Integrating OpenTelemetry & Security in eShop Report

José Miguel Costa Gameiro, 108840

Professor Cláudio Teixeira

Software Architectures
DETI
Universidade de Aveiro
March, 2025

Table of Contents

1 Introduction

The objective of this first assignment is to:

1. **Implement Open Telemetry tracing** on a single feature or use-case (end-to-end)
2. **Mask or exclude sensitive data** (e.g., email, payment details) from telemetry and logs
3. **Set up a basic Grafana dashboard** to visualize the traces and metrics
4. **Explore data encryption and compliance** in the database layer and introduce **column masking** for sensitive data

All of this points would be integrated in this full stack application available through this link https://github.com/dotnet/eShop/tree/main.

All the code developed and some instructions on how to execute the solution is available through this link.

This report presents a detailed explanation of the implemented solution.

2 Implementation

For this assignment I decided to integrate tracing and metrics in the add an item to the basket use-case. To achieve this I used ***Jaeger*** to collect **Traces** from the application, an ***Otel-Collector*** and ***Prometheus*** to collect metrics, the ***Open Telemetry Protocol*** was used to communicate with the ***Otel-Collector***.

The first step was to discover which services were included when an item was added to the basket, which are the ***Web App***, the ***Catalog API***, the ***Basket API*** and the ***RedisBasketRepository***.

2.1 Jaeger and Traces

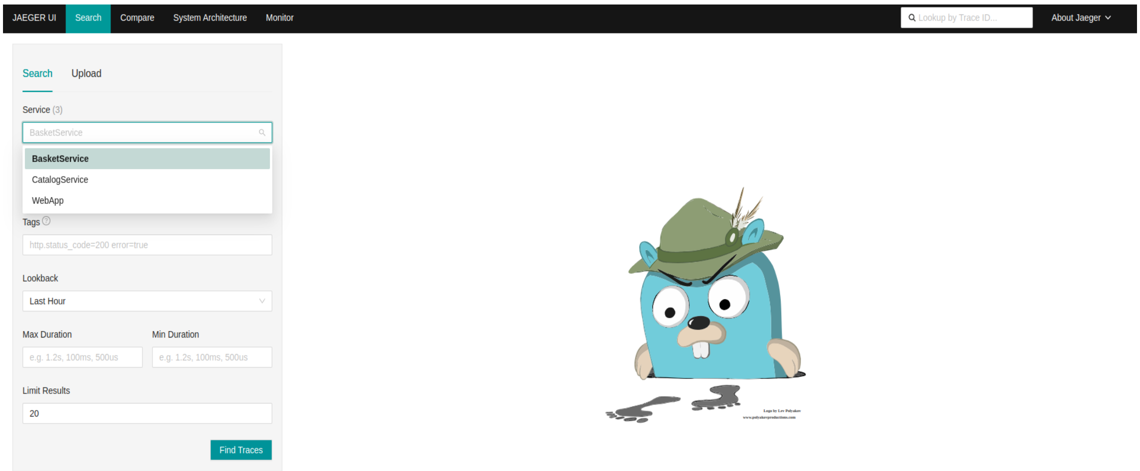
2.1.1 Configuring Jaeger

To configure Jaeger with the purpose of receiving and viewing traces from the application, I created a simple ***docker-compose.yml*** file with the following code:

```
services:
    jaeger:
        image: jaegertracing/
                all-in-one:latest
        container_name: jaeger
        ports:
            - 16686:16686 # Jaeger UI
            - 4317:4317 # OTLP gRPC
            - 4318:4318 # OTLP HTTP
        networks:
            - monitoring
```

Where the **image** used includes all Jaeger components in a single container, I named him *jaeger* for easy reference. I exposed the ports 16686 for the Jaeger UI (to view the traces), 4317 that enables OTLP communication over gRPC and 4318 that allows OTLP communication over HTTP. Finally I created a custom bridge network called *monitoring*, to allow all the containers to communication with each other while being separated from the host network, working like an isolated network.

After executing this docker with success, it is possible to access the ***Jaeger UI*** through the link http://localhost:16686:



2.1.2 Configuring Traces

With ***Jaeger*** running and working, the next step was to configure the services to send traces to Jaeger. And for that I decide to first test the integration of Open Telemetry on the Basket.API service. To achieve this I included the following code in the ***Program.cs*** file inside the ***src/Basket.API/*** directory.

```
var serviceName = "BasketService";
builder.Services.AddOpenTelemetry()
    .ConfigureResource(resource => resource
        .AddService(serviceName))
    .WithTracing(tracerProviderBuilder => tracerProviderBuilder
        .AddAspNetCoreInstrumentation()
        .AddProcessor(new MaskingActivityProcessor())
        .AddGrpcClientInstrumentation()
        .AddHttpClientInstrumentation()
        .AddSqlClientInstrumentation()
        .AddSource(serviceName)
        .AddOtlpExporter(opt =>
        {
            opt.Endpoint = new Uri("http://localhost:4317");
            opt.Protocol = OtlpExportProtocol.Grpc;
        }));
```

This code configures OpenTelemetry in the Basket.API service to send tracing data to Jaeger.

- It defines "BasketService" as the service name for tracing.
- OpenTelemetry is set up with instrumentation for ASP.NET Core, gRPC, HTTP clients, and SQL clients to capture trace data automatically.
- A custom processor (MaskingActivityProcessor) is included, likely for modifying or filtering traces, which will be explained later.
- The OTLP (OpenTelemetry Protocol) exporter is configured to send traces to Jaeger via gRPC at http://localhost:4317.

This ensures the Basket.API service collects and forwards trace data to Jaeger for monitoring and analysis.

Some custom traces where added to the Basket.API, to see if the configuration of the Open Telemetry worked. This traces where added to the functions ***GetBasket***, ***UpdateBasket***, ***MapToCustomerBasketResponse*** and ***MapToCustomerBasket***. The first step to have this traces is configure an activity source with the following line:

```
private static readonly ActivitySource activitySource
= new("BasketService");
```

Where it creates a new activitySource for the *BasketService*. Then in the functions ***GetBasket*** and ***UpdateBasket*** an activity was created from the activity source with this code:

```
using var activity =
activitySource.StartActivity("GetBasket", ActivityKind.Server);
```

Where the name of the activity that its tracking is *GetBasket* and the *ActivityKind.Server* specifies that the activity is happening on the server side.

With all of this conclude, all thats left is add some custom traces, like the following:

```
var userId = context.GetUserIdentity();
if (string.IsNullOrEmpty(userId))
{
    activity?.SetStatus(
        ActivityStatusCode.Error,
        "User not authenticated"
    );
    activity?.AddEvent(new ActivityEvent(
    "ERROR: User not authenticated"
    ));
    return new();
}
activity.SetTag("user.id", userId);
activity.SetTag("basket.access_time", DateTime.UtcNow);
```

```
var data = await repository.GetBasketAsync(userId);

if (data is not null)
{
    activity?.SetStatus(
        ActivityStatusCode.Ok,
        "Basket found with success"
    );
    activity?.AddEvent(new ActivityEvent(
        "SUCCESS: Basket found with success"
    ));
    return MapToCustomerBasketResponse(data, activity);
}

activity?.SetStatus(ActivityStatusCode.Ok, "Basket Empty");
```

Where if an error occurred the status of the activity is set to *ActivityStatusCode.Error* and a detailed message is provided. Sometimes an activity events are also used so that this messages/logs can be recorded in the monitoring tool used (Jaeger). Custom tags are also defined in the activity like the ***user ID*** and the ***access time***.

In the functions ***MapToCustomerBasketResponse*** and ***MapToCustomerBasket*** the following tags and events where added:

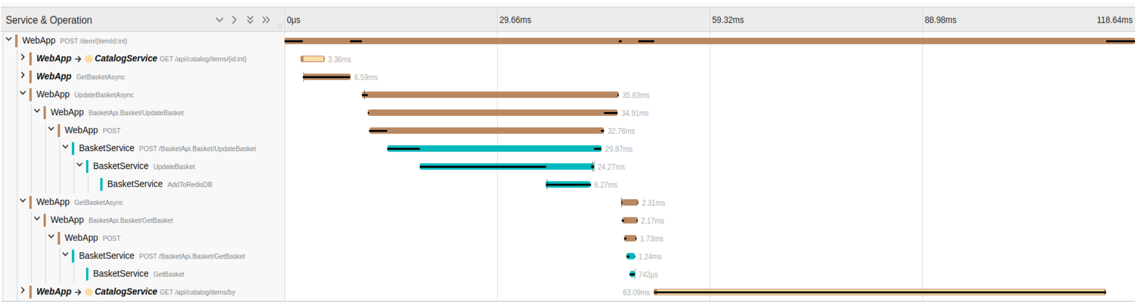
```
// This one in the MapToCustomerBasketResponse
foreach (var item in customerBasket.Items)
{
    // ... the rest of the implementation
    string eve = $"Updated item {item.ProductId} to {item.Quantity}";
    activity.AddEvent(new ActivityEvent(eve));
}

// This one in the MapToCustomerBasket
foreach (var item in customerBasketRequest.Items)
{
    // ... the rest of the implementation
    addToBasketItemsAddedCounter.Add(item.Quantity);
    string tag = $"basket.item.{item.ProductId}";
    activity.SetTag(tag, item.Quantity);
}
```

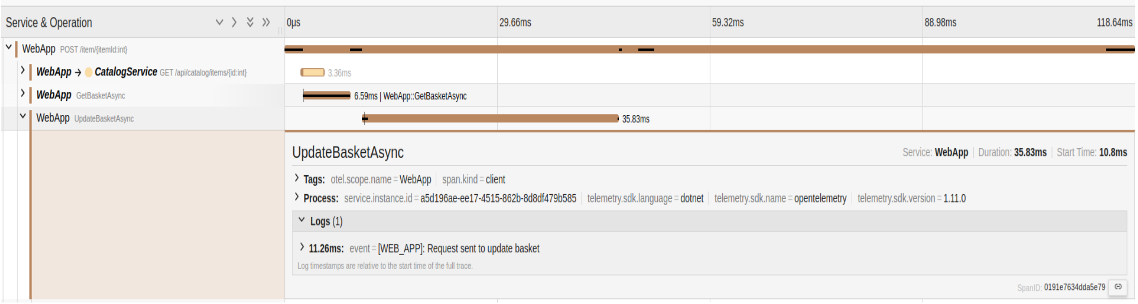
With this the products that were added to the basket were also tracked in the tags and logs of the activity. Some custom ***ActivityEvents*** were also added to the **WebApp** service and **RedisBasketRepository** to have a full trace from the web application to the storage process.

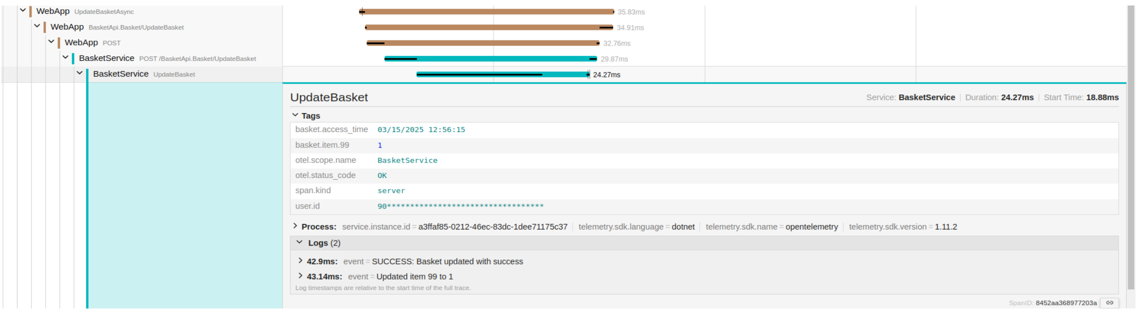
```
// This two in the Basket Service of the WebApp Component
activity?.AddEvent(new ActivityEvent(
```

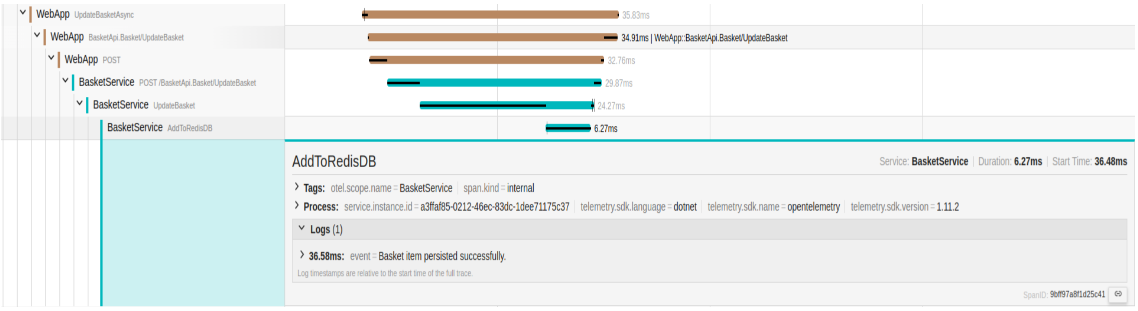
```
    "[WEB_APP]: Request sent to GetBasket in Basket.API"
));
activity?.AddEvent(new ActivityEvent(
    "[WEB_APP]: Request sent to update basket"
));

// This one in the RedisBasketRepository
activity.AddEvent(new ActivityEvent(
    "Basket item persisted successfully."
));
```

The following images show the traces capture by Jaeger for the functionality add an item to the basket:









2.2 Prometheus and Metrics

2.2.1 Configuring Prometheus

To allow the observability of metrics of the system, **Prometheus** was configured to collect metrics from an **Otel-Collector** and store them. For this 2 configuration files were created, called ***prometheus.yml*** and ***otel-collector.yml*** and 2 services were added to the ***docker-compose.yml*** file that was mentioned before.

An **Otel-Collector** was used to help gather metrics from different sources and send them to Prometheus. If the **Otel-Collector** was not used, it would have to scrape metrics directly from each service, which can be complex. So this makes **Otel-Collector** as a middleman, making it easier to collect, process, and send metrics in a structured way.

```
# prometheus.yml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'otel-collector'
    metrics_path: '/metrics'
    static_configs:
      - targets: ['otel-collector:8889']
```

This configuration file tells **Prometheus** how often to collect data and where to get it from, where it should collect metrics and evaluate alerting rules every **15 seconds**. The name of the data source where Prometheus should retrieve metrics is defined in the ***job-name*** and the endpoint to collect data is defined in the ***metrics-path***.

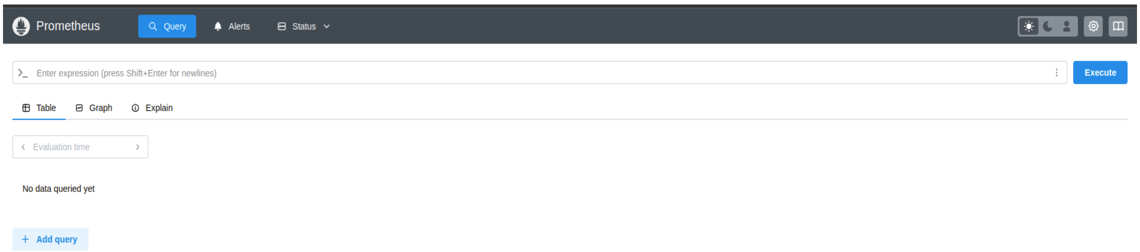
```
# otel-collector.yml
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: "0.0.0.0:4316"
processors:
  batch:
exporters:
  debug:
    verbosity: detailed
  prometheus:
    endpoint: "0.0.0.0:8889"
service:
  pipelines:
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [debug, prometheus]
```

This ***Otel-Collector* configuration** sets up how metrics are receive, processed, and exported. It collects metrics using **gRPC on port 4316**, batches them for efficiency, and then exports them to **Prometheus on port 8889** for monitoring while also logging detailed debug info.

```
# docker-compose.yml
services:
    # ... jaeger service
    prometheus:
        image: prom/prometheus:latest
        container_name: prometheus
        ports:
          - 9090:9090
        volumes:
          - ./prometheus.yml:/etc/prometheus/prometheus.yml
        networks:
          - monitoring
    otel-collector:
        image: otel/opentelemetry-collector:latest
        container_name: otel-collector
        command: ["--config=/etc/otel-collector-config.yml"]
        volumes:
          - ./otel-collector-config.yml:/etc/otel-collector-config.yml
        ports:
          - 4316:4316
          - 8889:8889
        networks:
          - monitoring
```

As stated before two new services were added to the docker-compose file, to set up ***Prometheus*** and ***Otel-Collector*** as services in a shared monitoring network. ***Prometheus*** runs on port 9090, using the configuration file explain before, and stores collected metrics. ***Otel-Collector*** runs on ports 4316 (for receiving metrics via gRPC) and 8889 (for exposing metrics to Prometheus), using the configuration file also explained above. Bot services run in separate containers but communicate within the same network.

With everything configured, when executing the new updated docker-compose file, the ***Prometheus UI*** should be accessible through the link http:/localhost:9090/ and it should look something like this:



2.2.2 Configuring Custom Metrics

To configure the application to send metrics to ***Otel-Collector*** with was required to add some configurations to the ***Program.cs*** file inside the **Basket.API** folder.

```
builder.Services.AddOpenTelemetry()
.ConfigureResource(resource => resource
    .AddService(serviceName))
// configuration of traces
.WithMetrics(metricsProviderBuilder =>
{
    metricsProviderBuilder
        .AddAspNetCoreInstrumentation()
        .AddMeter(serviceName)
        .AddOtlpExporter(opt =>
        {
            opt.Endpoint = new Uri("http://localhost:4316");
            opt.Protocol = OtlpExportProtocol.Grpc;
        });
});
```

This configures metrics collections:

- ***AddAspNetCoreInstrumentation()*:** Captures metrics from ***ASP.NET* Core requests** automatically
- ***AddMeter()*:** Register custom application metrics
- ***AddOtlpExporter(opt = {...})*:** Sends metrics to ***Otel-Collector*** at **http://localhost:4316/** using **gRPC**

Custom metrics were only added to the **Basket.API**, in the **BasketService.cs**, where three counters and an histogram where created:

```
private static readonly Meter meter = new("BasketService");

private static readonly Counter<long>
    addToBasketCounter =
        meter.CreateCounter<long>(
            "basket.add_to_basket.count"
        );

private static readonly Counter<long>
    addToBasketItemsAddedCounter =
        meter.CreateCounter<long>(
            "basket.add_to_basket.items_added"
        );

private static readonly Counter<long>
    addToBasketErrorsCounter =
```

```
        meter.CreateCounter<long>(
            "basket.add_to_basket.errors.count"
        );

private static readonly Histogram<double>
    addToBasketDurationHistogram =
        meter.CreateHistogram<double>(
            "basket.add_to_basket.duration", "milliseconds"
        );
```

- ***Meter meter = new("BasketService")*: Creates a meter to group related metrics for **BasketService**;
- **Counters:** Track how often something happens:
  - **addToBasketCounter:** Counts how many times items are added to the basket;
  - **addToBasketItemsAddedCounter:** Counts the total number of items add;
  - **addToBasketErrorsCounter:** Counts errors when adding items.

The process of incrementing the different counters depends on the purpose of each one, for example the ***addToBasketErrorsCounter*** only gets incremented when an error occurs, so it only appears inside this two *if statements*:

```
if (string.IsNullOrEmpty(userId))
{
    // .... rest of the implementation
    addToBasketErrorsCounter.Add(1);
    // .... rest of the implementation
}

if (response is null)
{
    // .... rest of the implementation
    addToBasketErrorsCounter.Add(1);
    // .... rest of the implementation
}
```

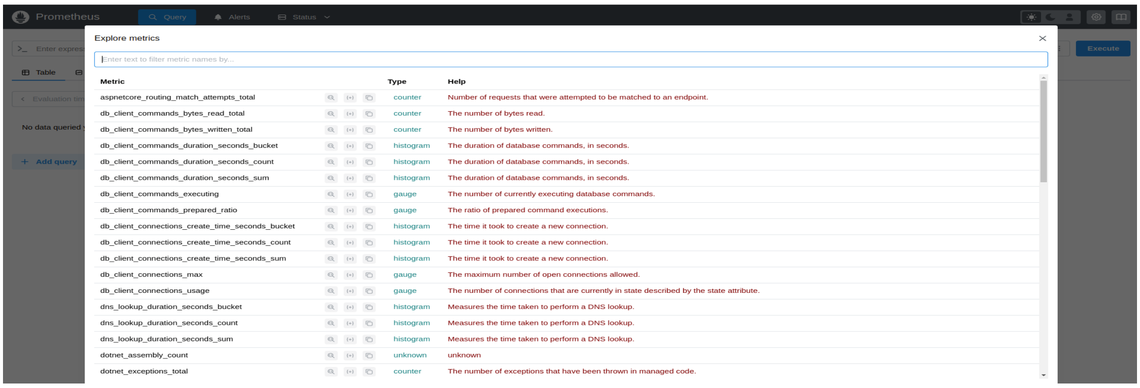
Since the ***addToBasketCounter*** counts how many times items were added to the basket then this counter is only incremented one time inside the ***UpdateBasket*** function.

```
// Function UpdateBasket
addToBasketCounter.Add(1);
```

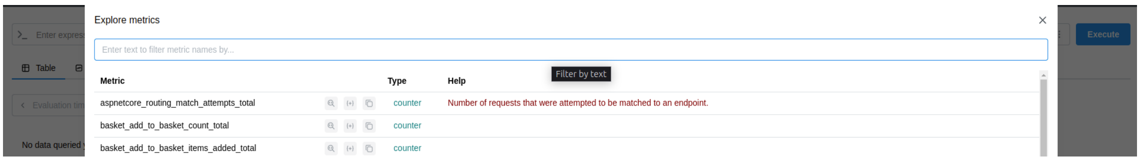
For the ***addToBasketItemsAddedCounter***, the process of increment this counter is achieved in the function ***MapToCustomerBasket***, and instead of incrementing one unit, it adds the total quantity of each item:

```
// function MapToCustomerBasket
foreach (var item in customerBasketRequest.Items)
{
    // ... rest of the implementation
    addToBasketItemsAddedCounter.Add(item.Quantity);
}
```

Now with all of this configured, when navigating to the ***Prometheus UI*** and explore the available metrics, some default of the **ASP.NET** application will appear:



And the custom ones will only appear once an action that creates this metrics occurs like adding an item to the basket:



2.3 Grafana Dashboard

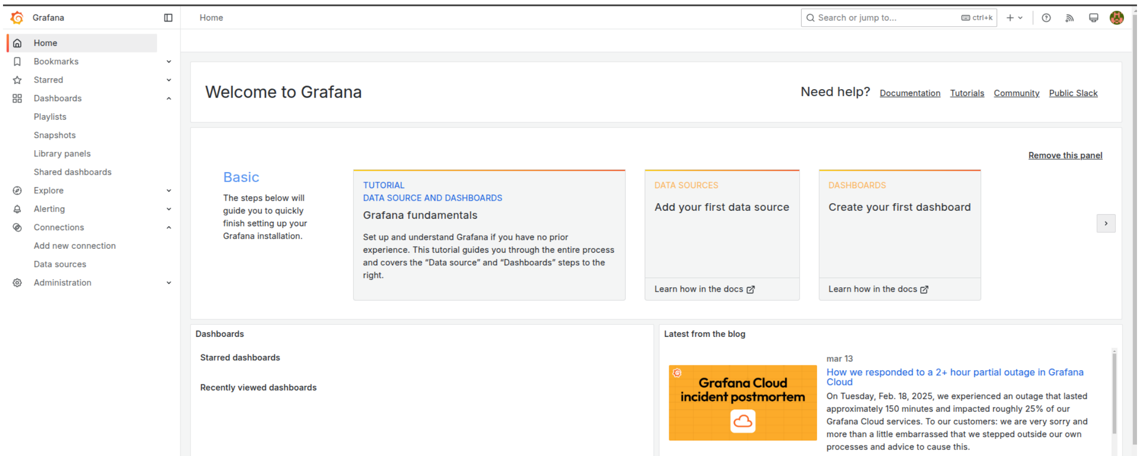
The last step to conclude this assignment was to create a dashboard in **Grafana** that allowed the visualization of the traces and metrics. So for this another service was added to the ***docker-compose.yml*** file:

```
services:
    # the rest of the services (Prometheus, Jaeger and Otel-Collector)
    grafana:
        image: grafana/grafana:latest
        container_name: grafana
        user: "0"
        ports:
          - 3000:3000
        environment:
          - GF_SECURITY_ADMIN_PASSWORD=admin
          - GF_SECURITY_ADMIN_USER=admin
          - GF_INSTALL_PLUGINS=grafana-clock-panel,grafana-simple-json-datasource
```

```
volumes:
  - ./grafana/provisioning:/etc/grafana/provisioning
  - ./grafana/dashboards:/var/lib/grafana/dashboards
depends_on:
  - jaeger
  - otel-collector
  - prometheus
networks:
  - monitoring
```

This service adds **Grafana** to the monitoring stack. It runs on port 3000 and connects to **Prometheus, Jaeger, and Otel-Collector** to display metrics and traces in a dashboard. The **admin user/password** are set to *admin* and some plugins are pre-installed like *clock-panel and simple-json-datasource*. This service has also mapped volumes, although it doesn't pre-configure dashboards, a **JSON file** is provided with the dashboard built.

With the addition of this service, when building all the services, **Grafana's UI** should be accessible through the URL http://localhost:3000/:



Then to create a new dashboard, the first step is to configure two data-sources one for **Prometheus** and another for **Jaeger**, here the ***server URLs*** should be *http://prometheus:9090* and *http://jaeger:16686.*

With the data-sources configured and the connection tested, the next step is to create the dashboard, and in this eight visualizations appear:

2.3.1 Metrics Visualizations

There are a total of four visualizations in the **Grafana dashboard** using the one metric from the default one's of the ***ASPE.NET Core*** and three others with the custom metrics.

- **Total Exceptions Occurred:** This visualization represents the number of exceptions that occurred while the application was running, it is possible to cause a change in this visualization by going to the application adding an item to the basket, after this go to basket, delete the quantity of the product and click enter (an exception should occur).

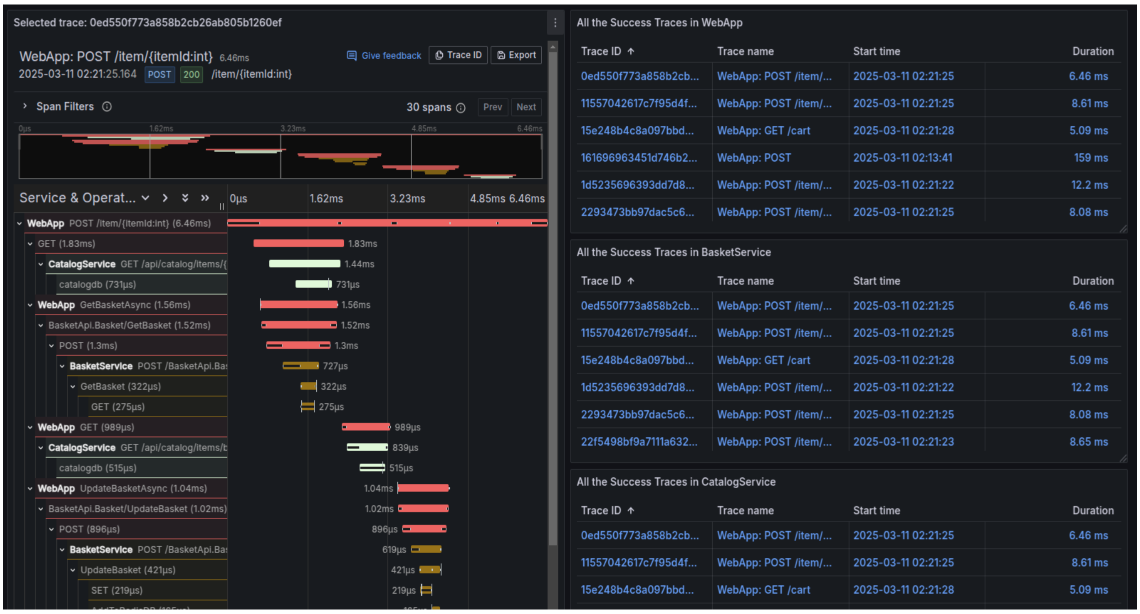
- **Rate of updates to basket:** This visualization represents the number of updates that happened to the basket per minute. To visualize a major transformation in this plot simply navigate to the application and add multiple distinct products to the basket.

- **Rate of items per minute:** This visualization represents the number of items that were added to the basket per minute. To see changes in this plot, add a big number of the same item to the basket.

- **Error Rate in Basket Service:** This visualization represents the rate of errors that occurred while adding an item to the basket per minute.

2.3.2 Traces Visualizations

For visualizing traces, three tables were added, one per each service (**Basket Service**, **Web App** and **Catalog Service**) to capture the traces with a status code of **Success (200)**, plus another section where if an id of trace is clicked on then it will appear in this section more detailed information about this trace, similar to whats visualized in Jaeger:



2.4 Mask or exclude sensitive data

In the chosen use-case, the only field that was considered sensitive data was the **User ID** and that data is only added to the traces in the Basket Service, so to mask it a processor was developed:

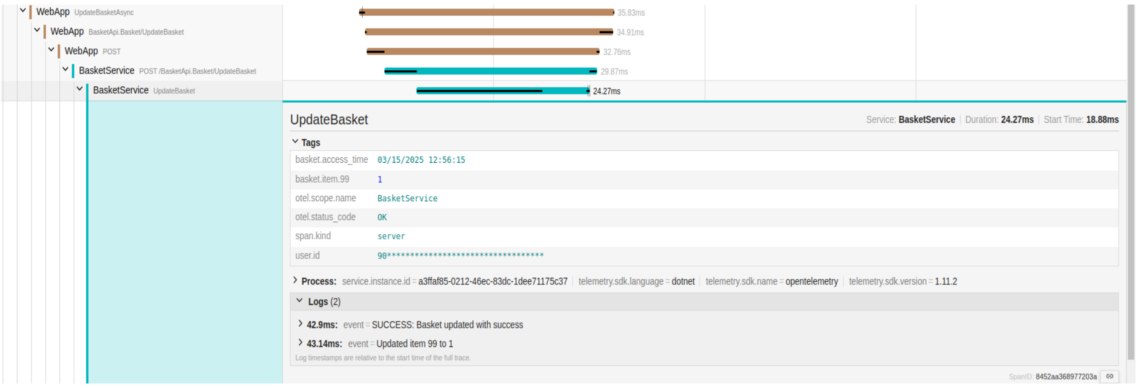
```
public class MaskingActivityProcessor : BaseProcessor<Activity>
{
    public override void OnEnd(Activity activity)
    {
        if (activity.Tags.Any(tag => tag.Key == "user.id"))
        {
            var userId =
                activity.Tags.FirstOrDefault(
                tag => tag.Key == "user.id"
            ).Value;
            var maskedUserId = MaskUserId(userId);
            activity.SetTag("user.id", maskedUserId);
        }
    }

    private static string MaskUserId(string userId)
    {
        return userId.Substring(0, 2) + new string('*', userId.Length - 2);
    }
}
```

This class masks user IDs before they are logged or exported. It extends ***Base-Processor¡Activity¿*** and overrides the ***OnEnd*** method, which runs when an **activity (trace) ends**. If the activity contains a **”user.id” tag**, it replaces most of the user ID with *, keeping only the first two characters visible.

To integrate this in this in the exporter, the following line was added in the **Program.cs** inside the **Basket.API project**:

```
builder.Services.AddOpenTelemetry()
.ConfigureResource(resource => resource
    .AddService(serviceName))
.WithTracing(tracerProviderBuilder => tracerProviderBuilder
    .AddProcessor(new MaskingActivityProcessor())
// ... rest of the configuration
```

It is possible to see the affect of this process in **Jaeger**, when capturing a trace that contains the user id, it appears like this:



3 Conclusion

In conclusion, properly configuring **traces and metrics** is essential for monitoring and improving system performance, reliability, and debugging. **Traces** help track requests across different services, making it easier to identify bottlenecks and errors, while **metrics** provide real-time insights into system health, such as request counts, latencies, and failures.

By integrating **OpenTelemetry, Prometheus, Grafana and Jaeger** a robust observability stack was created that collects, processes, and visualizes data efficiently.

ChatGPT and Gemini were used to help with the creation of the configuration files for the **Prometheus**, **Jaeger**, **Grafana** and **Otel-Collector**, explain the implementations for each file and some trouble-shooting. The configuration of **Open Telemetry** in each service was based on the documentation of the software. ***Github Copilot*** was also used to write code faster.

All the code developed and some instructions on how to execute the solution is available through this link.