universidade
de aveiro

14/02/2025

Assignment 1 - AS
Integrating OpenTelemetry & Security in eShop

Miguel Miragaia, 108317

Departamento de Electrónica, Telecomunicações e Informática

Mestrado em Engenharia Informática

1. Introduction

Modern applications require **observability** to ensure system reliability, performance, and security. This integration enhances eShop's monitoring and security by incorporating **OpenTelemetry** for distributed tracing, **Prometheus** for metrics collection, **Grafana** for visualization, and **Jaeger** for trace analysis.

During the development of this project I used **Copilot** for code assistant as it has access to my codebase and some context. It demonstrated that it is very helpful in some situations, as debugging with the ability to add the logs on the terminal for better context was a good experience. In coding development and feature integration I think it was not as great as it was in debugging, when asked for a feature implementation or configuration most of the times it couldn't make the full implementation as it lacked some crucial steps. If the feature needed code addition in various files the copilot was only able to give me code for a few of those, forcing me to interpret everything he tried to do and figure out the missing step.

ChatGPT was also used, but without any context of my codebase, which made it more difficult. Its usage was more focused on the understating of what I needed to implement and the necessary steps to do it. When the copilot was giving bad/incomplete instructions ChatGPT was what I was using to try to understand what was wrong.

After the project presentation during class, I have accomplished two more of the defined objectives. Mask Sensitive data and Load Testing, these features are documented along the report.

1.1. Objectives of the Integration

The primary goal of this integration is to improve observability and security in the **basket checkout workflow**. This includes:

1. **End-to-End Tracing:**
   - Implement OpenTelemetry tracing across the Blazor UI, BasketState, and backend services.
   - Track request flow and measure system performance.
   - Use activity tags to enrich trace data for better insights.
   - User Jaeger to visualize the traces.
2. **Monitoring & Metrics Collection:**
   - Capture counter and histogram metrics for key operations.
   - Export metrics to Prometheus, allowing real-time monitoring.
   - Use Grafana dashboards to visualize performance and system behavior.
3. **Security Enhancements**
   - Implemented a data masking strategy to prevent logging of sensitive information (e.g., userId, emails, payment details).
4. **Load Testing:**
   - Using locust I generated activity on the basket to test the workflow.

1.2. Technologies Used

It was used several observability and monitoring tools, such as:

- **OpenTelemetry** – For tracing API calls and capturing performance metrics.
- **Prometheus** – To store and expose application metrics.
- **Grafana** – To visualize real-time system behavior using custom dashboards.
- **Jaeger** – To analyze distributed traces.
- **Docker & Docker Compose –** To containerize and orchestrate the observability stack.
- **Locust -** To generate activity on the webapp

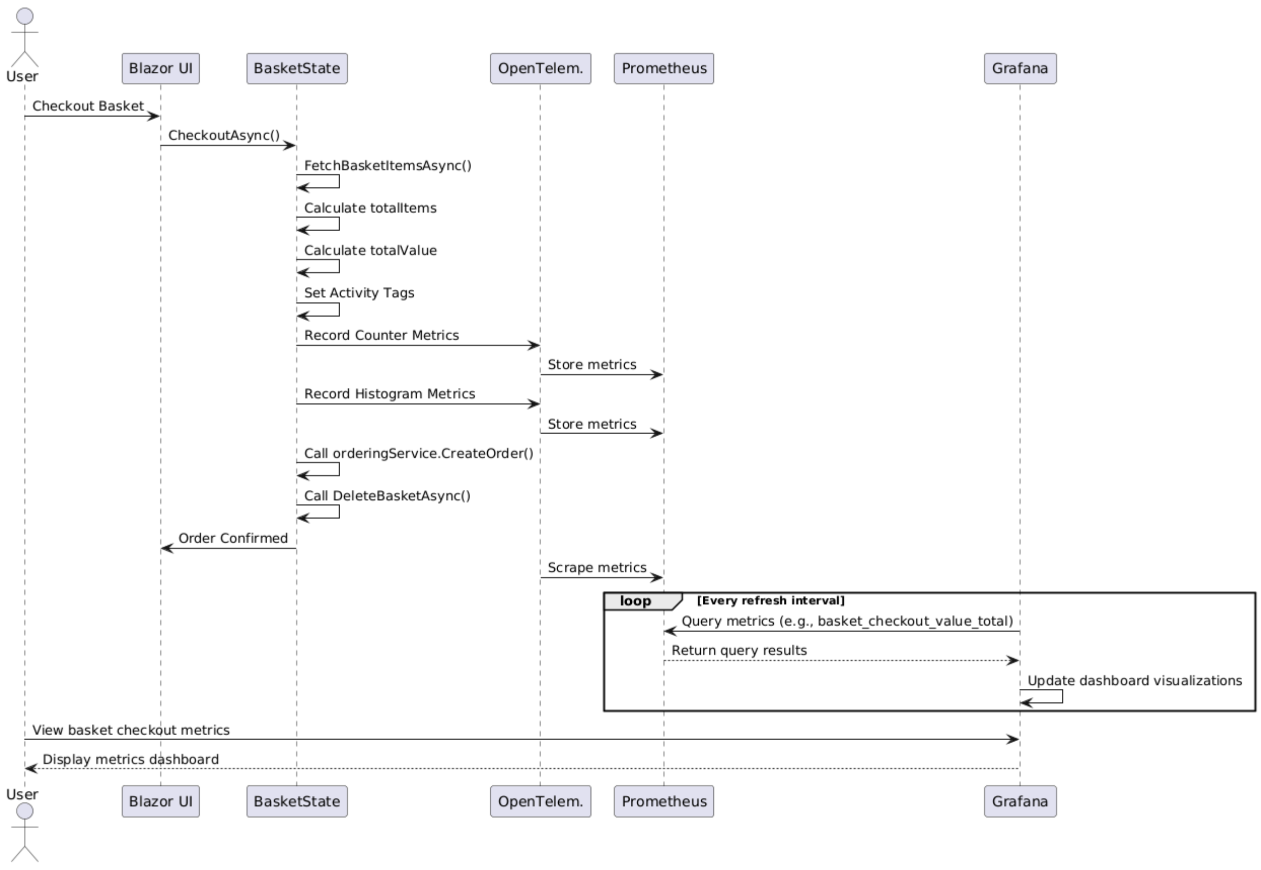
2. System Architecture

2.1 Objectives of the Integration

I incorporated **OpenTelemetry** for tracing, **Prometheus** for metrics collection, and **Jaeger** for distributed tracing visualization. These components work together to provide insights into system performance, request flows, and potential issues.

2.2 Sequence Diagram

The following diagram illustrates the tracing and monitoring workflow for the **basket checkout process**, showing how components interact during a user's request.

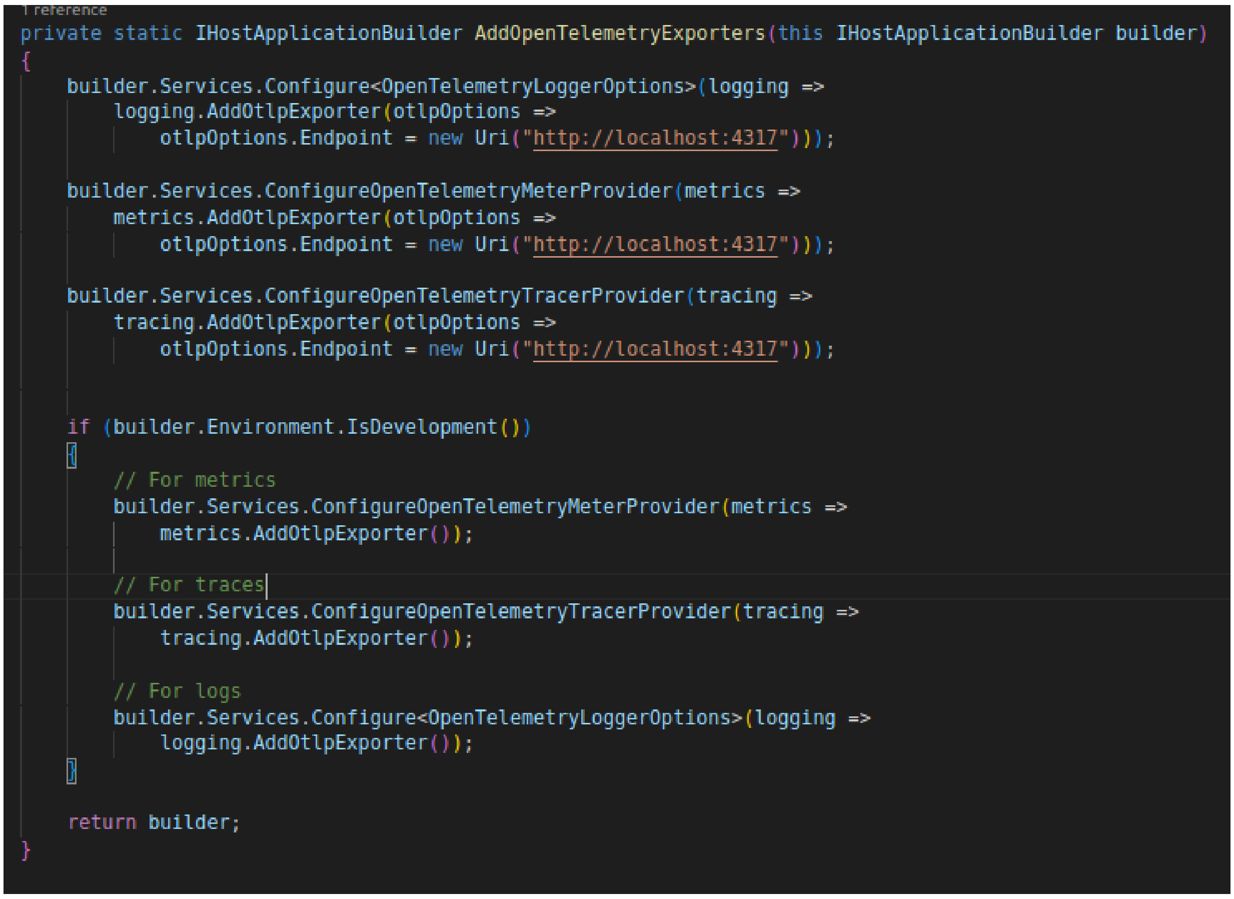


3. Implementation Details

3.1 Traces

3.1.1 Jaeger exporter implementation at the eShop

Jaeger enables detailed trace analysis, providing visibility into how requests propagate through the system, it was implemented at the eShop by the following code, present on https://github.com/Miragaia/PROJECT1_AS/blob/main/src/eShop.ServiceDefaults/Extensions.cs :



```
1 reference
private static IHostApplicationBuilder AddOpenTelemetryExporters(this IHostApplicationBuilder builder)
{
    builder.Services.Configure<OpenTelemetryLoggerOptions>(logging =>
        logging.AddOtlpExporter(otlpOptions =>
            otlpOptions.Endpoint = new Uri("http://localhost:4317")));

    builder.Services.ConfigureOpenTelemetryMeterProvider(metrics =>
        metrics.AddOtlpExporter(otlpOptions =>
            otlpOptions.Endpoint = new Uri("http://localhost:4317")));

    builder.Services.ConfigureOpenTelemetryTracerProvider(tracing =>
        tracing.AddOtlpExporter(otlpOptions =>
            otlpOptions.Endpoint = new Uri("http://localhost:4317")));


    if (builder.Environment.IsDevelopment())
    {
        // For metrics
        builder.Services.ConfigureOpenTelemetryMeterProvider(metrics =>
            metrics.AddOtlpExporter());

        // For traces
        builder.Services.ConfigureOpenTelemetryTracerProvider(tracing =>
            tracing.AddOtlpExporter());

        // For logs
        builder.Services.Configure<OpenTelemetryLoggerOptions>(logging =>
            logging.AddOtlpExporter());
    }

    return builder;
}
```

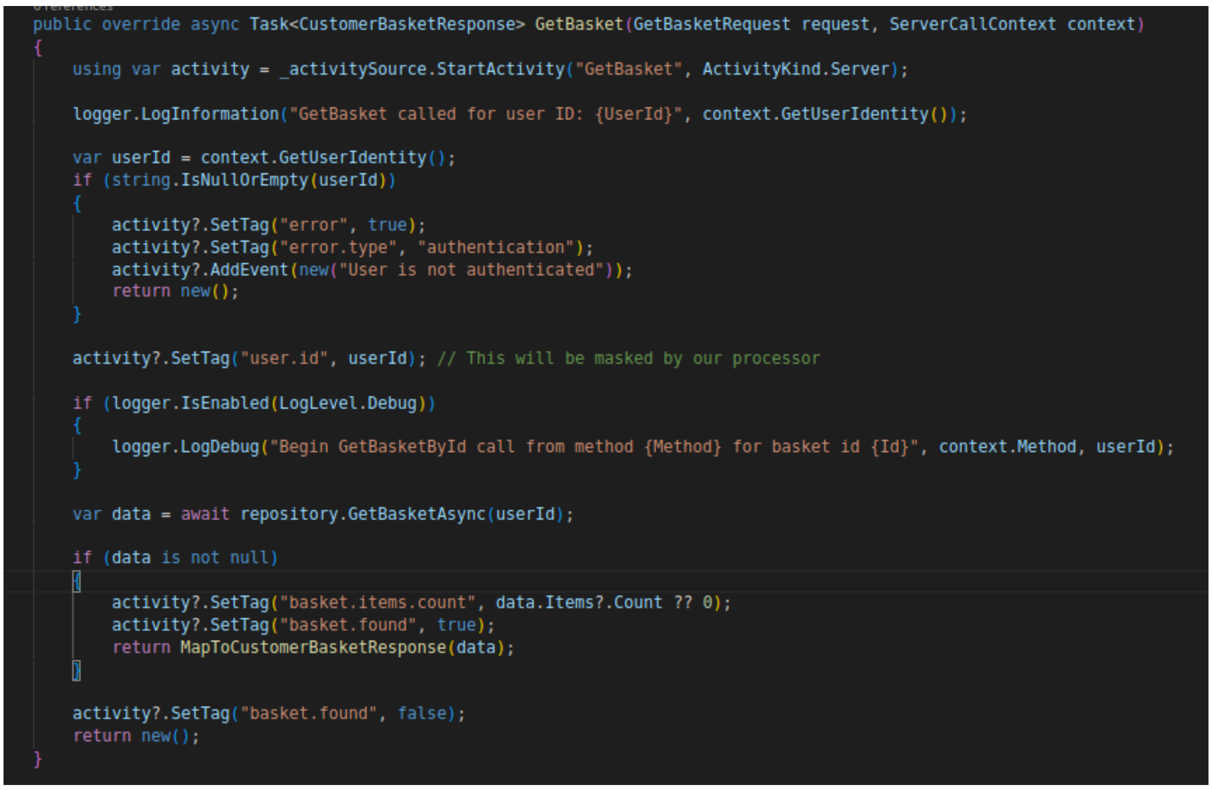
3.1.2 Tracing the Basket/Checkout Process with Jaeger

The Jaeger was configured to collect the traces, spans/activities and tags of all the application's services. The following images showcase the full trace for the Add to Basket/Checkout use case.

To complement this Use Case traceability custom tags were added to the Basket:

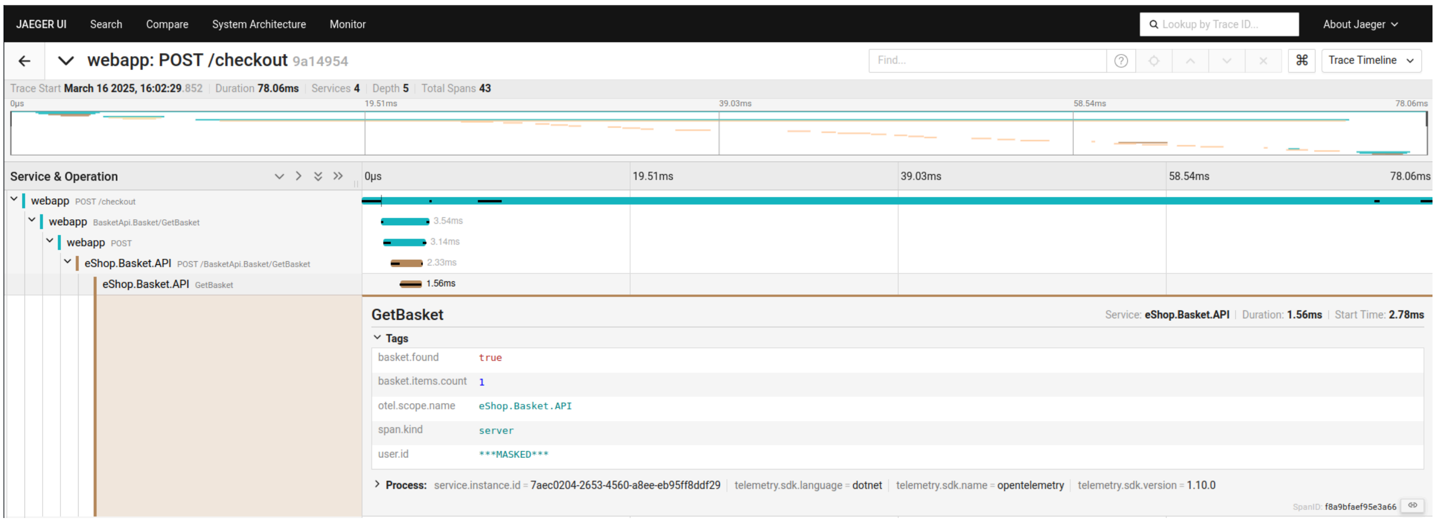
- basket.items.count
- basket.found

Implemented code is available at https://github.com/Miragaia/PROJECT1_AS/blob/main/src/Basket.API/Grpc/BasketService.cs:



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

    logger.LogInformation("GetBasket called for user ID: {UserId}", context.GetUserIdentity());

    var userId = context.GetUserIdentity();
    if (string.IsNullOrEmpty(userId))
    {
        activity?.SetTag("error", true);
        activity?.SetTag("error.type", "authentication");
        activity?.AddEvent(new("User is not authenticated"));
        return new();
    }

    activity?.SetTag("user.id", userId); // This will be masked by our processor

    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?.SetTag("basket.items.count", data.Items?.Count ?? 0);
        activity?.SetTag("basket.found", true);
        return MapToCustomerBasketResponse(data);
    }

    activity?.SetTag("basket.found", false);
    return new();
}
```

The next figure presents a trace where is possible to see the custom tags as well as the defaults of aspire:



webapp: POST /checkout 9a14954

Trace Start March 16 2025, 16:02:29.852 | Duration 78.06ms | Services 4 | Depth 5 | Total Spans 43

```
webapp POST /checkout
  webapp BasketApi.Basket/GetBasket                  3.54ms
    webapp POST                                      3.14ms
      eShop.Basket.API POST /BasketApi.Basket/GetBasket   2.33ms
        eShop.Basket.API GetBasket                   1.56ms
```

GetBasket — Service: eShop.Basket.API | Duration: 1.56ms | Start Time: 2.78ms

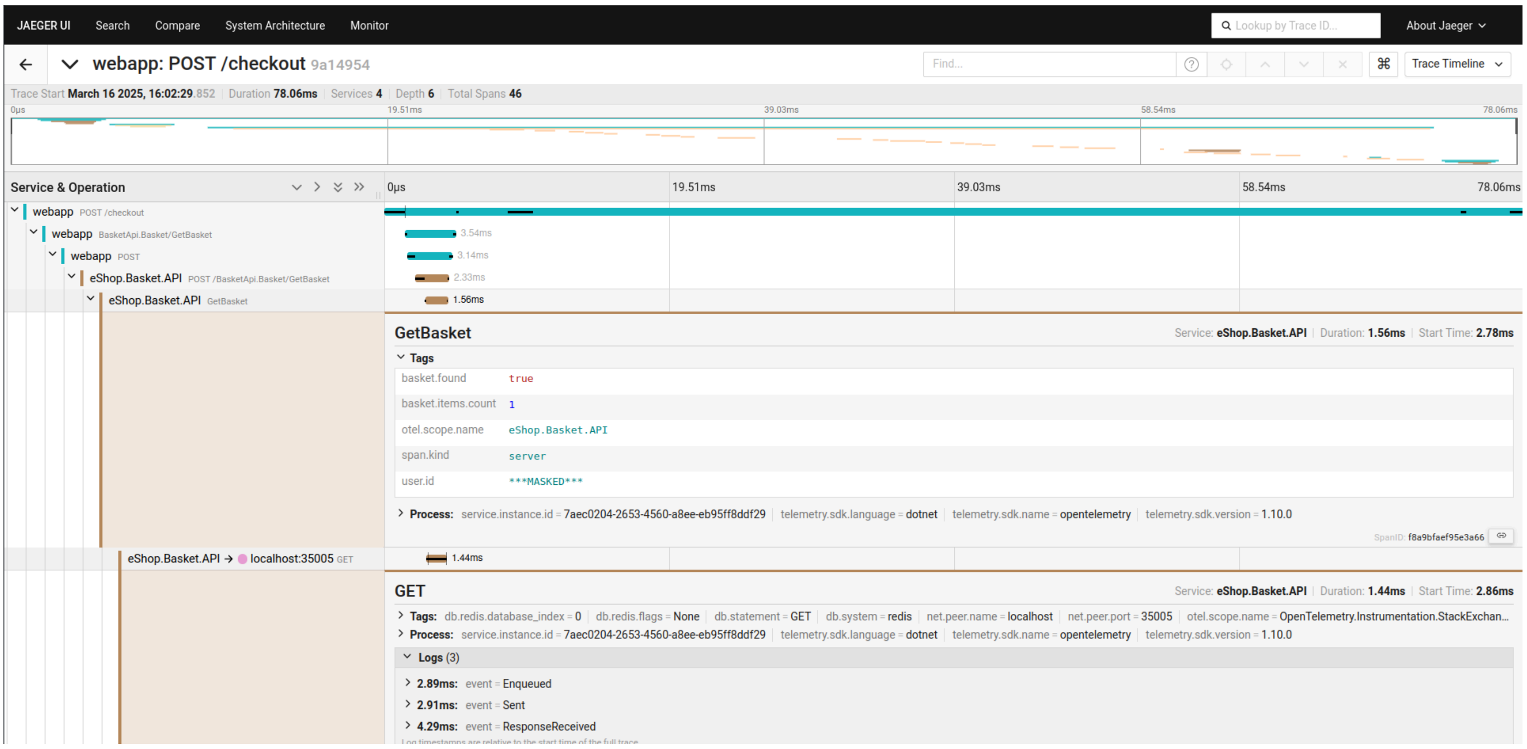
Tags

basket.found	true
basket.items.count	1
otel.scope.name	eShop.Basket.API
span.kind	server
user.id	***MASKED***

Process: service.instance.id = 7aec0204-2653-4560-a8ee-eb95ff8ddf29 | telemetry.sdk.language = dotnet | telemetry.sdk.name = opentelemetry | telemetry.sdk.version = 1.10.0

SpanID: f8a9bfaef95e3a66

It is also possible to confirm that the trace is implemented end-to-end, starting in the webapp, then the service and finally reaching the redis database for the items retrieval:



webapp: POST /checkout 9a14954

Trace Start **March 16 2025, 16:02:29**.852 | Duration **78.06ms** | Services **4** | Depth **6** | Total Spans **46**

```
Service & Operation                                   0μs      19.51ms      39.03ms      58.54ms      78.06ms
webapp POST /checkout
  webapp BasketApi.Basket/GetBasket                   3.54ms
    webapp POST                                       3.14ms
      eShop.Basket.API POST /BasketApi.Basket/GetBasket   2.33ms
        eShop.Basket.API GetBasket                    1.56ms
          eShop.Basket.API → localhost:35005 GET      1.44ms
```

GetBasket — Service: **eShop.Basket.API** | Duration: **1.56ms** | Start Time: **2.78ms**

Tags

basket.found	true
basket.items.count	1
otel.scope.name	eShop.Basket.API
span.kind	server
user.id	***MASKED***

Process: service.instance.id = 7aec0204-2653-4560-a8ee-eb95ff8ddf29 | telemetry.sdk.language = dotnet | telemetry.sdk.name = opentelemetry | telemetry.sdk.version = 1.10.0

SpanID: f8a9bfaef95e3a66

GET — Service: **eShop.Basket.API** | Duration: **1.44ms** | Start Time: **2.86ms**

Tags: db.redis.database_index = 0 | db.redis.flags = None | db.statement = GET | db.system = redis | net.peer.name = localhost | net.peer.port = 35005 | otel.scope.name = OpenTelemetry.Instrumentation.StackExchan...

Process: service.instance.id = 7aec0204-2653-4560-a8ee-eb95ff8ddf29 | telemetry.sdk.language = dotnet | telemetry.sdk.name = opentelemetry | telemetry.sdk.version = 1.10.0

Logs (3)

```
2.89ms: event = Enqueued
2.91ms: event = Sent
4.29ms: event = ResponseReceived
```

Log timestamps are relative to the start time of the full trace

3.2 Metrics

Metrics are more than essential for the insights of the system, performance, resources and application insights.

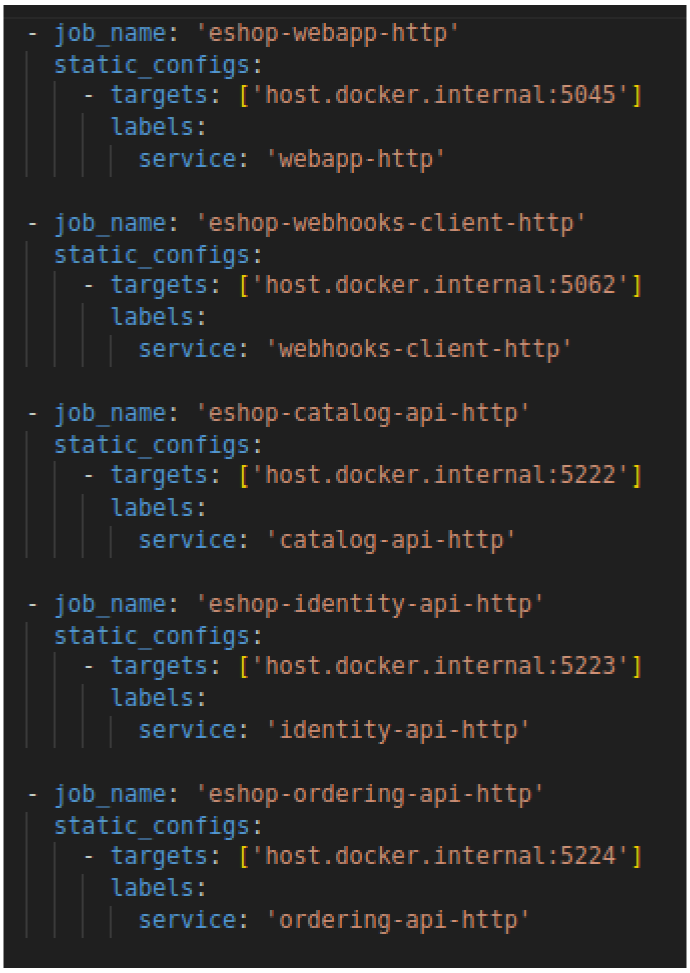
3.2.1 Prometheus Configuration

I have used Prometheus to collect and store these metrics in order to later export them to a Grafana dashboard to visualize them.

For that I added this to lines of code to the previously presented **Extensions.cs:**

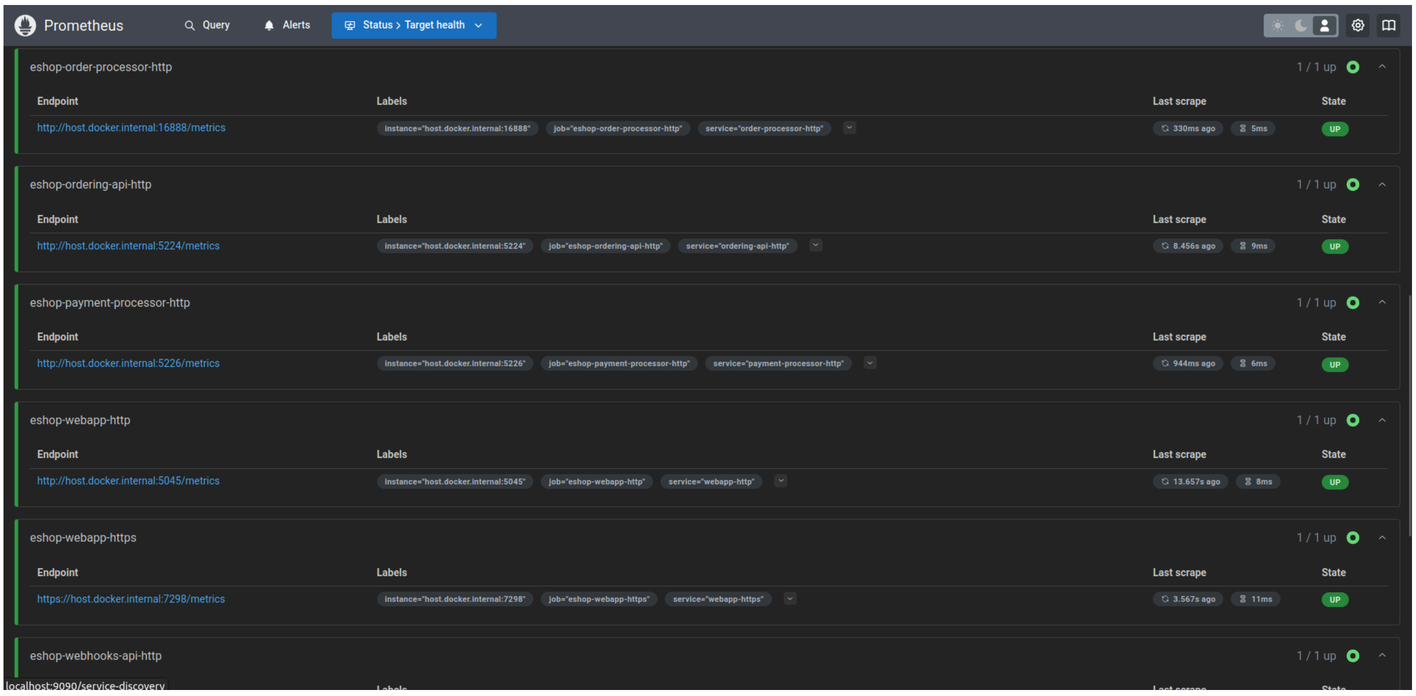
- builder.Services.ConfigureOpenTelemetryMeterProvider(metrics => metrics.AddPrometheusExporter());
- `app.MapPrometheusScrapingEndpoint()`

It was also needed to configure prometheus targets and map it to its docker container to gather the metrics. This is present at https://github.com/Miragaia/PROJECT1_AS/blob/main/prometheus/prometheus.yml



```yaml
- job_name: 'eshop-webapp-http'
  static_configs:
    - targets: ['host.docker.internal:5045']
      labels:
        service: 'webapp-http'

- job_name: 'eshop-webhooks-client-http'
  static_configs:
    - targets: ['host.docker.internal:5062']
      labels:
        service: 'webhooks-client-http'

- job_name: 'eshop-catalog-api-http'
  static_configs:
    - targets: ['host.docker.internal:5222']
      labels:
        service: 'catalog-api-http'

- job_name: 'eshop-identity-api-http'
  static_configs:
    - targets: ['host.docker.internal:5223']
      labels:
        service: 'identity-api-http'

- job_name: 'eshop-ordering-api-http'
  static_configs:
    - targets: ['host.docker.internal:5224']
      labels:
        service: 'ordering-api-http'
```

The next figure showcases the health status of the configured targets at the prometheus interface, concluding that prometheus is able to scrape the necessary metrics:



eshop-order-processor-http — 1 / 1 up

Endpoint	Labels	Last scrape	State
http://host.docker.internal:16888/metrics	instance="host.docker.internal:16888" job="eshop-order-processor-http" service="order-processor-http"	330ms ago 5ms	UP

eshop-ordering-api-http — 1 / 1 up

Endpoint	Labels	Last scrape	State
http://host.docker.internal:5224/metrics	instance="host.docker.internal:5224" job="eshop-ordering-api-http" service="ordering-api-http"	8.456s ago 9ms	UP

eshop-payment-processor-http — 1 / 1 up

Endpoint	Labels	Last scrape	State
http://host.docker.internal:5226/metrics	instance="host.docker.internal:5226" job="eshop-payment-processor-http" service="payment-processor-http"	944ms ago 6ms	UP

eshop-webapp-http — 1 / 1 up

Endpoint	Labels	Last scrape	State
http://host.docker.internal:5045/metrics	instance="host.docker.internal:5045" job="eshop-webapp-http" service="webapp-http"	13.657s ago 8ms	UP

eshop-webapp-https — 1 / 1 up

Endpoint	Labels	Last scrape	State
https://host.docker.internal:7298/metrics	instance="host.docker.internal:7298" job="eshop-webapp-https" service="webapp-https"	3.567s ago 11ms	UP

eshop-webhooks-api-http — 1 / 1 up

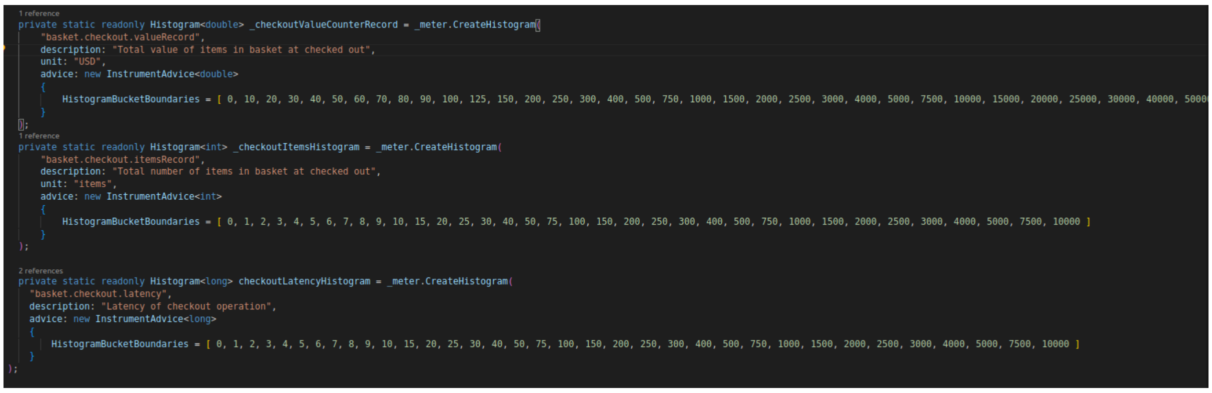
localhost:9090/service-discovery

3.2.2 Collected Metrics

At the code below are present some of the custom metrics:

- **basket.checkout.valueRecord:** Total value of items in basket at checked out.
- **basket.checkout.itemsRecord:** Total number of items in basket at checked out.
- **basket.checkout.latency:** Latency of checkout operation.

The full usage (consumptions/triggering and operations) is presented at https://github.com/Miragaia/PROJECT1_AS/blob/main/src/WebApp/Services/BasketState.cs



```
1 reference
private static readonly Histogram<double> _checkoutValueCounterRecord = _meter.CreateHistogram(
    "basket.checkout.valueRecord",
    description: "Total value of items in basket at checked out",
    unit: "USD",
    advice: new InstrumentAdvice<double>
    {
        HistogramBucketBoundaries = [ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 125, 150, 200, 250, 300, 400, 500, 750, 1000, 1500, 2000, 2500, 3000, 4000, 5000, 7500, 10000, 15000, 20000, 25000, 30000, 40000, 5000
    }
);
1 reference
private static readonly Histogram<int> _checkoutItemsHistogram = _meter.CreateHistogram(
    "basket.checkout.itemsRecord",
    description: "Total number of items in basket at checked out",
    unit: "items",
    advice: new InstrumentAdvice<int>
    {
        HistogramBucketBoundaries = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 40, 50, 75, 100, 150, 200, 250, 300, 400, 500, 750, 1000, 1500, 2000, 2500, 3000, 4000, 5000, 7500, 10000 ]
    }
);

2 references
private static readonly Histogram<long> checkoutLatencyHistogram = _meter.CreateHistogram(
  "basket.checkout.latency",
  description: "Latency of checkout operation",
  advice: new InstrumentAdvice<long>
  {
      HistogramBucketBoundaries = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 40, 50, 75, 100, 150, 200, 250, 300, 400, 500, 750, 1000, 1500, 2000, 2500, 3000, 4000, 5000, 7500, 10000 ]
  }
);
```

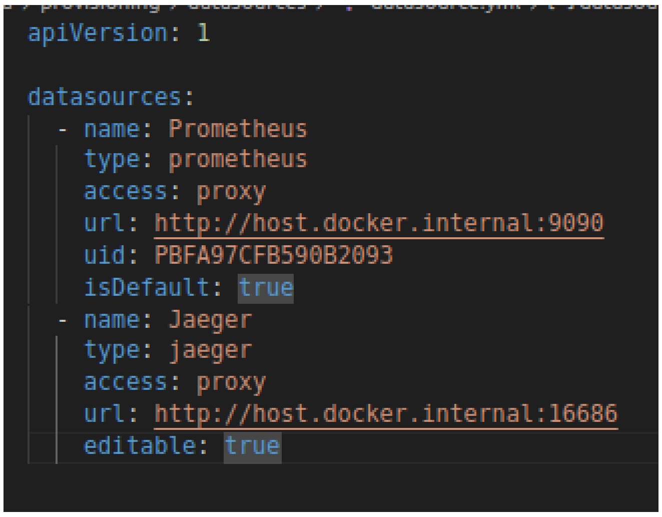
All these metrics and more are consumed by Prometheus and displayed at a Grafana dashboard.

3.2.3 Grafana Dashboard

Grafana queries **Prometheus** at regular intervals and updates the dashboards for real-time visualization of application performance and custom **basket/checkout metrics**.

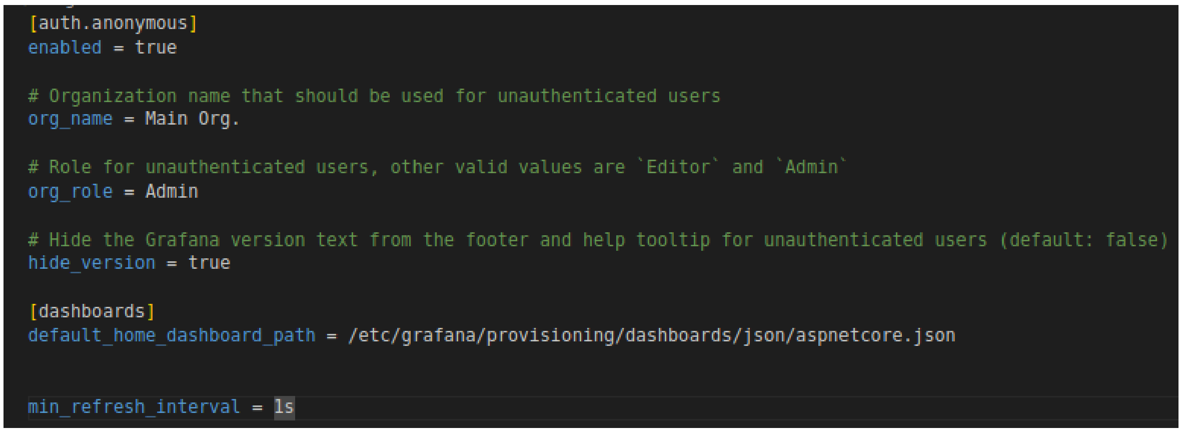
To enable seamless integration with Prometheus and Jaeger, Grafana is configured using **provisioning files** that automatically set up data sources and dashboards upon startup.

This file (datasoruce.yml) defines Prometheus and Jaeger as **data sources**:



```yaml
apiVersion: 1

datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://host.docker.internal:9090
    uid: PBFA97CFB590B2093
    isDefault: true
  - name: Jaeger
    type: jaeger
    access: proxy
    url: http://host.docker.internal:16686
    editable: true
```

The following settings (grafana.ini) ensure that Grafana is publicly accessible without authentication and loads the selected dashboard by default:



```
[auth.anonymous]
enabled = true

# Organization name that should be used for unauthenticated users
org_name = Main Org.

# Role for unauthenticated users, other valid values are `Editor` and `Admin`
org_role = Admin

# Hide the Grafana version text from the footer and help tooltip for unauthenticated users (default: false)
hide_version = true

[dashboards]
default_home_dashboard_path = /etc/grafana/provisioning/dashboards/json/aspnetcore.json


min_refresh_interval = 1s
```

The Grafana dashboard is shown in a later section of the report.

3.3 Security Considerations

Sensitive user information is **masked** from logs and traces before being stored and presented. To accomplish this I have created a **SensitiveDataProcessor.cs** available at https://github.com/Miragaia/PROJECT1_AS/blob/main/src/eShop.ServiceDefaults/Telemetry/SensitiveDataProcessor.cs

This processor is registered in **`Extensions.cs`** using:

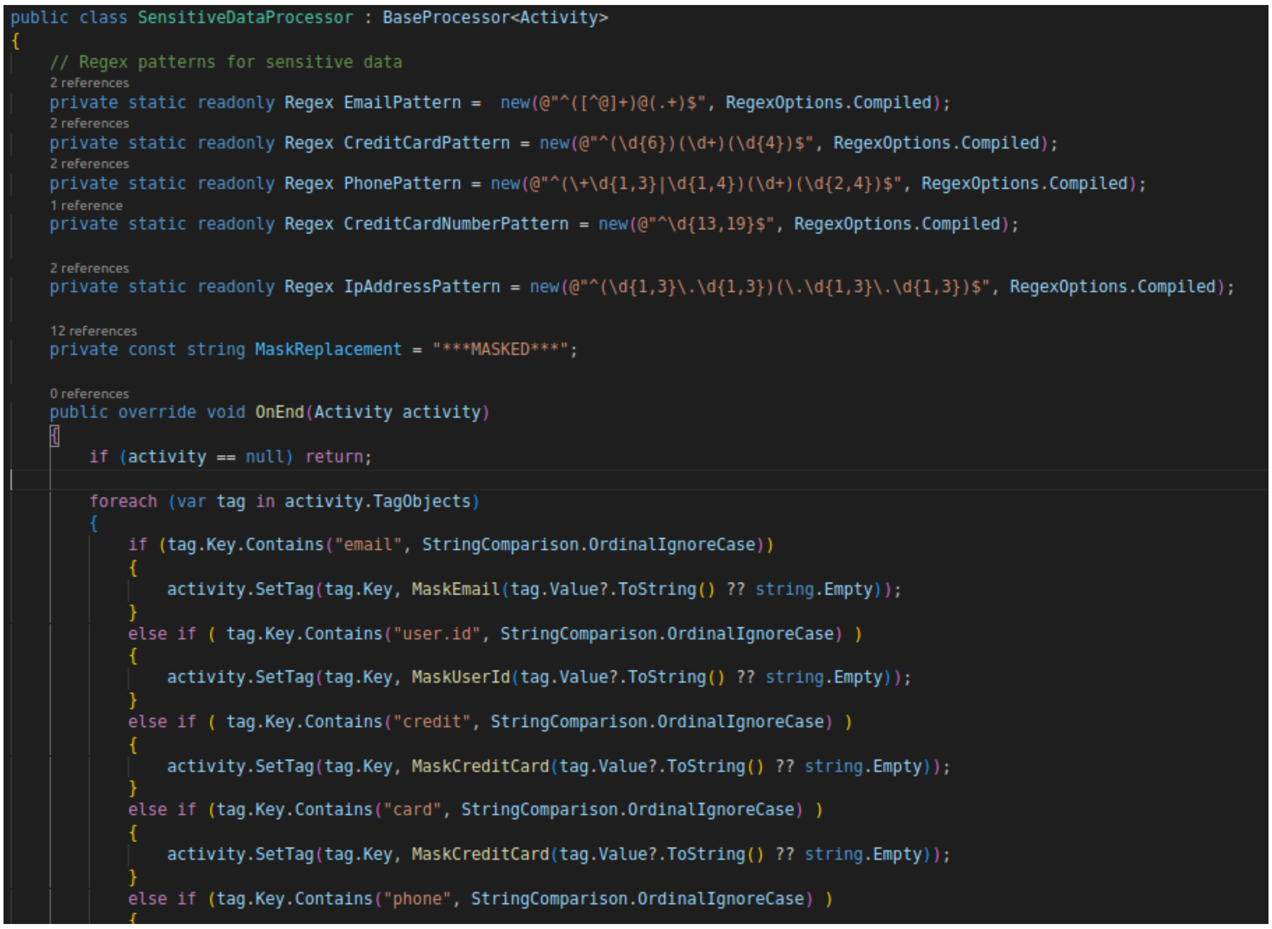
- `.AddProcessor<SensitiveDataProcessor>()`

This ensures that any telemetry data processed by OpenTelemetry is scrubbed of sensitive details before being exported.

The **`SensitiveDataProcessor`** extends OpenTelemetry's `BaseProcessor<Activity>` and applies regex to detect and mask sensitive data before it is recorded in traces and logs. The processor performs the following tasks:

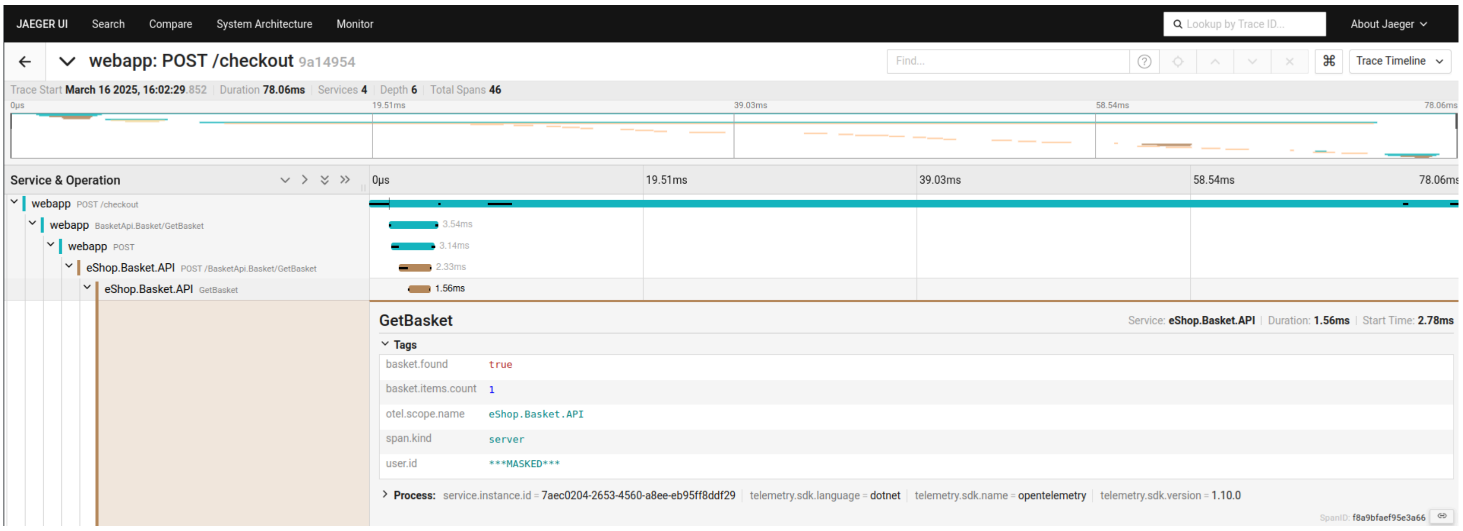
- **Identifies sensitive fields** (e.g., `email`, `user.id`, `credit_card`, `phone`, `password`, `token`).
- **Uses regex patterns** to match sensitive data such as:
  - Email addresses (`example@domain.com` → `***MASKED***@domain.com`)
  - Credit card numbers (`1234567812345678` → `123456***MASKED***5678`)
  - Phone numbers (`+1234567890` → `+1***MASKED***90`)
  - IP addresses (`192.168.1.1` → `192.168***MASKED***`)
- **Overrides logs and activity tags** before exporting them to telemetry storage (e.g., Jaeger, Prometheus).

This ensures that sensitive information never reaches the logs or monitoring tools, enhancing data privacy and compliance with security best practices.



```
public class SensitiveDataProcessor : BaseProcessor<Activity>
{
    // Regex patterns for sensitive data
    2 references
    private static readonly Regex EmailPattern =  new(@"^([^@]+)@(.+)$", RegexOptions.Compiled);
    2 references
    private static readonly Regex CreditCardPattern = new(@"^(\d{6})(\d+)(\d{4})$", RegexOptions.Compiled);
    2 references
    private static readonly Regex PhonePattern = new(@"^(\+\d{1,3}|\d{1,4})(\d+)(\d{2,4})$", RegexOptions.Compiled);
    1 reference
    private static readonly Regex CreditCardNumberPattern = new(@"^\d{13,19}$", RegexOptions.Compiled);

    2 references
    private static readonly Regex IpAddressPattern = new(@"^(\d{1,3}\.\d{1,3})(\.\d{1,3}\.\d{1,3})$", RegexOptions.Compiled);

    12 references
    private const string MaskReplacement = "***MASKED***";

    0 references
    public override void OnEnd(Activity activity)
    {
        if (activity == null) return;

        foreach (var tag in activity.TagObjects)
        {
            if (tag.Key.Contains("email", StringComparison.OrdinalIgnoreCase))
            {
                activity.SetTag(tag.Key, MaskEmail(tag.Value?.ToString() ?? string.Empty));
            }
            else if ( tag.Key.Contains("user.id", StringComparison.OrdinalIgnoreCase) )
            {
                activity.SetTag(tag.Key, MaskUserId(tag.Value?.ToString() ?? string.Empty));
            }
            else if ( tag.Key.Contains("credit", StringComparison.OrdinalIgnoreCase) )
            {
                activity.SetTag(tag.Key, MaskCreditCard(tag.Value?.ToString() ?? string.Empty));
            }
            else if (tag.Key.Contains("card", StringComparison.OrdinalIgnoreCase) )
            {
                activity.SetTag(tag.Key, MaskCreditCard(tag.Value?.ToString() ?? string.Empty));
            }
            else if (tag.Key.Contains("phone", StringComparison.OrdinalIgnoreCase) )
            {
```

The following trace from Jaeger demonstrates the masking of sensitive user information. As shown, the `user.id` field has been successfully masked, ensuring that personally identifiable information (PII) is not exposed in telemetry data:



webapp: POST /checkout 9a14954

Trace Start **March 16 2025, 16:02:29**.852 | Duration **78.06ms** | Services **4** | Depth **6** | Total Spans **46**

Service & Operation

```
webapp POST /checkout
  webapp BasketApi.Basket/GetBasket                    3.54ms
    webapp POST                                        3.14ms
      eShop.Basket.API POST /BasketApi.Basket/GetBasket  2.33ms
        eShop.Basket.API GetBasket                     1.56ms
```

GetBasket Service: **eShop.Basket.API** | Duration: **1.56ms** | Start Time: **2.78ms**

Tags

basket.found	true
basket.items.count	1
otel.scope.name	eShop.Basket.API
span.kind	server
user.id	***MASKED***

Process: service.instance.id = 7aec0204-2653-4560-a8ee-eb95ff8ddf29 | telemetry.sdk.language = dotnet | telemetry.sdk.name = opentelemetry | telemetry.sdk.version = 1.10.0

SpanID: f8a9bfaef95e3a66

4. Deployment & Setup

4.1 Running the System

This command executes the docker compose with the observability stack (Jaeger, Prometheus and Grafana):

- docker compose -f docker-compose.observability.yml up -d

On another terminal execute this command to run the application:

- dotnet run --project src/eShop.AppHost

4.2 Accessing Dashboards

Once the system is running, access the monitoring tools via:

- **Grafana** → http://localhost:3000
- **Jaeger** → http://localhost:16686
- **Prometheus** → http://localhost:9090

5. Observability Results

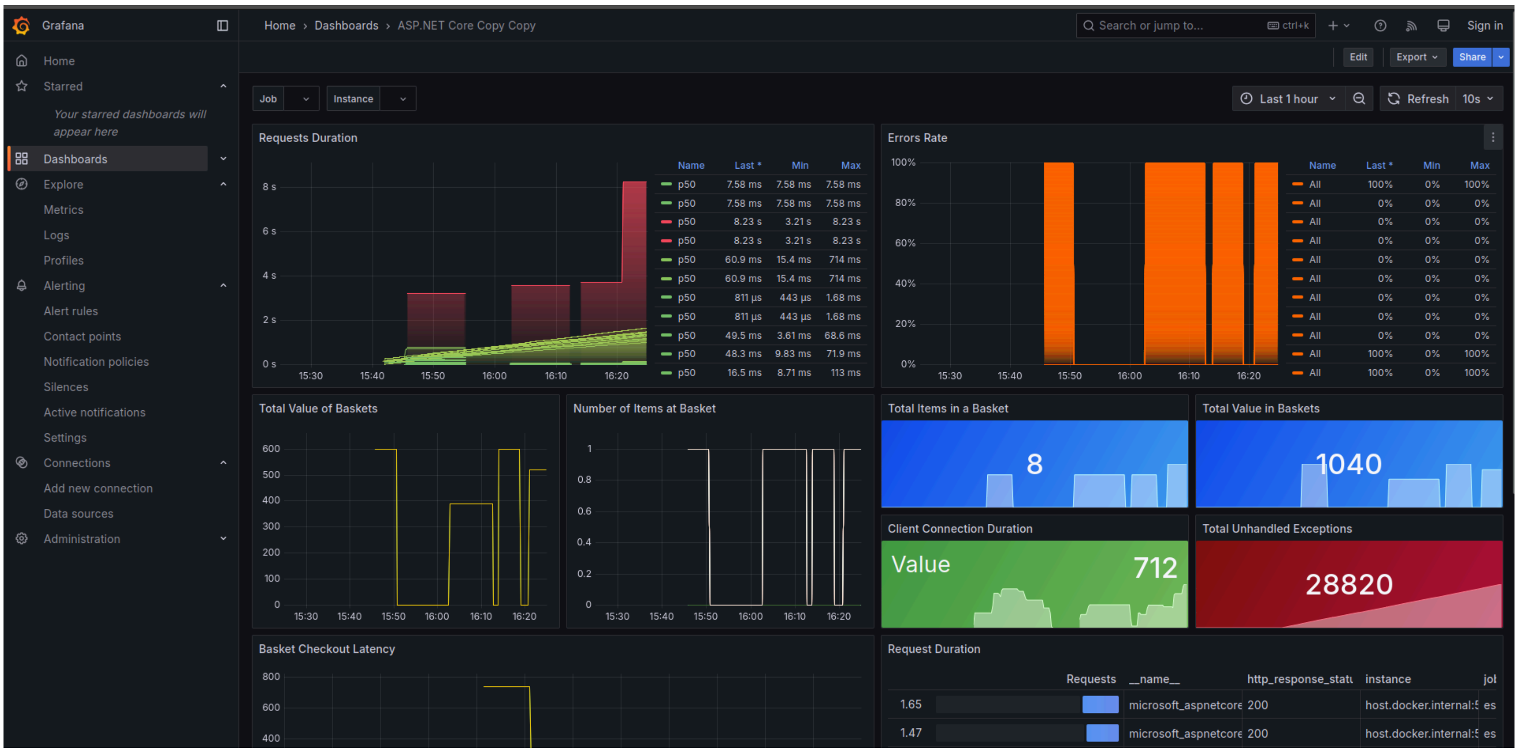
5.1 Grafana Dashboard

Grafana provides real-time visualization of system performance and custom business metrics. With the integration of Prometheus, key metrics such as request rates, response times, and error rates are continuously monitored.

Additionally, by adding custom **basket/checkout metrics**, I can observe specific events related to user purchases, such as:

- The number of items in a checkout request.
- The total value of transactions.
- Checkout frequency over time.
- Checkout latency

This dashboard contains both the system performance and custom basket/checkout metrics.



Requests Duration

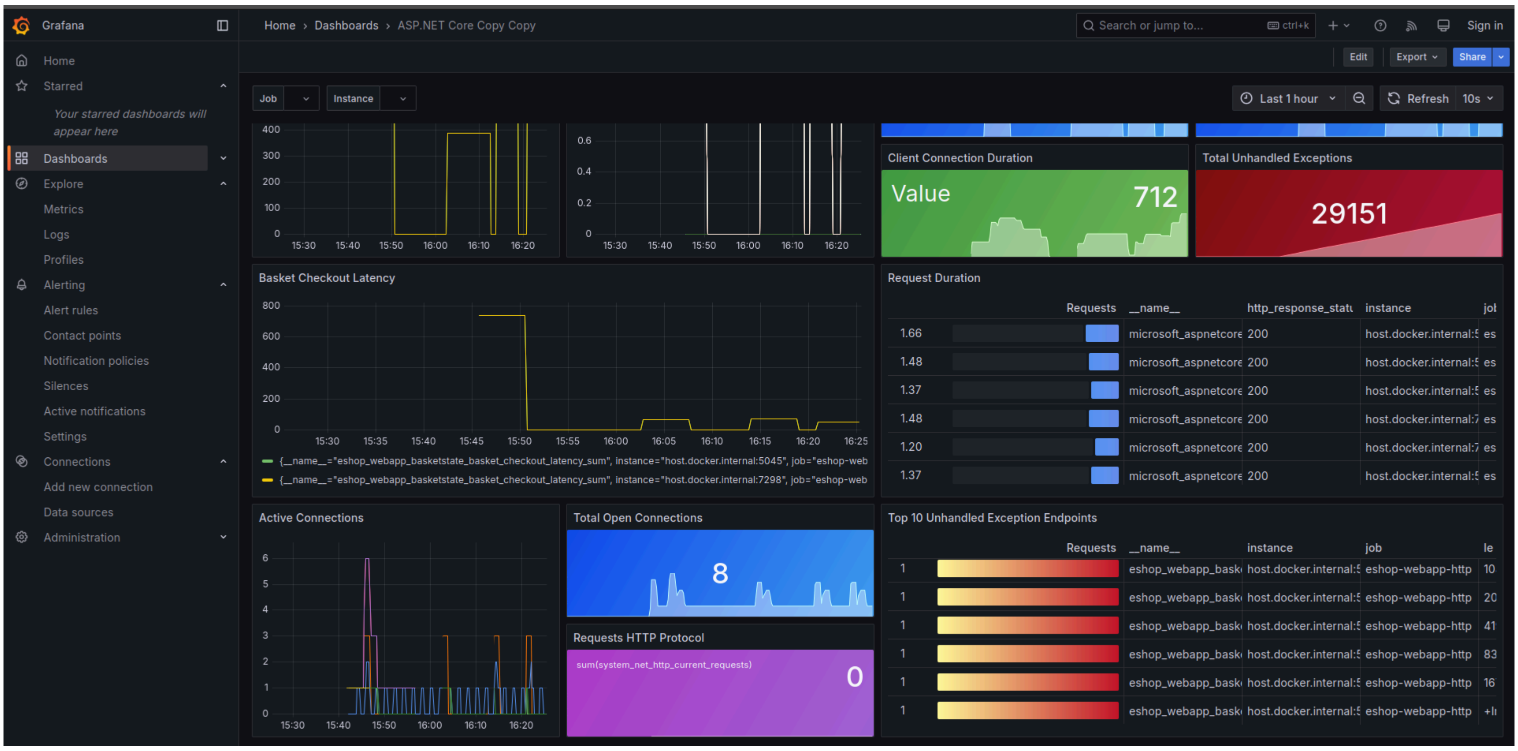
	Name	Last *	Min	Max
🟩	p50	7.58 ms	7.58 ms	7.58 ms
🟩	p50	7.58 ms	7.58 ms	7.58 ms
🟥	p50	8.23 s	3.21 s	8.23 s
🟥	p50	8.23 s	3.21 s	8.23 s
🟩	p50	60.9 ms	15.4 ms	714 ms
🟩	p50	60.9 ms	15.4 ms	714 ms
🟩	p50	811 µs	443 µs	1.68 ms
🟩	p50	811 µs	443 µs	1.68 ms
🟩	p50	49.5 ms	3.61 ms	68.6 ms
🟩	p50	48.3 ms	9.83 ms	71.9 ms
🟩	p50	16.5 ms	8.71 ms	113 ms

Errors Rate

	Name	Last *	Min	Max
🟧	All	100%	0%	100%
🟧	All	0%	0%	0%
🟧	All	0%	0%	0%
🟧	All	0%	0%	0%
🟧	All	0%	0%	0%
🟧	All	0%	0%	0%
🟧	All	0%	0%	0%
🟧	All	0%	0%	0%
🟧	All	0%	0%	0%
🟧	All	100%	0%	100%
🟧	All	100%	0%	100%

Request Duration

Requests	__name__	http_response_statu	instance	jol
1.65	microsoft_aspnetcore	200	host.docker.internal:5	es
1.47	microsoft_aspnetcore	200	host.docker.internal:5	es



Basket Checkout Latency legend:

🟩 {__name__="eshop_webapp_basketstate_basket_checkout_latency_sum", instance="host.docker.internal:5045", job="eshop-web

🟨 {__name__="eshop_webapp_basketstate_basket_checkout_latency_sum", instance="host.docker.internal:7298", job="eshop-web

Request Duration

Requests	__name__	http_response_statu	instance	jol
1.66	microsoft_aspnetcore	200	host.docker.internal:5	es
1.48	microsoft_aspnetcore	200	host.docker.internal:5	es
1.37	microsoft_aspnetcore	200	host.docker.internal:5	es
1.48	microsoft_aspnetcore	200	host.docker.internal:7	es
1.20	microsoft_aspnetcore	200	host.docker.internal:7	es
1.37	microsoft_aspnetcore	200	host.docker.internal:5	es

Top 10 Unhandled Exception Endpoints

Requests	__name__	instance	job	le
1	eshop_webapp_bask	host.docker.internal:5	eshop-webapp-http	10
1	eshop_webapp_bask	host.docker.internal:5	eshop-webapp-http	20
1	eshop_webapp_bask	host.docker.internal:5	eshop-webapp-http	41
1	eshop_webapp_bask	host.docker.internal:5	eshop-webapp-http	83
1	eshop_webapp_bask	host.docker.internal:5	eshop-webapp-http	16
1	eshop_webapp_bask	host.docker.internal:5	eshop-webapp-http	+Il

6. Load Testing

Initially, the load test was designed to evaluate the **basket/checkout flow**, but authentication challenges prevented successful execution. Despite attempting to bypass authentication using both **Locust** and **K6**, I was unable to proceed. Consequently, I shifted the focus of the load test to the **catalog item selection**, which does not require authentication.

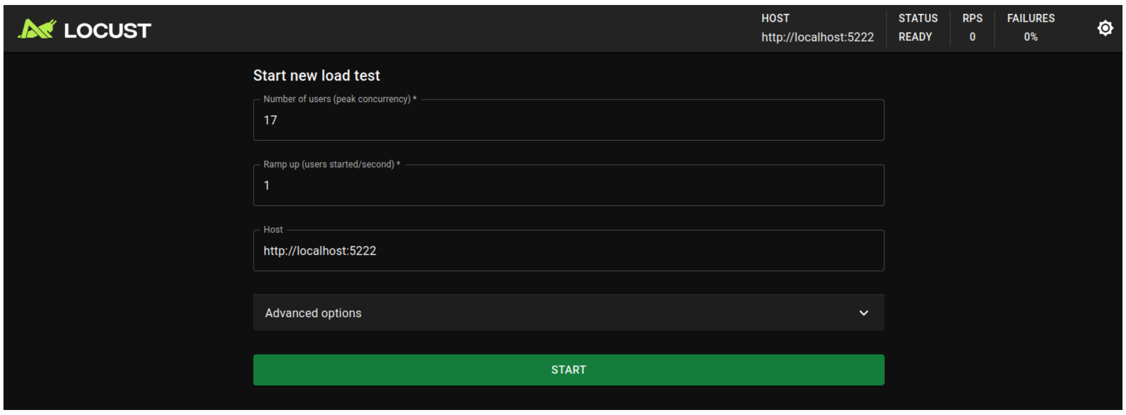
6.1 Load Test Setup

The load test was conducted using **Locust**. The test simulated users requesting **random catalog items** by querying the API endpoint.

To run the tests: **locust -f load-test.py**

The **Locust script**:

- Randomly selects an **item ID** between `1-100`.
- Sends a GET request to the catalog API.
- Measures **response times**, **request rate**, and **failure rates**.



Start new load test

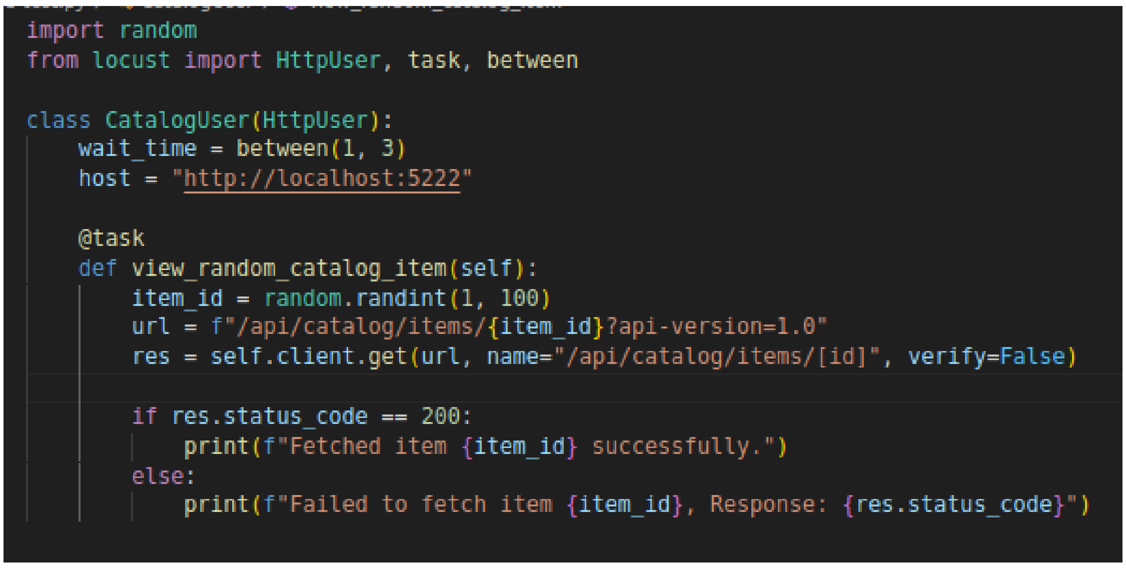
Number of users (peak concurrency) * 17

Ramp up (users started/second) * 1

Host http://localhost:5222

The test configuration in **Locust's web interface** involved:

1. **Defining the number of users**: The test was initialized with **17 users**.
2. **Setting the spawn rate**: Users were spawned at **1 user per second**.
3. **Starting the test**: After launching, results were monitored in real-time via the **Locust dashboard**.



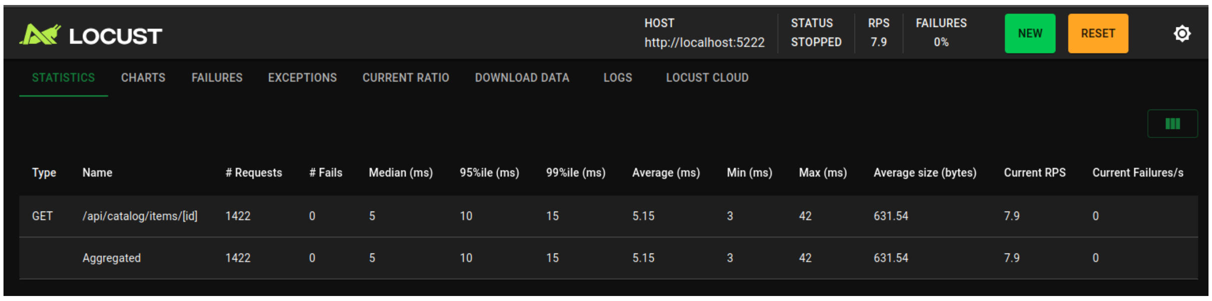
```python
import random
from locust import HttpUser, task, between

class CatalogUser(HttpUser):
    wait_time = between(1, 3)
    host = "http://localhost:5222"

    @task
    def view_random_catalog_item(self):
        item_id = random.randint(1, 100)
        url = f"/api/catalog/items/{item_id}?api-version=1.0"
        res = self.client.get(url, name="/api/catalog/items/[id]", verify=False)

        if res.status_code == 200:
            print(f"Fetched item {item_id} successfully.")
        else:
            print(f"Failed to fetch item {item_id}, Response: {res.status_code}")
```

6.2 Load Test Resutls

The load test ran **1422 requests** over a sustained period, generating the following key performance metrics:

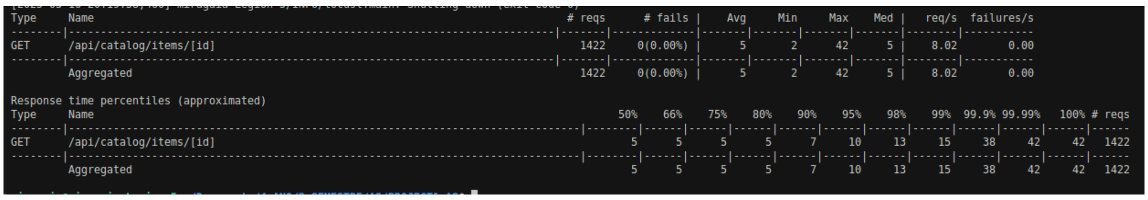
- **Total Requests: 1422**
- **Failures: 0 (0%)**
- **Average Response Time: 5.15 ms**
- **Median Response Time: 5 ms**
- **95th Percentile Response Time: 10 ms**
- **99th Percentile Response Time: 15 ms**
- **Requests per Second (RPS): 7.9 RPS**

Below is a **screenshot of the Locust UI** displaying the test results:



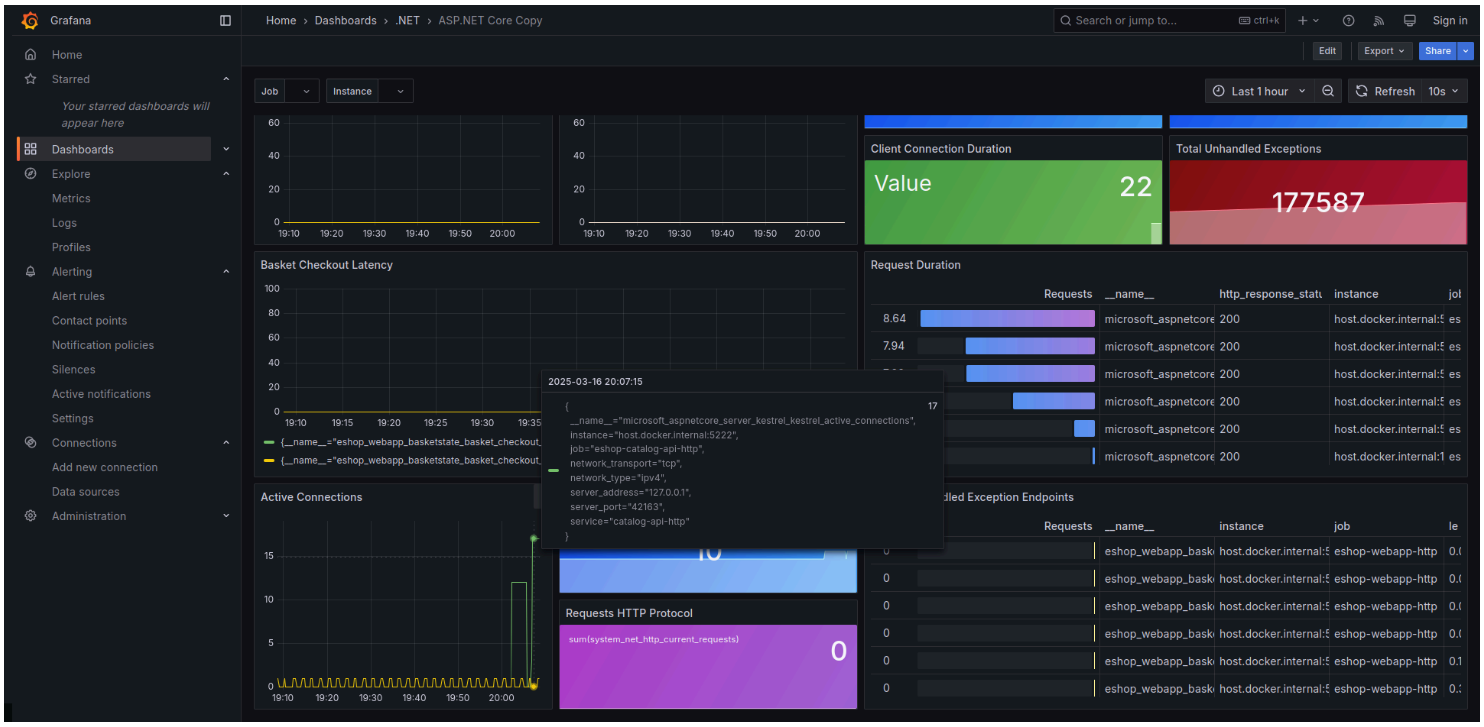
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/api/catalog/items/[id]	1422	0	5	10	15	5.15	3	42	631.54	7.9	0
	Aggregated	1422	0	5	10	15	5.15	3	42	631.54	7.9	0

Additionally, terminal output confirmed the response time distribution:



```
Type     Name                                                                          # reqs      # fails |    Avg     Min     Max    Med |   req/s  failures/s
--------|----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|-------|--------|-----------
GET      /api/catalog/items/[id]                                                         1422     0(0.00%) |      5       2      42      5 |    8.02        0.00
--------|----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|-------|--------|-----------
         Aggregated                                                                      1422     0(0.00%) |      5       2      42      5 |    8.02        0.00

Response time percentiles (approximated)
Type     Name                                                                                  50%    66%    75%    80%    90%    95%    98%    99%  99.9% 99.99%   100% # reqs
--------|--------------------------------------------------------------------------------|--------|------|------|------|------|------|------|------|------|------|------|------
GET      /api/catalog/items/[id]                                                                 5      5      5      5      7     10     13     15     38     42     42   1422
--------|--------------------------------------------------------------------------------|--------|------|------|------|------|------|------|------|------|------|------|------
         Aggregated                                                                              5      5      5      5      7     10     13     15     38     42     42   1422
```

The 17 users initialized are corroborated by this grafana graphic that I have at my dashboard registering the number of active connections:



```
2025-03-16 20:07:15
{                                                                              17
  __name__="microsoft_aspnetcore_server_kestrel_kestrel_active_connections",
  instance="host.docker.internal:5222",
  job="eshop-catalog-api-http",
  network_transport="tcp",
  network_type="ipv4",
  server_address="127.0.0.1",
  server_port="42163",
  service="catalog-api-http"
}
```

7. Conclusion

In my opinion, I have successfully completed all the objectives of this assignment, with the exception of the optional one. Despite this, I found the assignment to be a great introduction to the observability tools.

One of the initial challenges I faced was working with an unfamiliar codebase and the fact that I had never used .NET before. This required some additional effort to understand the structure and behavior of the application. The only major difficulty I encountered was related to authentication for the load tests. I attempted to test the basket/checkout flow using both Locust and K6, but I was unable to bypass authentication successfully. As a result, I adapted the load test to focus on catalog item selection, which did not require authentication.