universidade de aveiro deti departamento de eletrónica, telecomunicações e informática

Integrating OpenTelemetry & Security in eShop

Daniel Madureira
Nº.Mec: 107603

Professor Cláudio Teixeira

Software Architectures
DETI
Universidade de Aveiro
March 2025

Contents

1 Introduction

The purpose of this assignment was to implement observability using traces and metrics for a single feature in an eShop application. The objectives can be summarized as follows:

- **Implement OpenTelemetry tracing** for a specific feature or use case (end-to-end).
- **Mask or exclude sensitive data** (e.g., email, payment details) from telemetry and logs.
- Set up a basic **Grafana dashboard** to visualize **traces** and **metrics**.

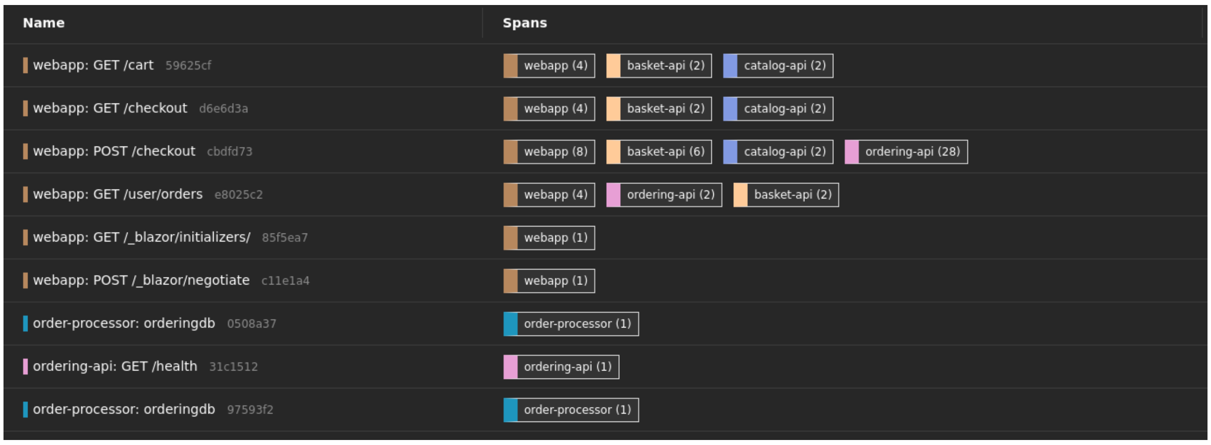
For this, we used a full-stack eShop application written in .NET, available here. Since some modifications were necessary to run the code locally on Fedora 41, the instructions for running the application, along with the code implementing my solution, are available here.

This report will explore the developed solution in more detail.

2 Implementation

As indicated in the assignment, I chose the *place an order from basket* use case to implement tracing and metrics. To achieve this, I used **Jaeger** for tracing, **Prometheus** for metrics, and the **OpenTelemetry Collector** to gather metrics via the OpenTelemetry protocol.

With the use case defined and the tech stack selected, the next step was to identify which microservices were involved in this workflow. To do this, I utilized the built-in traces dashboard:



Name	Spans
webapp: GET /cart 59625cf	webapp (4) basket-api (2) catalog-api (2)
webapp: GET /checkout d6e6d3a	webapp (4) basket-api (2) catalog-api (2)
webapp: POST /checkout cbdfd73	webapp (8) basket-api (6) catalog-api (2) ordering-api (28)
webapp: GET /user/orders e8025c2	webapp (4) ordering-api (2) basket-api (2)
webapp: GET /_blazor/initializers/ 85f5ea7	webapp (1)
webapp: POST /_blazor/negotiate c11e1a4	webapp (1)
order-processor: orderingdb 0508a37	order-processor (1)
ordering-api: GET /health 31c1512	ordering-api (1)
order-processor: orderingdb 97593f2	order-processor (1)

Figure 1: Microservices involved in the use case

From this, I determined that traces and metrics needed to be integrated into the following microservices: **Basket.API**, **Ordering.API**, **WebApp**, and **Order-Processor**.

2.1 Traces with Jaeger

2.1.1 Configuring Jaeger

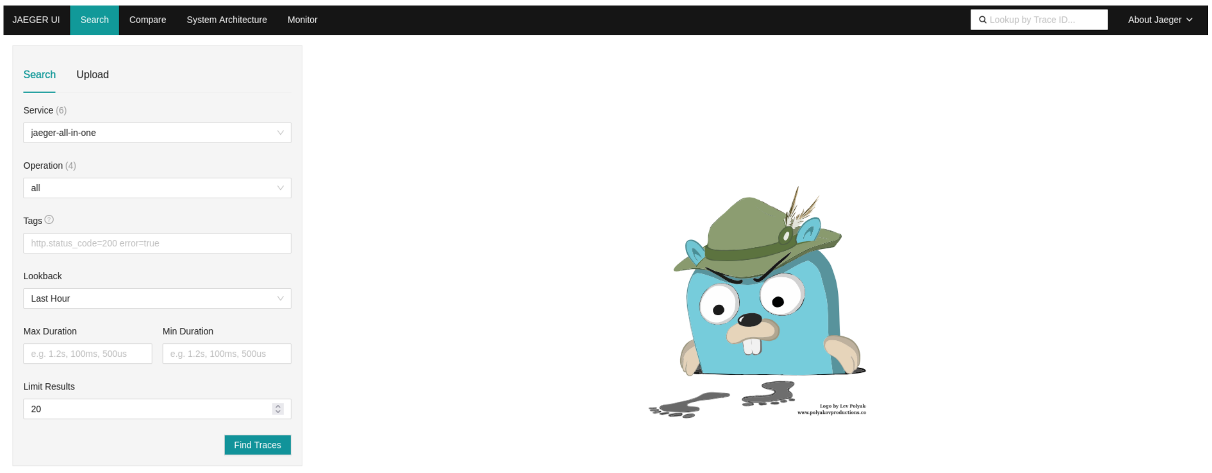
As mentioned before, I chose **Jaeger** to receive and view traces from the microservices. To achieve this, I used the **Jaeger Docker image** and included it in a `docker-compose.yml` file as follows:

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

With the **image defined** and the **container running** under the name *jaeger*, I can now access the **Jaeger UI** on **port 16686**. Additionally, I have defined **port 4317** for **OTLP communication over gRPC** and **port 4318** for **OTLP communication over HTTP**.

Since I didn't specify a custom network, all services inside this **Docker Compose** file are built within the **default network**, allowing them to communicate with each other.

By running the **Docker Compose file**, we can finally view the **Jaeger interface**.



Service (6): jaeger-all-in-one
Operation (4): all
Tags: http.status_code=200 error=true
Lookback: Last Hour
Max Duration: e.g. 1.2s, 100ms, 500us
Min Duration: e.g. 1.2s, 100ms, 500us
Limit Results: 20

Figure 2: Jaeger UI

2.1.2 Configure traces

Now that we have **Jaeger** running and working, we need to configure the microservices to send traces to it. To do this, I used **OpenTelemetry** to export traces over **gRPC** to **Jaeger** by modifying the `Program.cs` file in each microservice.

For example, in the **Basket.API** microservice, I added the following code:

```
const string serviceName = "eShop.Basket.API";

// Configure OpenTelemetry
builder.Services.AddOpenTelemetry()
    .WithTracing(
        (tracing) =>
        {
            tracing.AddSource(serviceName);
            tracing.AddProcessor(new MaskProcessor());
            tracing.AddAspNetCoreInstrumentation();
            tracing.AddHttpClientInstrumentation();
            tracing.AddGrpcClientInstrumentation();
            tracing.AddOtlpExporter(
                options =>
                {
                    options.Endpoint = new Uri("http://localhost:4317");
                    options.Protocol = OtlpExportProtocol.Grpc;
                }
            );
        }
    );
```

Looking deeper into the code, we can see where we:

- **Configure OpenTelemetry Tracing:** The `.WithTracing` method is used to set up tracing, enabling the collection and recording of application execution details, such as requests, responses, and events.
- **Add a Telemetry Source:** The `tracing.AddSource(serviceName)` method identifies the service that is generating telemetry data, helping to distinguish between different data sources.
- **Mask Sensitive Data:** By using `tracing.AddProcessor(new MaskProcessor())`, a custom processor is added to filter or mask sensitive information before it is exported, ensuring data privacy.
- **Enable Instrumentation:**
  - `tracing.AddAspNetCoreInstrumentation()` captures details of incoming HTTP requests in ASP.NET Core applications.
  - `tracing.AddHttpClientInstrumentation()` tracks and records outgoing HTTP requests.
  - `tracing.AddGrpcClientInstrumentation()` captures traces of gRPC client calls.
- **Configure OTLP Exporter:** The `tracing.AddOtlpExporter` method configures the exporter to send telemetry data to an OTLP collector over gRPC at `http://localhost:4317`, ensuring that the data is transmitted accurately for storage and analysis.

This ensures that the **Basket.API** service collects and forwards data to **Jaeger**. After this, I added custom traces to the services. I will showcase the work done on the **Basket.API**, but the other microservices were configured using the same principles.

In this case, I created a custom `ActivitySource` with the same `serviceName` as the one configured inside the `Program.cs` file:

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

This `ActivitySource` is used to create and manage tracing activities within the `BasketService` class. Inside each function (`GetBasket`, `UpdateBasket`, and `DeleteBasket`), a new activity is started using:

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

We then collect **custom traces** by setting **tags** as follows:

```
activity?.SetTag("user.id", context.GetUserIdentity());
activity?.SetTag("request.method", context.Method);

var userId = context.GetUserIdentity();
if (string.IsNullOrEmpty(userId))
{
    activity?.SetStatus(ActivityStatusCode.Error, "User ID is null or
     ↪  empty");
    return new();
}

if (logger.IsEnabled(LogLevel.Debug))
{
    logger.LogDebug("Begin GetBasketById call from method {Method} for
     ↪  basket id {Id}", context.Method, userId);
}

var data = await repository.GetBasketAsync(userId);

if (data is not null)
{
    activity?.SetStatus(ActivityStatusCode.Ok, "Basket retrieved
     ↪  successfully");
    return MapToCustomerBasketResponse(data);
}

activity?.SetStatus(ActivityStatusCode.Error, "Basket not found");
return new();
```

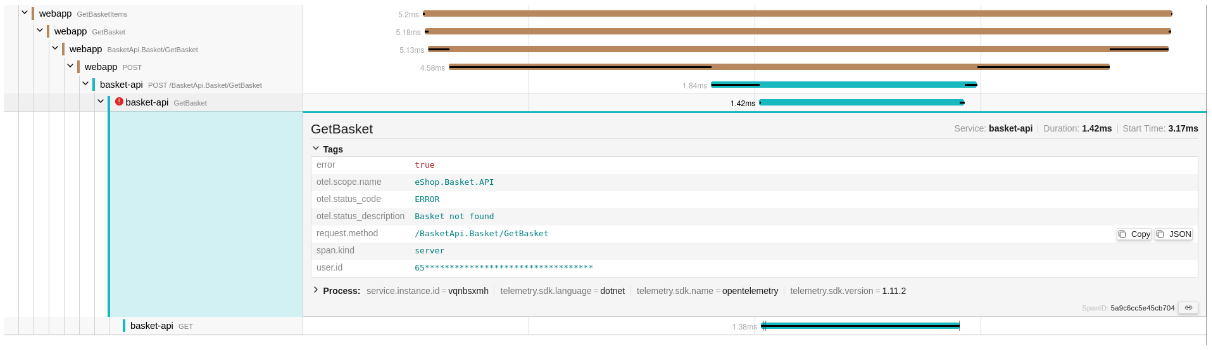
In this case, we:

- **Add Tags:**

  - `activity?.SetTag("user.id", context.GetUserIdentity())` attaches the user identity to the trace for better context.
  - `activity?.SetTag("request.method", context.Method)` adds the gRPC method name, providing additional context for the trace.

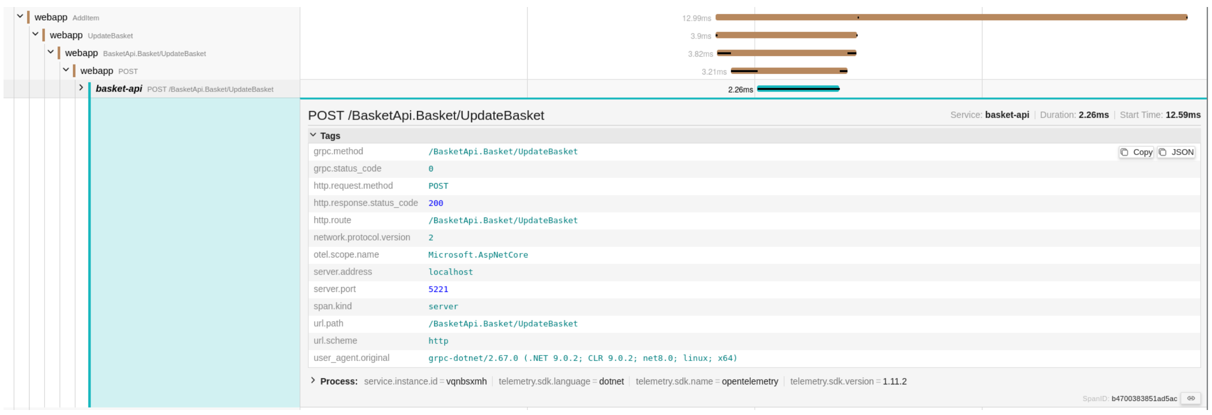
- **Set Status:**
  - If the user ID is **null or empty**, the trace status is marked as **Error**.
  - If the basket is **successfully retrieved**, the status is set to **OK**.
  - If the basket is **not found**, the trace status is marked as **Error**.

These traces can then be visualized in **Jaeger UI** under **basket-api**.



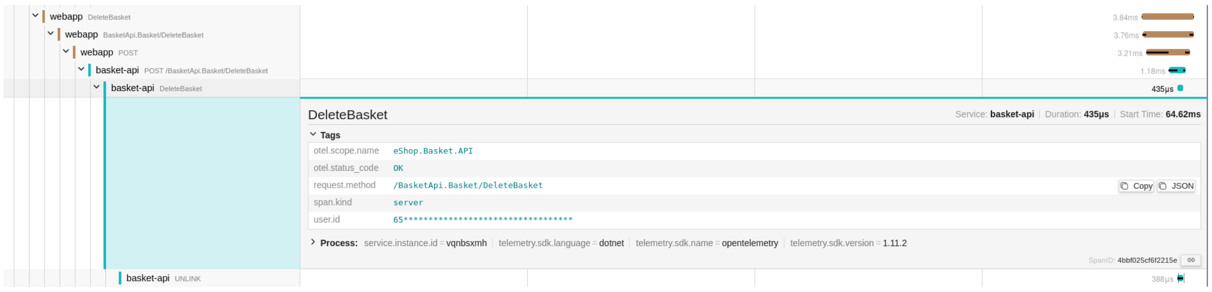
```
webapp GetBasketItems                                  5.2ms
  webapp GetBasket                                     5.18ms
    webapp BasketApi.Basket/GetBasket                  5.13ms
      webapp POST                                      4.58ms
        basket-api POST /BasketApi.Basket/GetBasket    1.84ms
          basket-api GetBasket                         1.42ms

GetBasket                Service: basket-api | Duration: 1.42ms | Start Time: 3.17ms
Tags
error                   true
otel.scope.name         eShop.Basket.API
otel.status_code        ERROR
otel.status_description Basket not found
request.method          /BasketApi.Basket/GetBasket                Copy  JSON
span.kind               server
user.id                 65**********************************
Process: service.instance.id = vqnbsxmh | telemetry.sdk.language = dotnet | telemetry.sdk.name = opentelemetry | telemetry.sdk.version = 1.11.2
                                                         SpanID: 5a9c6cc5e45cb704

          basket-api GET                               1.38ms
```

Figure 3: GetBasket traces



```
webapp AddItem                                           12.99ms
  webapp UpdateBasket                                    3.9ms
    webapp BasketApi.Basket/UpdateBasket                 3.82ms
      webapp POST                                        3.21ms
        basket-api POST /BasketApi.Basket/UpdateBasket   2.26ms

POST /BasketApi.Basket/UpdateBasket   Service: basket-api | Duration: 2.26ms | Start Time: 12.59ms
Tags
grpc.method                /BasketApi.Basket/UpdateBasket          Copy  JSON
grpc.status_code           0
http.request.method        POST
http.response.status_code  200
http.route                 /BasketApi.Basket/UpdateBasket
network.protocol.version   2
otel.scope.name            Microsoft.AspNetCore
server.address             localhost
server.port                5221
span.kind                  server
url.path                   /BasketApi.Basket/UpdateBasket
url.scheme                 http
user_agent.original        grpc-dotnet/2.67.0 (.NET 9.0.2; CLR 9.0.2; net8.0; linux; x64)
Process: service.instance.id = vqnbsxmh | telemetry.sdk.language = dotnet | telemetry.sdk.name = opentelemetry | telemetry.sdk.version = 1.11.2
                                                         SpanID: b4700383851ad5ac
```

Figure 4: UpdateBasket traces



webapp DeleteBasket 3.84ms
webapp BasketApi.Basket/DeleteBasket 3.76ms
webapp POST 3.21ms
basket-api POST /BasketApi.Basket/DeleteBasket 1.18ms
basket-api DeleteBasket 435μs

DeleteBasket
Service: **basket-api** | Duration: **435μs** | Start Time: **64.62ms**

Tags

otel.scope.name	`eShop.Basket.API`
otel.status_code	`OK`
request.method	`/BasketApi.Basket/DeleteBasket`
span.kind	`server`
user.id	`65*********************************`

Process: service.instance.id = vqnbsxmh | telemetry.sdk.language = dotnet | telemetry.sdk.name = opentelemetry | telemetry.sdk.version = 1.11.2

SpanID: 4bbf025cf6f2215e

basket-api UNLINK 388μs

Figure 5: DeleteBasket traces

2.2 Metrics with Prometheus

2.2.1 Configure Prometheus

In order to collect metrics, I chose to use **Prometheus** and **OpenTelemetry Collector** (**otel-collector**) to collect and store the metrics defined in the application. For this purpose, I created two configuration files, `prometheus.yml` and `otel-collector/config.yml`, that hold the configuration for these two services.

The **otel-collector** is responsible for collecting metrics from the different services and exporting them to **Prometheus**, acting as a middleman. Without the otel-collector, Prometheus would need to pull the metrics from each service individually. However, by using the otel-collector, we can collect, process, and send metrics to Prometheus in a more structured and efficient way. In this case, the otel-collector receives data over **gRPC** on port **4316** or **HTTP** on port **4315**, and exports them to the Prometheus endpoint on port **8889**:

```yaml
#otel-collector/config.yml
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4316
      http:
        endpoint: 0.0.0.0:4315

exporters:
  prometheus:
    endpoint: 0.0.0.0:8889

service:
  pipelines:
    metrics:
      receivers: [otlp]
      exporters: [prometheus]
```

On the Prometheus side, it is configured to collect data every **15 seconds** from the defined endpoint. In this case, Prometheus collects data from the **otel-collector** on port **8889**. The configuration for this is defined in the `prometheus.yml` file:

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

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

Similar to the Jaeger setup, I used the respective **Docker images** to launch these services on the same network as Jaeger by adding them to the `docker-compose.yml` file. Here's the configuration:

```
  # ... jaeger service
  otel-collector:
    container_name: otel-collector
    image: otel/opentelemetry-collector-contrib
    volumes:
      - ./observability/otel-collector/config.yaml:
       ↪  /etc/otelcol-contrib/config.yaml
    ports:
      - 1888:1888 # pprof extension
      - 8888:8888 # Prometheus metrics exposed by the Collector
      - 8889:8889 # Prometheus exporter metrics
      - 13133:13133 # health_check extension
      - 4316:4316 # OTLP gRPC receiver
      - 4315:4315 # OTLP http receiver
      - 55679:55679 # zpages extension
    depends_on:
      - prometheus

  prometheus:
    container_name: prometheus
    image: prom/prometheus:latest
    volumes:
      - ./observability/prometheus/prometheus.yml:
       ↪  /etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"
```

After relaunching the Docker Compose file, we can now see two additional services running: the **otel-collector** service is running on port **4316** to receive metrics over **gRPC**, and on port **8889** to export them to Prometheus. Additionally, the **Prometheus UI** is now accessible on port **9090**, where we can view and analyze the collected metrics.

2.2.2 Configure Custom Metrics

Similar to adding tracing to the microservices, I had to modify the `Program.cs` file in each microservice to enable the collection of metrics. Once again, I will use the `Basket.API`

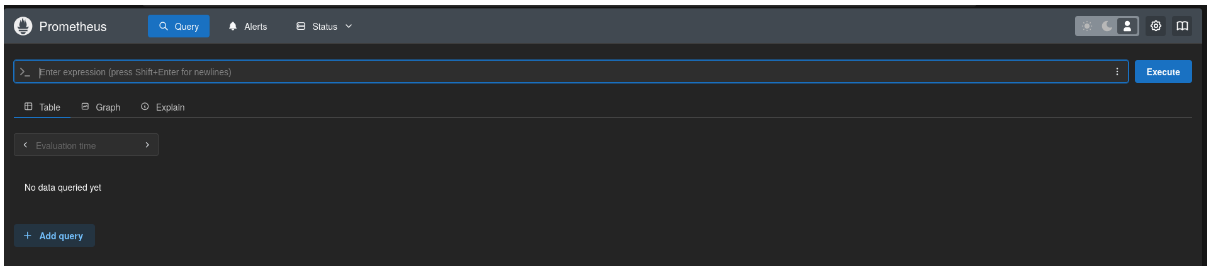


Figure 6: Prometheus UI

microservice as an example, but the other microservices in the use case had similar changes made to them.

To collect the metrics, I added the following code to `Program.cs`:

```
// builder.Services.AddOpenTelemetry()
// .WithTracing(
//     ...tracing config
// )
.WithMetrics(
    metrics =>
    {
        metrics.AddHttpClientInstrumentation();
        metrics.AddAspNetCoreInstrumentation();
        metrics.AddMeter(serviceName, serviceVersion);
        metrics.AddOtlpExporter(
            options =>
            {
                options.Endpoint = new Uri("http://localhost:4316");
            }
        );
    })
```

Looking at the changes, we can identify the following:

- **Metrics Collection Setup:** Initializes OpenTelemetry to collect performance metrics such as latency, errors, and throughput.
- **Outgoing HTTP Request Metrics:** Tracks external HTTP requests, capturing metrics like request count, response time, and errors.
- **ASP.NET Core HTTP Request Metrics:** Captures incoming HTTP request data, including count, duration, and response status.
- **Meter Setup:** Records metrics tied to a specific service and version, providing better tracking.
- **OTLP Exporter Configuration:** Sends collected metrics to the `otel-collector` via OTLP for analysis and processing.

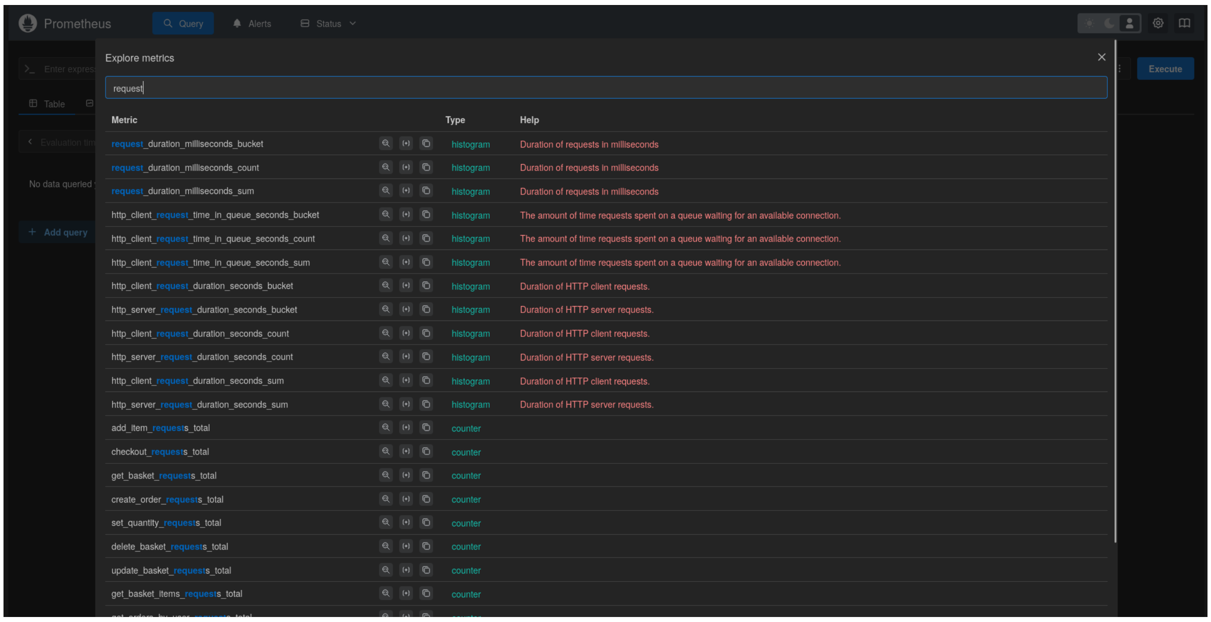
On the other side, I added custom metrics to the `BasketService.cs` using counters and histograms. The following code defines these metrics:

```
private static readonly Meter Meter = new Meter("eShop.Basket.API",
 ↪  "1.0.0");
private static readonly Counter<int> GetBasketCounter =
 ↪  Meter.CreateCounter<int>("get_basket_requests");
private static readonly Counter<int> UpdateBasketCounter =
 ↪  Meter.CreateCounter<int>("update_basket_requests");
private static readonly Counter<int> DeleteBasketCounter =
 ↪  Meter.CreateCounter<int>("delete_basket_requests");
private static readonly Histogram<double> RequestDurationHistogram =
 ↪  Meter.CreateHistogram<double>("request_duration", "ms", "Duration of
 ↪  requests in milliseconds");
```

Here, I defined a meter group under the service name `eShop.Basket.API` and created counters to track the number of times requests are made to get, update, or delete the basket. I also created a histogram to measure the duration of requests. These counters and histograms are updated in their respective functions. For example, in the `GetBasket` function:

```
public override async Task<CustomerBasketResponse>
 ↪  GetBasket(GetBasketRequest request, ServerCallContext context)
{
    GetBasketCounter.Add(1);

    // ... define traces

    var stopwatch = Stopwatch.StartNew();

    // ... get basket logic

    stopwatch.Stop();
    RequestDurationHistogram.Record(stopwatch.ElapsedMilliseconds, new
     ↪  KeyValuePair<string, object>("method", "GetBasket"));

    // ... rest of the function
}
```

With everything configured, when we access the **Prometheus UI** and explore the available metrics, we can see both our custom metrics and some default metrics from `ASP.NET`.



Metric	Type	Help
request_duration_milliseconds_bucket	histogram	Duration of requests in milliseconds
request_duration_milliseconds_count	histogram	Duration of requests in milliseconds
request_duration_milliseconds_sum	histogram	Duration of requests in milliseconds
http_client_request_time_in_queue_seconds_bucket	histogram	The amount of time requests spent on a queue waiting for an available connection.
http_client_request_time_in_queue_seconds_count	histogram	The amount of time requests spent on a queue waiting for an available connection.
http_client_request_time_in_queue_seconds_sum	histogram	The amount of time requests spent on a queue waiting for an available connection.
http_client_request_duration_seconds_bucket	histogram	Duration of HTTP client requests.
http_server_request_duration_seconds_bucket	histogram	Duration of HTTP server requests.
http_client_request_duration_seconds_count	histogram	Duration of HTTP client requests.
http_server_request_duration_seconds_count	histogram	Duration of HTTP server requests.
http_client_request_duration_seconds_sum	histogram	Duration of HTTP client requests.
http_server_request_duration_seconds_sum	histogram	Duration of HTTP server requests.
add_item_requests_total	counter	
checkout_requests_total	counter	
get_basket_requests_total	counter	
create_order_requests_total	counter	
set_quantity_requests_total	counter	
delete_basket_requests_total	counter	
update_basket_requests_total	counter	
get_basket_items_requests_total	counter	

Figure 7: Metrics from Prometheus

2.3 Masking sensitive data

As my use case involves sensitive information, such as payment details like card numbers and security codes, there was a need to either avoid tracing this kind of data or at least mask it in the traces. To achieve this, when exporting metrics to Jaeger, I added an `ActivityProcessor` to mask sensitive data.

Here is the code for the custom `MaskProcessor`:

```
public class MaskProcessor : 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 = MaskString(userId);
            activity.SetTag("user.id", maskedUserId);
        }
        if (activity.Tags.Any(tag => tag.Key == "request.command.buyer"))
        {
            var buyer = activity.Tags.FirstOrDefault(tag => tag.Key ==
             ↪  "request.command.buyer").Value;
            var maskedBuyer = MaskString(buyer);
            activity.SetTag("request.command.buyer", maskedBuyer);
        }
        if (activity.Tags.Any(tag => tag.Key ==
         ↪  "request.command.buyerId"))
        {
```

```
    var buyerId = activity.Tags.FirstOrDefault(tag => tag.Key ==
    ↪  "request.command.buyerId").Value;
    var maskedBuyerId = MaskString(buyerId);
    activity.SetTag("request.command.buyerId", maskedBuyerId);
}
if (activity.Tags.Any(tag => tag.Key ==
↪  "request.command.cardNumber"))
{
    var cardNumber = activity.Tags.FirstOrDefault(tag => tag.Key
    ↪  == "request.command.cardNumber").Value;
    var maskedCardNumber = MaskString(cardNumber);
    activity.SetTag("request.command.cardNumber",
    ↪  maskedCardNumber);
}
if (activity.Tags.Any(tag => tag.Key ==
↪  "request.command.cardHolderName"))
{
    var cardHolderName = activity.Tags.FirstOrDefault(tag =>
    ↪  tag.Key == "request.command.cardHolderName").Value;
    var maskedCardHolderName = MaskString(cardHolderName);
    activity.SetTag("request.command.cardHolderName",
    ↪  maskedCardHolderName);
}
if (activity.Tags.Any(tag => tag.Key ==
↪  "request.command.cardExpiration"))
{
    var cardExpiration = activity.Tags.FirstOrDefault(tag =>
    ↪  tag.Key == "request.command.cardExpiration").Value;
    var maskedCardExpiration = MaskString(cardExpiration);
    activity.SetTag("request.command.cardExpiration",
    ↪  maskedCardExpiration);
}
if (activity.Tags.Any(tag => tag.Key ==
↪  "request.command.cardSecurityNumber"))
{
    var cardSecurityNumber = activity.Tags.FirstOrDefault(tag =>
    ↪  tag.Key == "request.command.cardSecurityNumber").Value;
    var maskedCardSecurityNumber = MaskCVV(cardSecurityNumber);
    activity.SetTag("request.command.cardSecurityNumber",
    ↪  maskedCardSecurityNumber);
}
if (activity.Tags.Any(tag => tag.Key ==
↪  "request.command.Street"))
{
    var street = activity.Tags.FirstOrDefault(tag => tag.Key ==
    ↪  "request.command.Street").Value;
    var maskedStreet = MaskString(street);
    activity.SetTag("request.command.Street", maskedStreet);
```

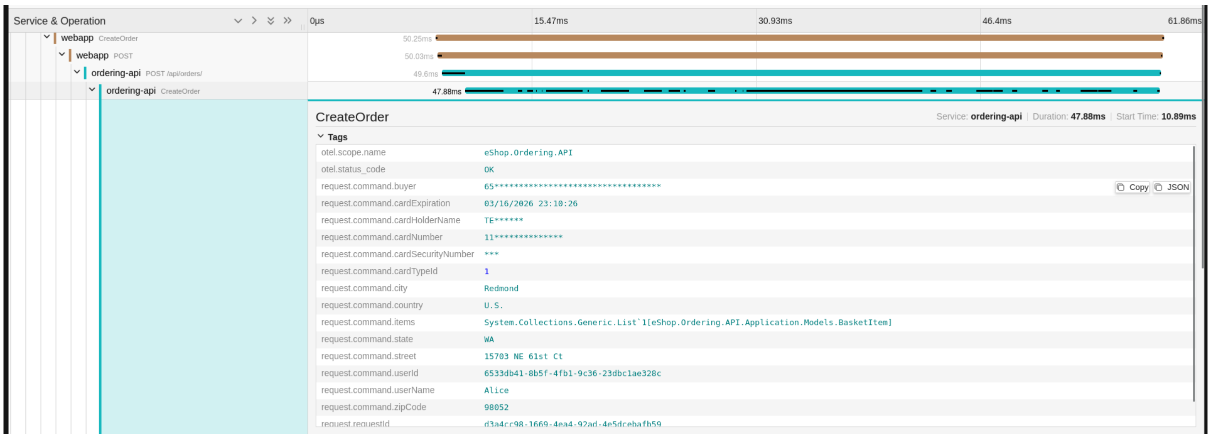
```
        }
        if (activity.Tags.Any(tag => tag.Key ==
         ↪  "request.command.ZipCode"))
        {
            var zipCode = activity.Tags.FirstOrDefault(tag => tag.Key ==
             ↪  "request.command.ZipCode").Value;
            var maskedZipCode = MaskString(zipCode);
            activity.SetTag("request.command.ZipCode", maskedZipCode);
        }
    }

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

    private static string MaskCVV(string str)
    {
        return new string('*', str.Length);
    }
}
```

This class processes telemetry data, specifically activities, and masks sensitive information before it is exported. The `OnEnd` method is overridden to apply the masking operations when an activity ends. Inside the `OnEnd` method, we define the tags containing sensitive information and apply a mask accordingly.

The `MaskProcessor` is then added to the tracing configuration in the `Program.cs` file of each microservice, as shown below:

```
// config openTelemetry
builder.Services.AddOpenTelemetry()
    .WithTracing(
        (tracing) =>
        {
            \\ ... rest of implementation

            tracing.AddProcessor(new MaskProcessor());

            \\ ... rest of implementation
        }
    );
```

As a result, we can confirm that the sensitive fields are masked in the traces, ensuring that no sensitive information is exposed:



CreateOrder

Service: **ordering-api** | Duration: **47.88ms** | Start Time: **10.89ms**

Tags

otel.scope.name	`eShop.Ordering.API`
otel.status_code	`OK`
request.command.buyer	`65**********************************`
request.command.cardExpiration	`03/16/2026 23:10:26`
request.command.cardHolderName	`TE******`
request.command.cardNumber	`11**************`
request.command.cardSecurityNumber	`***`
request.command.cardTypeId	`1`
request.command.city	`Redmond`
request.command.country	`U.S.`
request.command.items	``System.Collections.Generic.List`1[eShop.Ordering.API.Application.Models.BasketItem]``
request.command.state	`WA`
request.command.street	`15703 NE 61st Ct`
request.command.userId	`6533db41-8b5f-4fb1-9c36-23dbc1ae328c`
request.command.userName	`Alice`
request.command.zipCode	`98052`
request.requestId	`d3a4cc98-1669-4ea4-92ad-4e5dcebafb59`

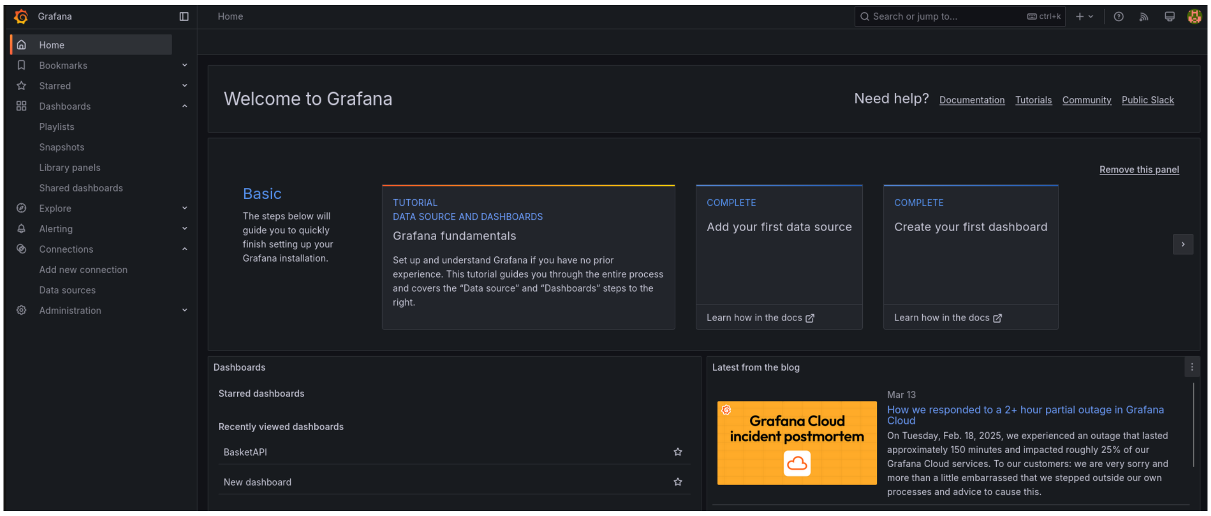
Figure 8: Masked fields in the request

2.4 Visualization with Grafana

After collecting traces and metrics, we need to create a dashboard to visualize the data over time as users interact with the application. To achieve this, I used **Grafana**. Once again, I leveraged Docker images and added them to the Docker Compose file:

```yaml
services:
    # ... rest of services
    grafana:
        container_name: grafana
        image: grafana/grafana:latest
        ports:
          - "3000:3000"
        environment:
          - GF_SECURITY_ADMIN_PASSWORD=admin
        volumes:
          - ./observability/grafana/provisioning:
           ↪  /var/lib/grafana/provisioning
          - ./observability/grafana/dashboards:
           ↪  /var/lib/grafana/dashboards
```

When we run the Docker Compose file, **Grafana** will be running on **port 3000**. It connects to **Prometheus**, **Jaeger**, and **otel-collector** to display the metrics and traces collected by these services in custom dashboards. By default, **Grafana** comes with an admin login, with both the **username** and **password** set to 'admin'. It also maps volumes, though it doesn't provide default dashboards. However, Grafana can import **JSON files** with pre-configured dashboards.



Welcome to Grafana

Need help? Documentation Tutorials Community Public Slack

Basic

The steps below will guide you to quickly finish setting up your Grafana installation.

TUTORIAL
DATA SOURCE AND DASHBOARDS
Grafana fundamentals

Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.

COMPLETE
Add your first data source
Learn how in the docs

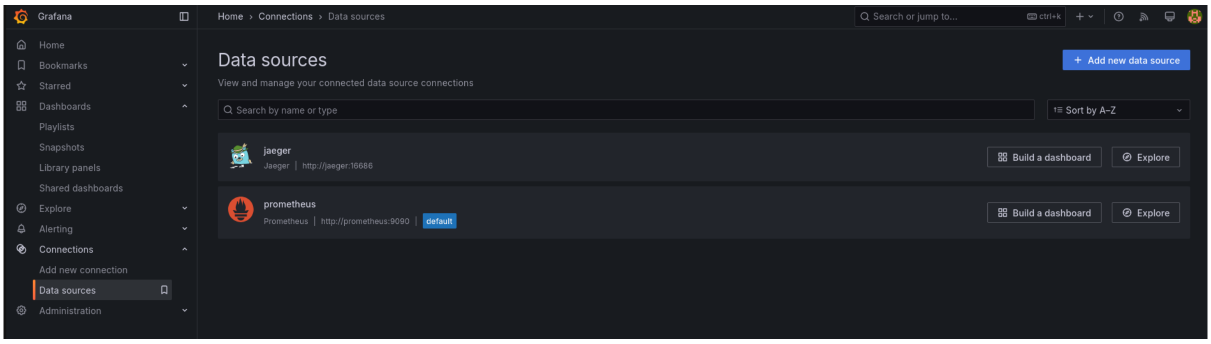
COMPLETE
Create your first dashboard
Learn how in the docs

Mar 13
How we responded to a 2+ hour partial outage in Grafana Cloud
On Tuesday, Feb. 18, 2025, we experienced an outage that lasted approximately 150 minutes and impacted roughly 25% of our Grafana Cloud services. To our customers: we are very sorry and more than a little embarrassed that we stepped outside our own processes and advice to cause this.

Figure 9: Grafana UI

To create a new dashboard, we first need to configure the **data sources** by adding the **Prometheus** and **Jaeger** URLs:

- `http://prometheus:9090`
- `http://jaeger:16686`



Data sources
View and manage your connected data source connections

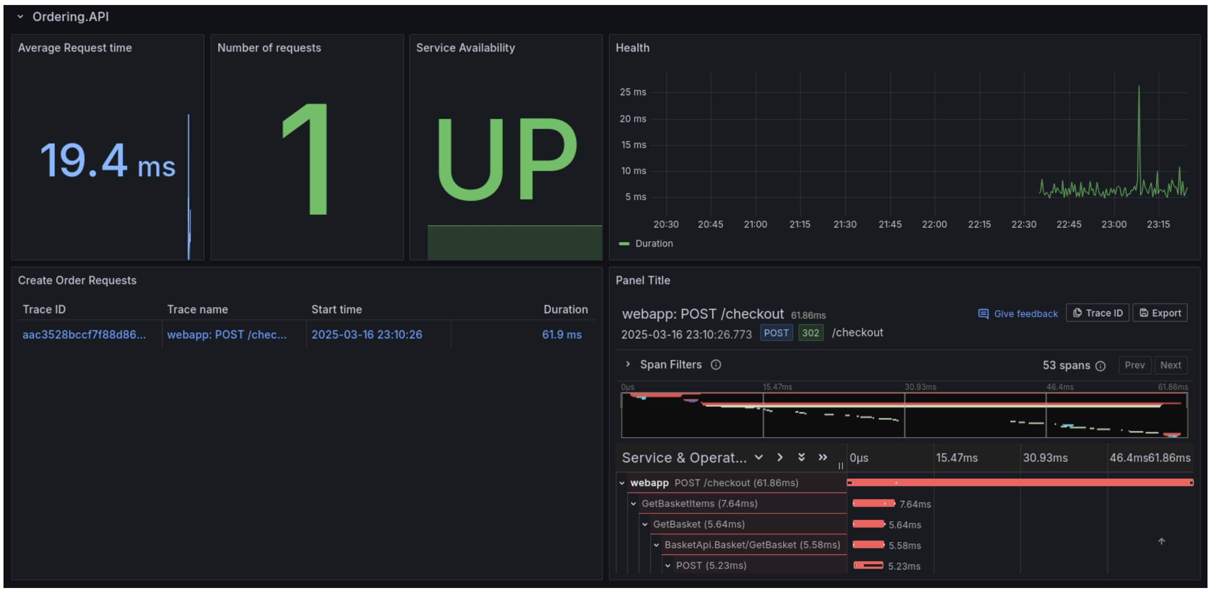
jaeger
Jaeger | http://jaeger:16686

prometheus
Prometheus | http://prometheus:9090 | default

Figure 10: Data sources in Grafana

2.4.1 Dashboards

Having the data flowing from the sources, I built a dashboard to display some metrics and traces that the services collect over time for two microservices: **Basket.API** and **Ordering.API**:



Trace ID	Trace name	Start time	Duration
aac3528bccf7f88d86...	webapp: POST /chec...	2025-03-16 23:10:26	61.9 ms

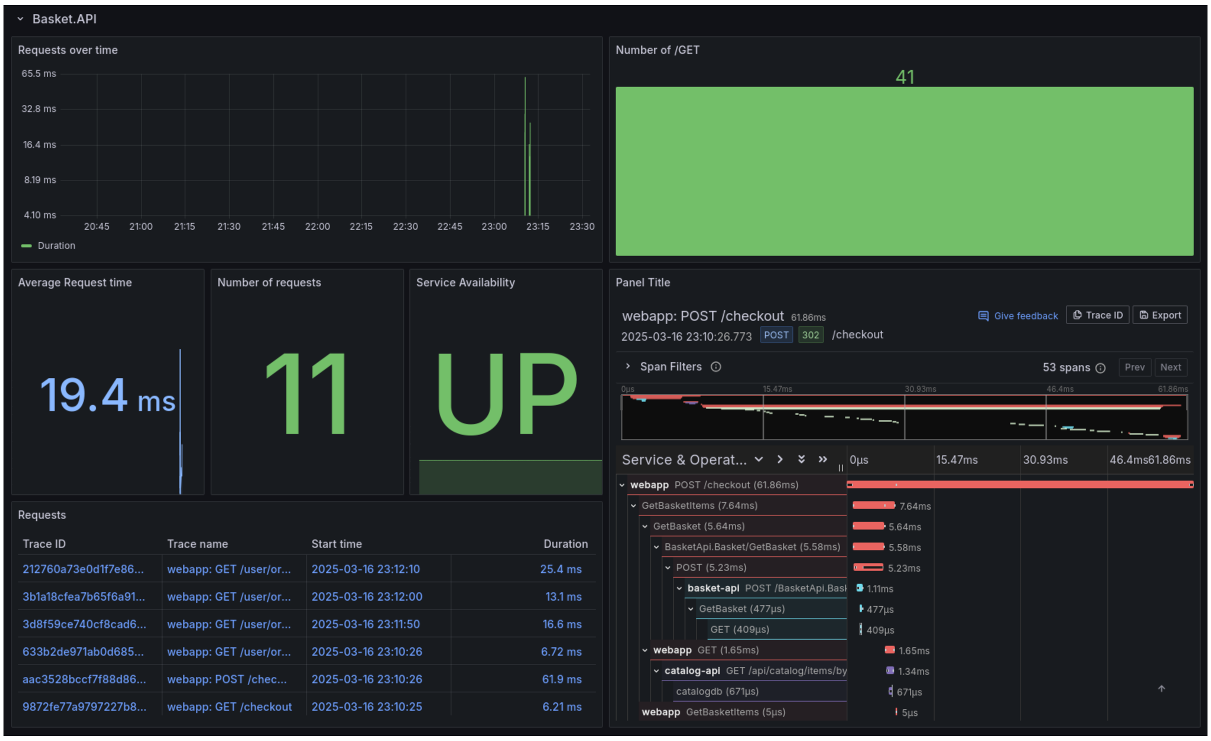
webapp: POST /checkout 61.86ms
2025-03-16 23:10:26.773 POST 302 /checkout

```
webapp POST /checkout (61.86ms)
  GetBasketItems (7.64ms)          7.64ms
    GetBasket (5.64ms)             5.64ms
      BasketApi.Basket/GetBasket (5.58ms)   5.58ms
        POST (5.23ms)              5.23ms
```

Figure 11: Ordering dashboard

Here, we display the following:

- **Average request time**: The average time for each request to the Ordering API.
- **Number of requests**: This metric counts the number of requests made to the Ordering API, collected by **Jaeger**.
- **Service availability**: This data is gathered from the health checks in **ASP.NET**.
- **Health**: A Time Series Graph that displays the API response times over time, showing fluctuations and potential anomalies.
- **Create Order Requests (Table)**: A table with traces collected by **Jaeger**, with trace details shown on the side.



Requests

Trace ID	Trace name	Start time	Duration
212760a73e0d1f7e86...	webapp: GET /user/or...	2025-03-16 23:12:10	25.4 ms
3b1a18cfea7b65f6a91...	webapp: GET /user/or...	2025-03-16 23:12:00	13.1 ms
3d8f59ce740cf8cad6...	webapp: GET /user/or...	2025-03-16 23:11:50	16.6 ms
633b2de971ab0d685...	webapp: GET /user/or...	2025-03-16 23:10:26	6.72 ms
aac3528bccf7f88d86...	webapp: POST /chec...	2025-03-16 23:10:26	61.9 ms
9872fe77a9797227b8...	webapp: GET /checkout	2025-03-16 23:10:25	6.21 ms

```
webapp: POST /checkout 61.86ms
2025-03-16 23:10:26.773 POST 302 /checkout

0µs  15.47ms  30.93ms  46.4ms  61.86ms
webapp POST /checkout (61.86ms)
  GetBasketItems (7.64ms)                          7.64ms
    GetBasket (5.64ms)                             5.64ms
      BasketApi.Basket/GetBasket (5.58ms)          5.58ms
        POST (5.23ms)                              5.23ms
          basket-api POST /BasketApi.Bask          1.11ms
            GetBasket (477µs)                      477µs
              GET (409µs)                          409µs
    webapp GET (1.65ms)                            1.65ms
      catalog-api GET /api/catalog/items/by        1.34ms
        catalogdb (671µs)                          671µs
  webapp GetBasketItems (5µs)                      5µs
```

Figure 12: Basket Dashboard

Here, we display the following:

- **Requests over time**: A Time Series Graph that displays the API response times over time.
- **Number of /gets**: This metric comes from **Prometheus**, specifically from the meter that counts the number of `/GET` requests to the service.
- **Average request time**: The average time for each request to the Basket API.
- **Service availability**: This data comes from the health checks in **ASP.NET**.
- **Number of requests**: This metric counts the number of requests made to the Basket API, collected by **Jaeger**.
- **Requests (Table)**: A table with traces collected by **Jaeger**, with trace details shown on the side.

3 Conclusion

In this report, I explored the process of implementing **observability** in a **microservices architecture** using tools such as **OpenTelemetry**, **Jaeger**, **Prometheus**, and **Grafana**. By instrumenting the microservices with **tracing** and **metrics collection**, we were able to gain valuable insights into the **performance** and **health** of the services, which are critical for maintaining a reliable and efficient application.

During the process, I encountered several challenges. The first challenge was understanding how the application was built and how it functioned. The second was learning how to incorporate **OpenTelemetry**, **Prometheus**, and **Jaeger** with an app written in **.NET** and **C#**, tools I was not very familiar with. After overcoming this, I struggled with ensuring the communication between services and establishing proper connectivity with **Grafana**. One particular challenge was automating the saving of dashboards in **Grafana**, which, in the end, I was unable to achieve.

To help overcome these challenges, I leveraged **ChatGPT**, **Gemini**, and **Copilot**. These AI-powered tools assisted me in generating **configuration files**, exploring and navigating **C#** code, and troubleshooting issues I encountered during development. Additionally, **Copilot** helped me write code faster by suggesting relevant snippets and improving my coding efficiency.

In the end, I faced difficulties in creating **load tests** with **k6** to test the app due to time constraints, as well as explore data encryption and compliance in the **database** layer and introduce **column masking** for sensitive data, ending up not implementing these topics.

All the code developed,a s well as the instructions to run it are available in my repository via this link (https://github.com/Dan1m4D/AS-Assignment-1).