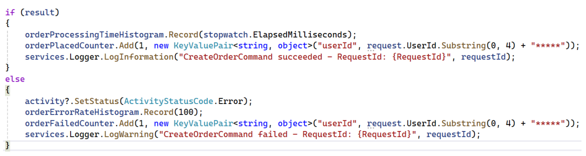
```
//Create processing time histogram
var orderProcessingTimeHistogram = meter.CreateHistogram<double>("order_processing_duration_ms", unit: "ms",
    description: "Time taken to process an order from request to completion.");

// Count Number of orders placed
var orderPlacedCounter = meter.CreateCounter<long>("order_placed_total", description: "Total number of successfully placed orders.");

// Count Number of failed orders
var orderFailedCounter = meter.CreateCounter<long>("order_failed_total", description: "Total number of failed order attempts.");

// Error rate histogram
var orderErrorRateHistogram = meter.CreateHistogram<double>("order_error_rate_percent",unit: "percent",
    description: "Percentage of orders that encountered an error.");
```

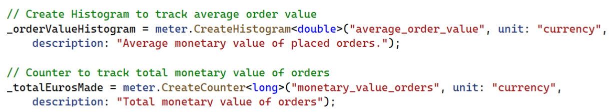
Figure 3.1: Metrics in *OrdersApi.cs*



```
if (result)
{
    orderProcessingTimeHistogram.Record(stopwatch.ElapsedMilliseconds);
    orderPlacedCounter.Add(1, new KeyValuePair<string, object>("userId", request.UserId.Substring(0, 4) + "*****"));
    services.Logger.LogInformation("CreateOrderCommand succeeded - RequestId: {RequestId}", requestId);
}
else
{
    activity?.SetStatus(ActivityStatusCode.Error);
    orderErrorRateHistogram.Record(100);
    orderFailedCounter.Add(1, new KeyValuePair<string, object>("userId", request.UserId.Substring(0, 4) + "*****"));
    services.Logger.LogWarning("CreateOrderCommand failed - RequestId: {RequestId}", requestId);
}
```

Figure 3.2: Metrics usage in *OrdersApi.cs*

The *average_order_value* and *monetary_value_orders* metrics were implemented in *Create-OrderCommandHandler.cs* (*Github File Link*):

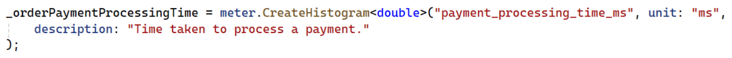


```
// Create Histogram to track average order value
_orderValueHistogram = meter.CreateHistogram<double>("average_order_value", unit: "currency",
    description: "Average monetary value of placed orders.");

// Counter to track total monetary value of orders
_totalEurosMade = meter.CreateCounter<long>("monetary_value_orders", unit: "currency",
    description: "Total monetary value of orders");
```

Figure 3.3: Metrics in *CreateOrderCommandHandler.cs*

The last metric, *payment_processing_time_ms* was made in *OrderStatusChangedToPaidDomainEventHandler.cs* (*Github File Link*):



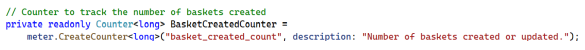
```
_orderPaymentProcessingTime = meter.CreateHistogram<double>("payment_processing_time_ms", unit: "ms",
    description: "Time taken to process a payment."
);
```

Figure 3.4: Metrics in *OrderStatusChangedToPaidDomainEventHandler.cs*

3.1.2 *Basket.API*

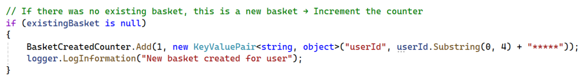
- **basket_created_count**: Count the number of baskets that were created.

This metric was implemented in *BasketService.cs* (*Github File Link*):



```
// Counter to track the number of baskets created
private readonly Counter<long> BasketCreatedCounter =
    meter.CreateCounter<long>("basket_created_count", description: "Number of baskets created or updated.");
```

Figure 3.5: Metrics in *BasketService.cs*



```
// If there was no existing basket, this is a new basket → Increment the counter
if (existingBasket is null)
{
    BasketCreatedCounter.Add(1, new KeyValuePair<string, object>("userId", userId.Substring(0, 4) + "*****"));
    logger.LogInformation("New basket created for user");
}
```

Figure 3.6: Metrics usage in *BasketService.cs*

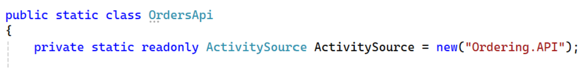
Each metric was then exported, allowing real-time monitoring through **Grafana Dashboard**.

3.2 Traces

To ensure full observability of the *Place an Order* feature, I used traces to track the entire flow across two microservices used in this feature (*Ordering.API*, *Basket.API*, and Databases Writes).

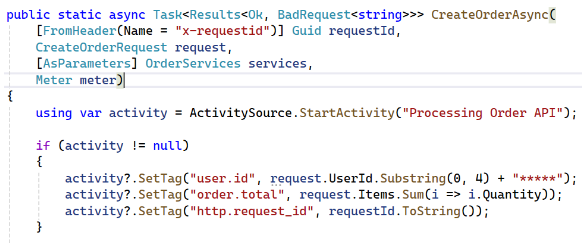
3.2.1 *Ordering.API*

When a order request reaches the *Ordering.API* microservice, a new activity is initialized in *OrdersApi.cs*(*Github File Link*) to keep track of that flow.



```
public static class OrdersApi
{
    private static readonly ActivitySource ActivitySource = new("Ordering.API");
```

Figure 3.7: Activity Initialization *OrdersApi.cs*



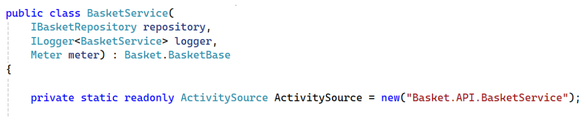
```
public static async Task<Results<Ok, BadRequest<string>>> CreateOrderAsync(
    [FromHeader(Name = "x-requestid")] Guid requestId,
    CreateOrderRequest request,
    [AsParameters] OrderServices services,
    Meter meter)
{
    using var activity = ActivitySource.StartActivity("Processing Order API");

    if (activity != null)
    {
        activity?.SetTag("user.id", request.UserId.Substring(0, 4) + "*****");
        activity?.SetTag("order.total", request.Items.Sum(i => i.Quantity));
        activity?.SetTag("http.request_id", requestId.ToString());
    }
```

Figure 3.8: Setting tags with the Activity

3.2.2 *Basket.API*

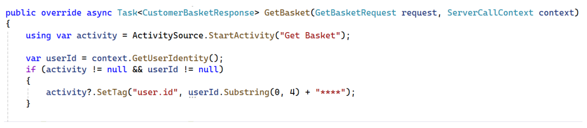
In *BasketService.cs* (*Github File Link*) tracing was added to keep track of Get the Basket, Update the Basket, and Delete the Basket actions.



```
public class BasketService(
    IBasketRepository repository,
    ILogger<BasketService> logger,
    Meter meter) : Basket.BasketBase
{

    private static readonly ActivitySource ActivitySource = new("Basket.API.BasketService");
```

Figure 3.9: Activity Initialization *BasketService.cs*



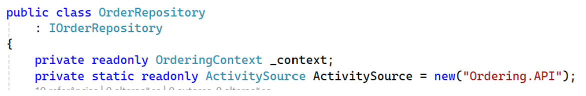
```
public override async Task<CustomerBasketResponse> GetBasket(GetBasketRequest request, ServerCallContext context)
{
    using var activity = ActivitySource.StartActivity("Get Basket");

    var userId = context.GetUserIdentity();
    if (activity != null && userId != null)
    {
        activity?.SetTag("user.id", userId.Substring(0, 4) + "****");
    }
```

Figure 3.10: Setting tags with the Activity

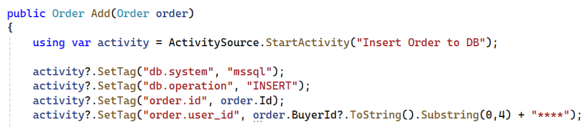
3.2.3 Databases Writes

I also added tracing to capture database interactions in *OrderRepository.cs* (*Github Link File*). Every time an order is created, retrieved, or updated, a new trace activity is created.



```
public class OrderRepository
    : IOrderRepository
{
    private readonly OrderingContext _context;
    private static readonly ActivitySource ActivitySource = new("Ordering.API");
```

Figure 3.11: Activity Initialization *OrderRepository.cs*



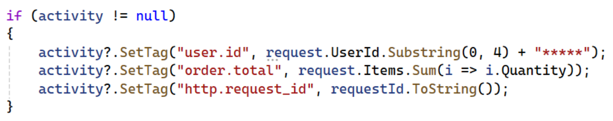
```
public Order Add(Order order)
{
    using var activity = ActivitySource.StartActivity("Insert Order to DB");

    activity?.SetTag("db.system", "mssql");
    activity?.SetTag("db.operation", "INSERT");
    activity?.SetTag("order.id", order.Id);
    activity?.SetTag("order.user_id", order.BuyerId?.ToString().Substring(0,4) + "****");
```

Figure 3.12: Setting tags with the Activity

3.2.4 Sensitive Data Masking in Traces and Logs

To fulfill one of the requirements of this assignment and to ensure security and compliance, all logs and activity traces containing sensitive data, such as user IDs or payment details were masked before being stored or transmitted.



```
if (activity != null)
{
    activity?.SetTag("user.id", request.UserId.Substring(0, 4) + "*****");
    activity?.SetTag("order.total", request.Items.Sum(i => i.Quantity));
    activity?.SetTag("http.request_id", requestId.ToString());
}
```

Figure 3.13: Example of masking sensitive data in traces

4. Grafana Dashboard

In this assignment *Grafana* was used to help the user visualize the metrics and traces implemented. *Grafana* will query *Prometheus* for having metrics and query *Jaeger* to have traces.

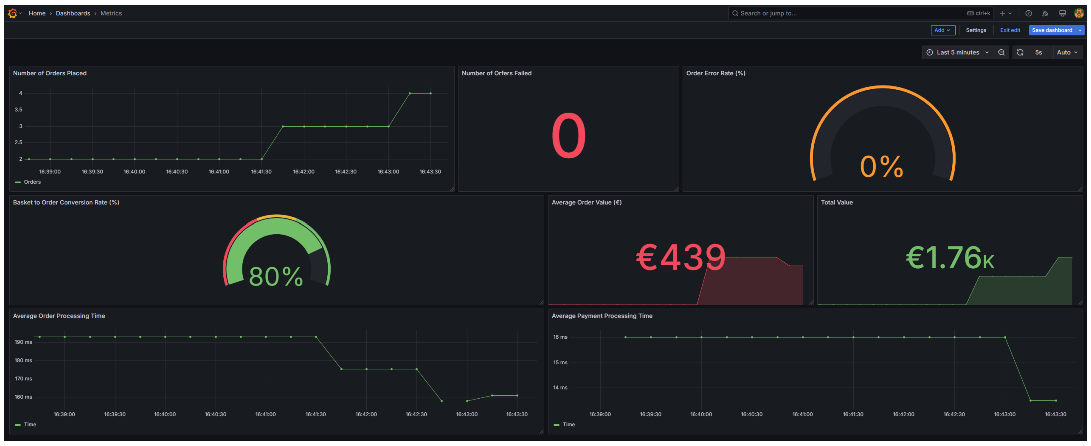

Home › Dashboards › Metrics
Number of Orders Placed
Number of Orfers Failed
0
Order Error Rate (%)
0%
Basket to Order Conversion Rate (%)
80%
Average Order Value (€)
€439
Total Value
€1.76K
Average Order Processing Time
Average Payment Processing Time

Figure 4.1: Dashboard to vizualize metrics

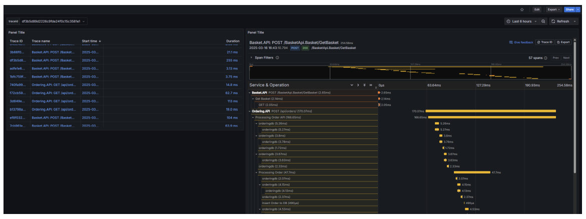
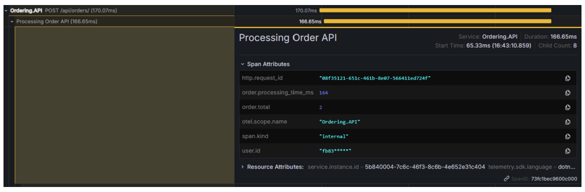


Figure 4.2: Dashboard to vizualize metrics



Ordering.API POST /api/orders/ (170.07ms) 170.07ms
Processing Order API (166.65ms) 166.65ms

Processing Order API

Service: Ordering.API | Duration: 166.65ms
Start Time: 65.33ms (16:43:10.859) | Child Count: 8

Span Attributes

http.request_id	"08f35121-651c-461b-8e07-566411ed724f"
order.processing_time_ms	164
order.total	2
otel.scope.name	"Ordering.API"
span.kind	"internal"
user.id	"fb83*****"

Resource Attributes: service.instance.id = 5b840004-7c6c-46f3-8c6b-4e652e31c404 telemetry.sdk.language = dotn...

SpanID: 73fc1bec9600c000

Figure 4.3: Traces Tag Masked when needed

5. Conclusion

OpenTelemetry integration into the eShop microservices system improved the already exiting observability present. Through Prometheus and Jaeger, we were able to monitor system performance, track order processing, and ensure efficient debugging of service interactions when an error or fail happens.

AI tools were used to help in the initial configuration of metrics and traces. This provided me some guidance on how to set up *OTel Collector* to export metrics to *Prometheus* and traces to *Jaeger*. Additionally, AI was also to improve the clarity and structure of this report.

Overall, this assignment gave me the opportunity to learn more about observability and how to implement it in practical. However, some challenges have appeared such as the initial configuration of *OpenTelemetry* exporters and ensuring the correct data flow between microservices, the collector, and the visualization tools. Despite that, the eShop system resulted with a functional observability system, improving monitoring, and performance analysis.