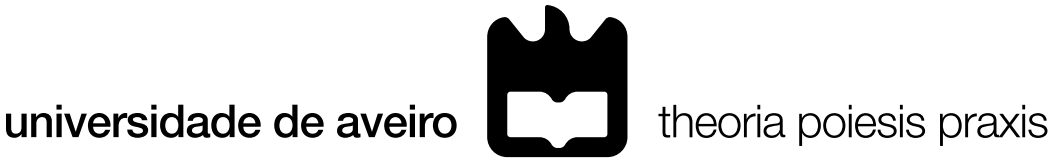

Integrating OpenTelemetry & Security in eShop

1st Assignment Report
Software Architectures Course
2024/2025

Sara Almeida - 108796
Master's in Software Engineering
March 16th, 2025

Contents

1. Introduction

This project aimed to integrate **Open Telemetry** tracing and security measures in an already developed solution, in this case, in the *eShop* microservices system. This report details the implementation for enabling end-to-end observability for a selected feature. In the end, it is possible to visualize traces and metrics in a **Grafana** dashboard for the features of *Add to Cart* and *Place an Order* and masking of sensitive data was also accomplished. The mentioned features were selected as they show interesting and critical interactions with the system and all calls can be traced end-to-end.

1.1 Useful Links

- Github

2. OpenTelemetry Setup

To setup OpenTelemetry, I tried to follow the previous class's assignment on deploying observability. I knew I had to setup Docker files and the correct exporters for traces and metrics. This way, after a detailed analysis of the code provided for this project and some *gen AI*, these were the first changes I made:

- Uncomment this line in *Extensions.cs* in the **eShop.ServiceDefaults**:

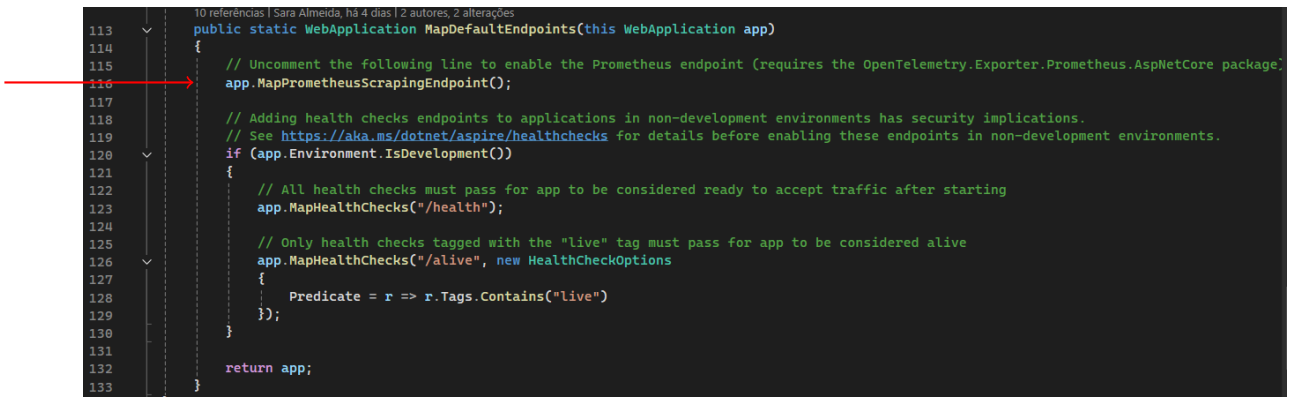

Figure 2.1: Uncommenting line to enable the Prometheus endpoint

- Install *OpenTelemetry.Exporter.Prometheus.AspNetCore* with the command:

```
dotnet add package OpenTelemetry.Exporter.Prometheus.AspNetCore --prerelease
```

- Add Prometheus Exporter in *Extensions.cs* in the **eShop.ServiceDefaults**:

```
1 referência | Sara Almeida, há 4 dias | 2 autores, 2 alterações
private static IHostApplicationBuilder AddOpenTelemetryExporters(this IHostApplicationBuilder builder)
{
    builder.Services.ConfigureOpenTelemetryMeterProvider(metrics =>
    {
        metrics.AddPrometheusExporter();
    });

    var useOtlpExporter = !string.IsNullOrWhiteSpace(builder.Configuration["OTEL_EXPORTER_OTLP_ENDPOINT"]);

    if (useOtlpExporter)
    {
        builder.Services.Configure<OpenTelemetryLoggerOptions>(logging => logging.AddOtlpExporter());
        builder.Services.ConfigureOpenTelemetryMeterProvider(metrics => metrics.AddOtlpExporter());
        builder.Services.ConfigureOpenTelemetryTracerProvider(tracing => tracing.AddOtlpExporter());
    }

    return builder;
}
```

Figure 2.2: Adding Prometheus Exporter

2.1 Docker files

In resemblance with the "Deploying Observability" assignment's code, I created Docker files to successfully export metrics to **Prometheus** and traces to **Jaeger** and define the **OpenTelemetry Collector** configuration:

- **otel-collector-config.yaml**: Configures how telemetry data (traces and metrics) is collected and exported:
  - Receives telemetry data from services via OTLP (4317 gRPC, 4318 HTTP);
  - Exports traces to Jaeger (14268);
  - Exports metrics to Prometheus (9464).
- **prometheus.yml**: Defines how Prometheus collects and stores metrics:
  - Scrapes metrics from otel-collector:9464 every 5 seconds;
  - Stores metrics for visualization in Grafana.
- **docker-compose.yml**: Defines and starts all monitoring services together:
  - otel-collector → Receives and exports telemetry data;
  - prometheus → Collects and stores metrics (9090);
  - grafana → Visualizes metrics (3000);
  - jaeger → Displays traces (16686).

3. OpenTelemetry Integration

After the initial setup, I had to make changes in both the **Basket.API** and the **Ordering.API** given the features I chose.

For the OpenTelemetry setup in each of the APIs modified, some additions were needed in the *Program.cs* files of both. This changes are **identical** for both Basket.API and Ordering.API, the only difference being when instantiating something, like in the meter instance e.g., where `Basket.API` is replaced by `Ordering.API`. The new code is as follow:

- Adding necessary imports:

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

Figure 3.1: Needed imports

- Creating Meter instance to enable collecting metrics:

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

Figure 3.2: Meter Instance

- Configuration of OpenTelemetry to enable exporting both metrics and traces through the `otel collector`:

```
builder.Services.AddOpenTelemetry()
    .WithTracing(tracerProviderBuilder =>
    {
        tracerProviderBuilder
            .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("Basket.API"))
            .AddAspNetCoreInstrumentation()
            .AddGrpcClientInstrumentation()
            .AddHttpClientInstrumentation()
            .AddSource("Basket.API")
            .AddOtlpExporter(options =>
            {
                options.Endpoint = new Uri("http://localhost:4317");
                options.Protocol = OpenTelemetry.Exporter.OtlpExportProtocol.Grpc;
            });
    })
    .WithMetrics(metrics =>
    {
        metrics
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddMeter("Basket.API")
            .AddOtlpExporter(options =>
            {
                options.Endpoint = new Uri("http://localhost:4317");
            });
    });
```

Figure 3.3: OpenTelemetry Configuration in the API

- Enabling Prometheus Scraping Endpoint:

```
// Export Metric
app.UseOpenTelemetryPrometheusScrapingEndpoint();
```

Figure 3.4: Enable Prometheus Scraping Endpoint

3.1 Basket.API

Now regarding the implementation itself of the metrics and traces particular for the `Basket.API`, this was done in the **/Grpc/BasketService.cs** starting with:

- Injecting Meter instance;
- Defining an Activity Source for Tracing;
- Creating OpenTelemetry Counters for Metrics;

```
8 referências | Sara Almeida, há 4 dias | 1 autor, 2 alterações
13  public class BasketService(
14      IBasketRepository repository,
15      ILogger<BasketService> logger,
16      Meter _meter) : Basket.BasketBase
17
18
19  {
20      private static readonly ActivitySource ActivitySource = new("Basket.API");
21      private readonly Counter<int> addToCartCounter = _meter.CreateCounter<int>("basket_add_to_cart_total", "items", "Total number of items added to the cart");
22      private readonly Counter<int> activeBasketsCounter = _meter.CreateCounter<int>("baskets_created_total","baskets","Total number of active baskets");
```

Figure 3.5: Registering OpenTelemetry Metrics and Traces in Basket.API

For the following, all the code necessary was added in the *UpdateBasket* method in the same file referenced above:

3.1.1 Basket.API Metrics

- **basket_add_to_cart_total**: Counts the total number of items added to cart;

```
74          int new_total = request.Items.Sum(item => item.Quantity);
75          int difference = new_total - previous;
76
77          if (difference > 0)
78          {
79              addToCartCounter.Add(difference);
80              logger.LogDebug($"Added {difference} items to cart (previous: {previous}, new: {new_total})");
81          }
```

Figure 3.6: Example basket_add_to_cart_total metric

- **baskets_created_total**: Counts the total number of different baskets opened.

3.1.2 Basket.API Traces

- **"AddToCart"**: Traces the UpdateBasket operation:
  - Includes masked *user.id*;
  - Tracks the items count in the basket;
  - Sets trace status (OK/Error) based on request success.

```
50          using var activity = ActivitySource.StartActivity("AddToCart", ActivityKind.Server);
51          var userId = context.GetUserIdentity();
52          if (string.IsNullOrEmpty(userId))...
56
57          if (logger.IsEnabled(LogLevel.Debug))...
61
62          string maskedUserId = MaskUserId(userId);
63          activity?.SetTag("user.id", maskedUserId);
64          activity?.SetTag("basket.item_count", request.Items.Count);
```

Figure 3.7: "AddToCart" Tracing

```
89          if (response is null)
90          {
91              activity?.SetStatus(ActivityStatusCode.Error);
92              ThrowBasketDoesNotExist(userId);
93          }
94          else
95          {
96              activity?.SetStatus(ActivityStatusCode.Ok);
97          }
```

Figure 3.8: "AddToCart" Tracing

3.1.3 Masking Sensitive Data

As it can be seen in Figure 3.7, for masking sensitive data in this API's trace, the method *MaskUserId* is called and it uses the `System.Security.Cryptography` import to hash the user.id:

```
155     private static string MaskUserId(string userId)
156     {
157         using var sha256 = SHA256.Create();
158         byte[] hashedBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(userId));
159         return Convert.ToBase64String(hashedBytes).Substring(0, 10); // Shorten for readability
160     }
161 }
```

Figure 3.9: Meter Instance

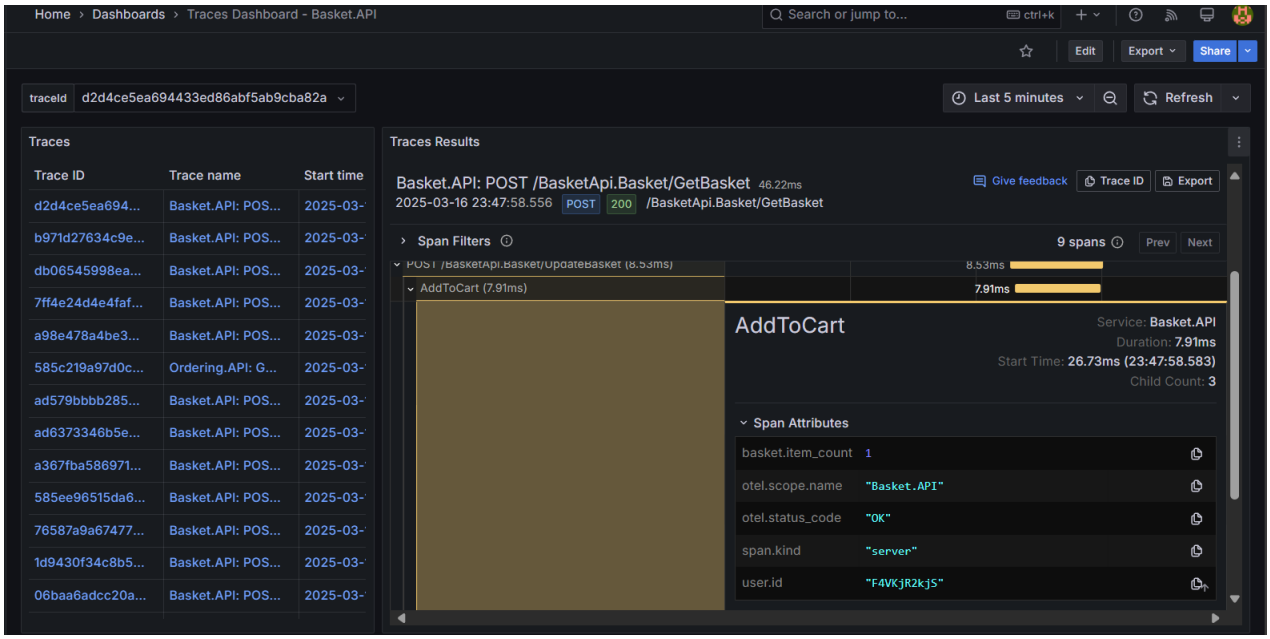

Figure 3.10: user.id masked in Grafana

3.2 Ordering.API

Now for the `Ordering.API`'s metrics and traces, this was done in the **/Application/Commands/CreateOrderCommandHandler.cs** and, in similarity with the before presented, starts with:

- Injecting Meter instance;
- Defining an Activity Source for Tracing;
- Creating OpenTelemetry Counters and Histogram for Metrics;

```
19
20      private static readonly ActivitySource ActivitySource = new("Ordering.API");
21      private readonly Counter<int> total_items_purchased;
22      private readonly Counter<int> total_orders;
23      private readonly Counter<double> total_value;
24      private readonly Histogram<double> order_processing_time;
25
26      // Using DI to inject infrastructure persistence Repositories
        1 referência | Sara Almeida, há 4 dias | 1 autor, 1 alteração
27      public CreateOrderCommandHandler(IMediator mediator,
28          IOrderingIntegrationEventService orderingIntegrationEventService,
29          IOrderRepository orderRepository,
30          IIdentityService identityService,
31          ILogger<CreateOrderCommandHandler> logger,
32          Meter meter)
33      {
34          _orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
35          _identityService = identityService ?? throw new ArgumentNullException(nameof(identityService));
36          _mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));
37          _orderingIntegrationEventService = orderingIntegrationEventService ?? throw new ArgumentNullException(nameof(orderingIntegrationEventService));
38          _logger = logger ?? throw new ArgumentNullException(nameof(logger));
39
40          total_items_purchased = meter.CreateCounter<int>("total_items_purchased", "Total items purchased");
41          total_orders = meter.CreateCounter<int>("total_orders", "Total orders number");
42          total_value = meter.CreateCounter<double>("total_value", "Total money made");
43          order_processing_time = meter.CreateHistogram<double>("order_processing_time", "seconds", "Time taken to process an order");
```

Figure 3.11: Registering OpenTelemetry Metrics and Traces in Ordering.API

For the following, all the code necessary was added in the *Handle* method in the same file referenced above:

3.2.1 Ordering.API Metrics

- **total_items_purchased**: Counts the total number of items actually purchased;
- **total_orders**: Counts the total number of orders finished;
- **total_value**: Counts the total value in money made with all the orders;
- **order_processing_time**: Total time taken to process an order, used to analyze performance.

```
68          int total_items_topurchase = 0; // Contador de itens
69          double total_money = 0;
70
71
72          foreach (var item in message.OrderItems)
73          {
74              order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice, item.Discount, item.PictureUrl, item.Units);
75
76              total_items_topurchase += item.Units;
77              int units = item.Units;
78              _logger.LogInformation("units:" + units);
79              _logger.LogInformation("total items" + total_items_topurchase);
80
81              total_money += (double)(item.Units * item.UnitPrice);
82              _logger.LogInformation("money for " + item + ":" + total_money);
83
84          }
85
86          _logger.LogInformation("Creating Order - Order: {@Order}", order);
87
88          total_value.Add(total_money);
89          total_items_purchased.Add(total_items_topurchase);
90
91          _orderRepository.Add(order);
92          total_orders.Add(1);
93          _logger.LogInformation($"Items Purchased: {total_items_topurchase}");
```

Figure 3.12: Example Ordering.API metrics

3.2.2 Ordering.API Traces

- **"PlaceOrder"**: Traces order processing time and masked user data for each order:
  - Includes masked *user.id* and *cart.security_number* (I first tried masking cart.number but soon realized this already came masked);
  - Tracks order processing time;

```
1 referência | Sara Almeida, há 4 dias | 1 autor, 1 alteração
46          public async Task<bool> Handle(CreateOrderCommand message, CancellationToken cancellationToken)
47          {
48              using var activity = ActivitySource.StartActivity("PlaceOrder", ActivityKind.Server);
49              activity?.SetTag("user.id", MaskUserId(message.UserId)); // ◇ Masked User ID
50              //activity?.SetTag("card.number", MaskCardNumber(message.CardNumber)); // ◇ Masked Card Number
51              activity?.SetTag("card.number", message.CardNumber); // Card Number
52              activity?.SetTag("card.security_number", MaskCardSecurityNumber(message.CardSecurityNumber));
53
54
55              var stopwatch = Stopwatch.StartNew(); // Start measuring time
56
```

Figure 3.13: "PlaceOrder" Tracing

```
95              stopwatch.Stop();
96              double elapsedSeconds = stopwatch.Elapsed.TotalSeconds;
97
98              order_processing_time.Record(elapsedSeconds);
99              activity?.SetTag("order.processing_time", elapsedSeconds);
100             _logger.LogInformation($"Order processing time: {elapsedSeconds} seconds");
```

Figure 3.14: "PlaceOrder" Tracing

3.2.3 Masking Sensitive Data

As it can be seen in Figure 3.13, for masking the card number in this API's trace, the method *MaskCardSecurityNumber* which is the same as the one presented in the section before for the same mater.

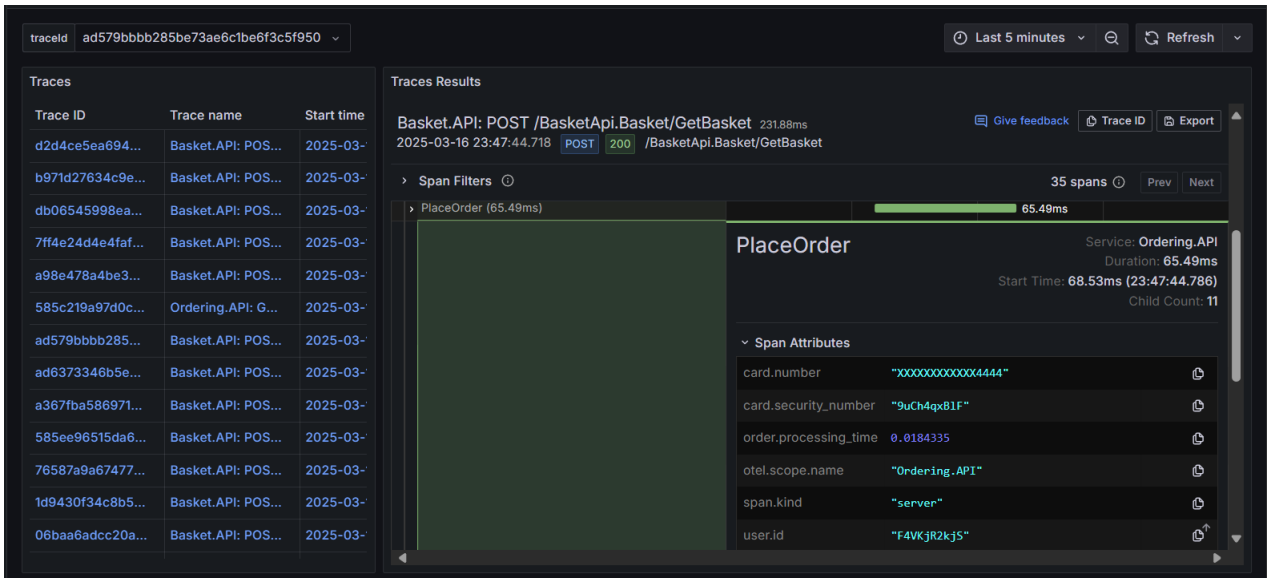

Figure 3.15: user.id and card.security_number masked in Grafana

4. Grafana Dashboard

The **Grafana** Dashboard allowed for visualization of the metrics and traces implemented throughout the time, as it is exemplified in the images bellow:

Figure 4.1: Grafana Metrics Dashboard

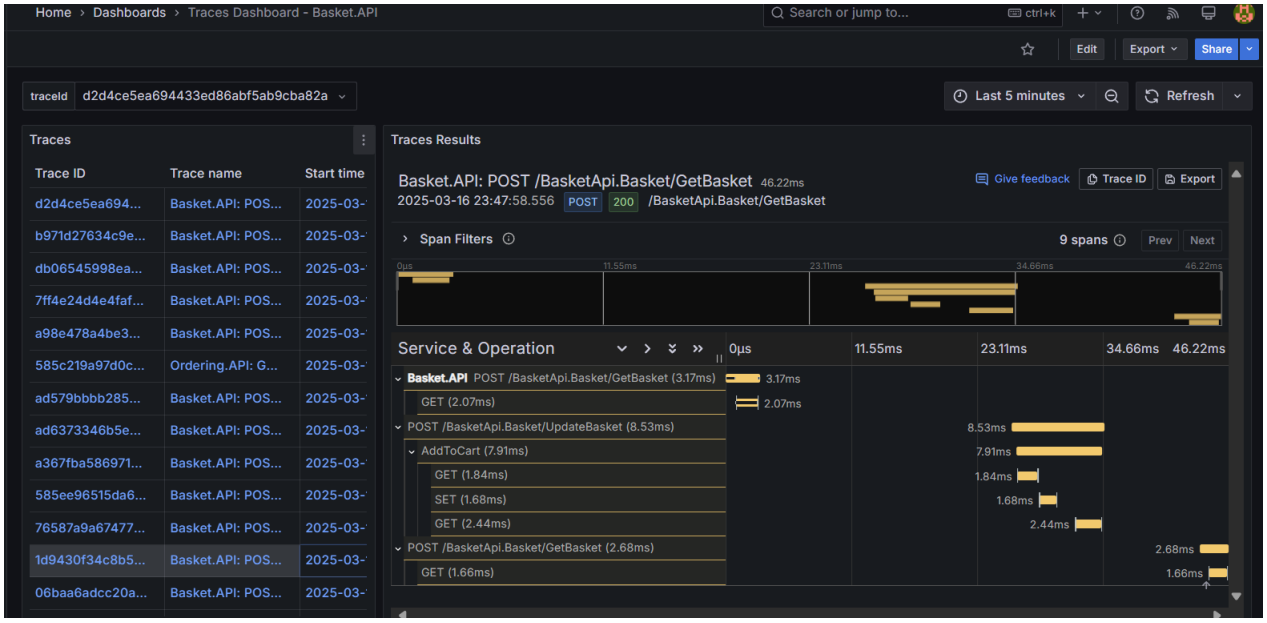

Figure 4.2: Grafana Traces Dashboard

5. Architecture Diagram

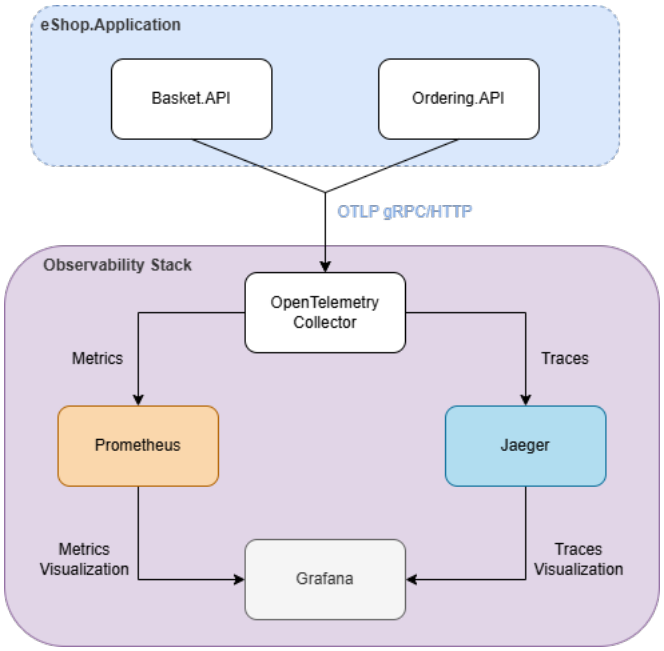

Figure 5.1: Observability Architecture Diagram

6. Conclusion

With this project I was able to successfully integrate OpenTelemetry tracing and security measures into the eShop microservices system, enhancing observability and security.

Through instrumentation of Basket.API and Ordering.API, I was able to:

- Implement metrics and traces for the "Add to Cart" and "Place an Order" features.
- Ensure traces and metrics were collected in OpenTelemetry and can be visualized not only in Prometheus and Jaeger, but also in a Grafana dashboard.
- Mask sensitive user data (user.id, card.security_number) to comply with security best practices.

For this, I had some help of *gen AI* tools, namelly ChatGPT, especially for some docker configurations and first insights on OpenTelemetry configurations and setup.

In conclusion, this project demonstrated how tracing, metrics, and logging can be combined to create a reliable and secure monitoring solution, helping developers debug, optimize, and enhance system performance.