Integrating OpenTelemetry & Security in *eShop*

Maria João Machado Sardinha
(108756)

Project 1

Software Architectures Course

University of Aveiro

2024/2025

GitHub: https://github.com/mariiajoao/eShop

Contents

Abstract

This report details the integration of OpenTelemetry [1] and security measures into the eShop's [2] microservices architecture. Conducted as part of the Software Architectures course [3], this project [4] implements comprehensive tracing for the "Place an Order" workflow while ensuring data privacy through a custom *PII scrubbing* mechanism.

The observability stack, comprising *Jaeger* [5], *Prometheus* [6], and *Grafana* [7], enables real-time system monitoring and performance analysis. Additionally, load testing was conducted using *K6* [8].

I Introduction

I-A Project Background

eShop [2] is a modern e-commerce platform (1) built with `ASP.NET Core` and follows a microservices architecture. The system is containerized and employs event-driven design patterns, enhancing scalability, but introducing complexities in **monitoring**, **debugging**, and **security**.

As distributed systems grow in complexity, **observability** becomes critical for understanding the system behavior, identifying bottlenecks, and ensuring optimal performance.

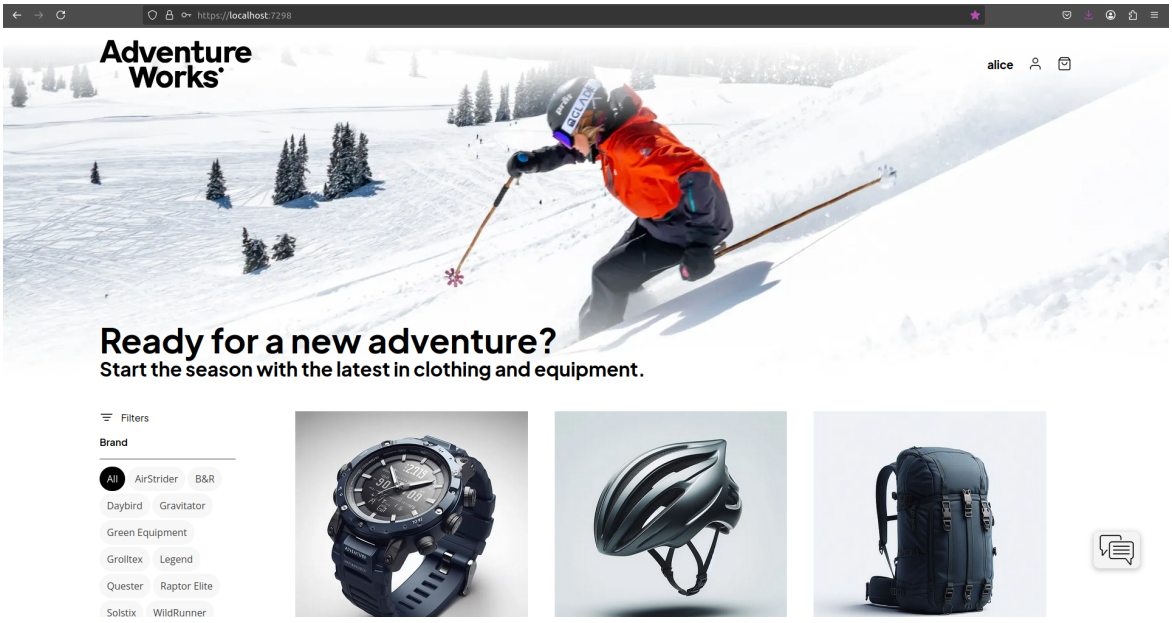

Fig. 1: Adventure Works (eShop Website)

I-B Objectives of the Project

The primary goal of this project was to enhance the observability and security of the **eShop** [2] e-commerce system, through:

- Implementing **OpenTelemetry** [1] tracing for a selected feature.
- Ensuring sensitive data is masked or excluded from logs and telemetry.
- Setting up observability tools for real-time monitoring and analysis, including:
  - ***Jaeger*** [5].
  - ***Prometheus*** [6].
  - ***Grafana*** [7]).
- Creating dashboards to efficiently visualize metrics and traces.

II Selected Feature & Architecture Diagram

II-A Feature Chosen

The "**Place an Order**" feature was selected as the focus of this implementation. The request flow spans multiple services, and each service contributes to the overall transaction.

II-B Basic Architecture Diagram

The following diagram (2) illustrates a basic architecture diagram to visualize the interactions involved in ”Placing an Order”:

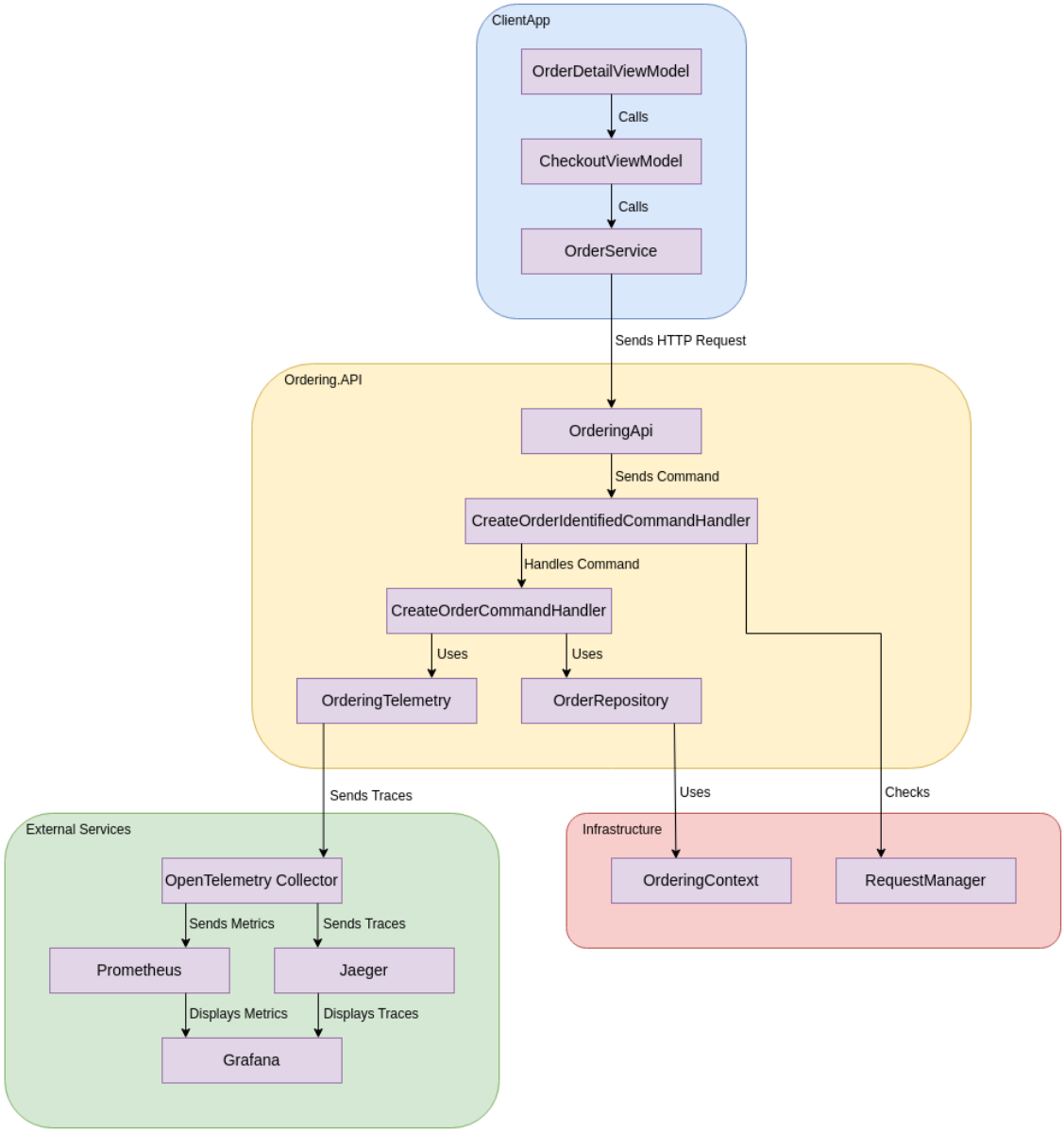

Fig. 2: Basic Architecture Diagram

III OpenTelemetry Integration

III-A Integration Setup

To integrate **OpenTelemtry** [1], the following steps were taken:

- Installed the required packages to the project, as evidenced in the ”*Directory.Packages.props*” file.
- Configured **OpenTelemetry** [1] in each service by modifying their respective ”*Program.cs*” files. For example, in the ”*Ordering.API*” (3):

```
var tracingOtlpEndpoint = builder.Configuration["OTLP_ENDPOINT_URL"];
var otel = builder.Services.AddOpenTelemetry();

builder.AddServiceDefaults();
builder.AddApplicationServices();
builder.Services.AddProblemDetails();

builder.Services.AddRazorComponents().AddInteractiveServerComponents();

otel.ConfigureResource(resource => resource
      .AddService("ordering-api"));

otel.WithMetrics(metrics => metrics
      .AddAspNetCoreInstrumentation()
      .AddRuntimeInstrumentation()
      .AddPrometheusExporter()
      .AddMeter("Microsoft.AspNetCore.Hosting")
      .AddMeter("Microsoft.AspNetCore.Server.Kestrel")
      .AddMeter("System.Net.Http")
      .AddMeter("System.Net.NameResolution"));

otel.WithTracing(tracing =>
{
      tracing.AddProcessor(new PiiScrubberProcessor());
      tracing.AddAspNetCoreInstrumentation();
      tracing.AddHttpClientInstrumentation();
      tracing.AddSource("eShop.WebApp");
      if (tracingOtlpEndpoint != null)
      {
            tracing.AddOtlpExporter(otlpOptions =>
                  {
                        otlpOptions.Endpoint = new Uri(tracingOtlpEndpoint);
                  });
      }
      else
      {
            tracing.AddConsoleExporter();
      }
});
```

Fig. 3: OpenTelemetry Integration Code (1)

(Similar configurations were applied to "*WebApp*" and "*Identity.API*".)

- Key service calls wrapped with "`ActivitySource`" spans to capture distributed traces. For example, in the "*CreateOrderCommandHandler.cs*" of the "*Ordering.API*" (4, 5):

```
public async Task<bool> Handle(CreateOrderCommand message, CancellationToken cancellationToken)
{
    using var activity = OrderingTelemetry.ActivitySource.StartActivity("CreateOrder");

    activity?.SetTag("userId", message.UserId ?? "unknown");
    activity?.SetTag("orderItems", message.OrderItems?.Count() ?? 0);

    try
```

Fig. 4: OpenTelemetry Integration Code (2)

```
    // Record success metric
    if (result)
    {
        _telemetry.OrdersCreatedCounter.Add(1);
        activity?.SetTag("orderCreated", true);
    }

    return result;
}
catch (Exception ex)
{
    // Record failure metric
    _telemetry.OrdersFailedCounter.Add(1);

    activity?.SetTag("error", true);
    activity?.SetTag("exception", ex.ToString());

    _logger.LogError(ex, "Error creating order for user {UserId}", message.UserId);
    throw;
}
```

Fig. 5: OpenTelemetry Integration Code (3)

III-B Data Scrubbing & Security Measures

To **protect sensitive user data**, a custom processor, "`PiiScrubberProcessor` was implemented within "*eShop.ServiceDefaults/Telem*
This processor masks/removes Personality Identifiable Information (**PII**) before traces are sent to exporters.

Implementation of PII scrubbing (6):

```
public class PiiScrubberProcessor : BaseProcessor<Activity>
{
    2 references
    private static readonly Regex EmailPattern = new(@"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}", RegexOptions.Compiled);
    2 references
    private static readonly Regex CreditCardPattern = new(@"\b(?:\d{4}[-\s]?){3}\d{4}\b", RegexOptions.Compiled);
    1 reference
    private static readonly HashSet<string> SensitiveKeys = new(StringComparer.OrdinalIgnoreCase)
    {
        "email", "card", "password", "creditcard", "credit", "phone", "address", "ssn", "username", "cvv", "db.user", "db.connection_string"
    };
    1 reference | Tabnine | Edit | Test | Explain | Document
    private string MaskEmail(string value)
    {
        return EmailPattern.Replace(value, match =>
        {
            var email = match.Value;
            var atIndex = email.IndexOf('@');
            if (atIndex <= 1) return "***@***";

            return email[0] + "***" + email.Substring(atIndex);
        });
    }
    1 reference | Tabnine | Edit | Test | Explain | Document
    private string MaskCreditCard(string value)
    {
        return CreditCardPattern.Replace(value, match => "****-****-****-" + match.Value.Substring(match.Value.Length - 4));
    }
    0 references | Tabnine | Edit | Test | Explain | Document
    private string ScrubValue(string value)
    {
        if (string.IsNullOrEmpty(value)) return value;

        value = MaskEmail(value);
        value = MaskCreditCard(value);

        return value;
    }
    1 reference | Tabnine | Edit | Test | Explain | Document
    private bool IsSensitiveKey(string key)
    {
        if (string.IsNullOrEmpty(key)) return false;

        return SensitiveKeys.Any(k => key.ToLowerInvariant().Contains(k));
    }
    0 references | Tabnine | Edit | Test | Explain | Document
    public override void OnEnd(Activity activity)
    {
        if (activity == null) return;
        // Process activity tags
        foreach (var tag in activity.Tags.ToList())
        {
            if (IsSensitiveKey(tag.Key) || EmailPattern.IsMatch(tag.Key) || CreditCardPattern.IsMatch(tag.Key))
            {
                int length = tag.Value?.Length ?? 0;
                activity.SetTag(tag.Key, new string('*', length));
            }
        }
        base.OnEnd(activity);
    }
}
```

Fig. 6: PII Scrubbing Implementation Code

This processor was applied across services like "*eShop.ServiceDefaults*" and "*Ordering.API*" to ensure consistent data privacy protection.

III-C Exporters & Visualization

To visualize traces and metrics, the following exporters were configured:

- ***Jaeger*** [5]: For distributed tracing (http://localhost:16686).
- ***Prometheus*** [6]: For metrics collection (http://localhost:9090)
- ***Grafana*** [7]: For data visualization (http://localhost:3000)

Configurations of these tools were defined in `docker-compose.observability.yml`, enabling deployment in a containerized environment. This docker-compose file can be seen below (7):

Fig. 7: Docker-Compose

Custom Grafana [7] dashboards were created for tace visualization, defined in:

- "*grafana/provisioning/dashboards/grafanaDashboard.json*".
- "*grafana/provisioning/dashboards/dashboards.yml*".

Below are some screenshot examples of *Jaeger* [5] (8, 9), *Prometheus* [6] (10), and *Grafana* [7] (11, 12) implemented:

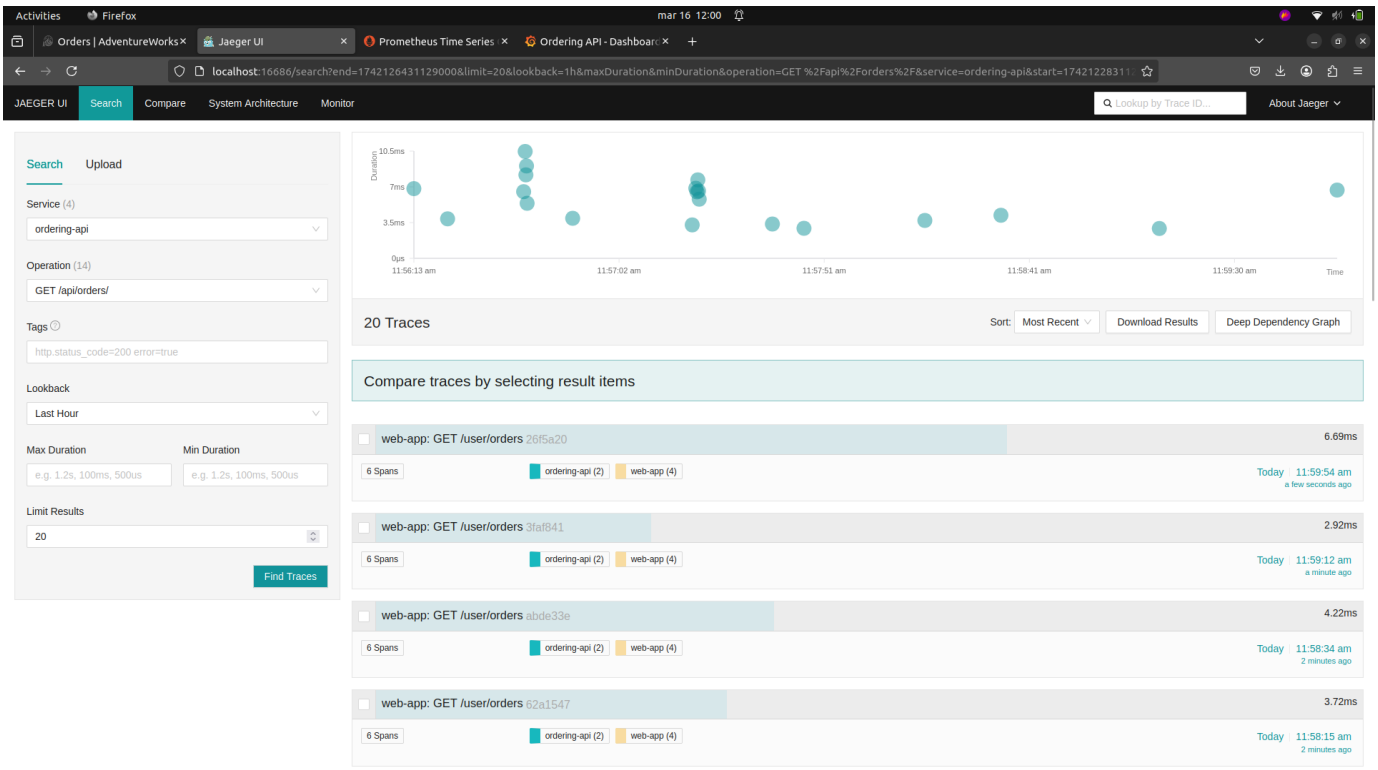

Fig. 8: Jaeger (1)

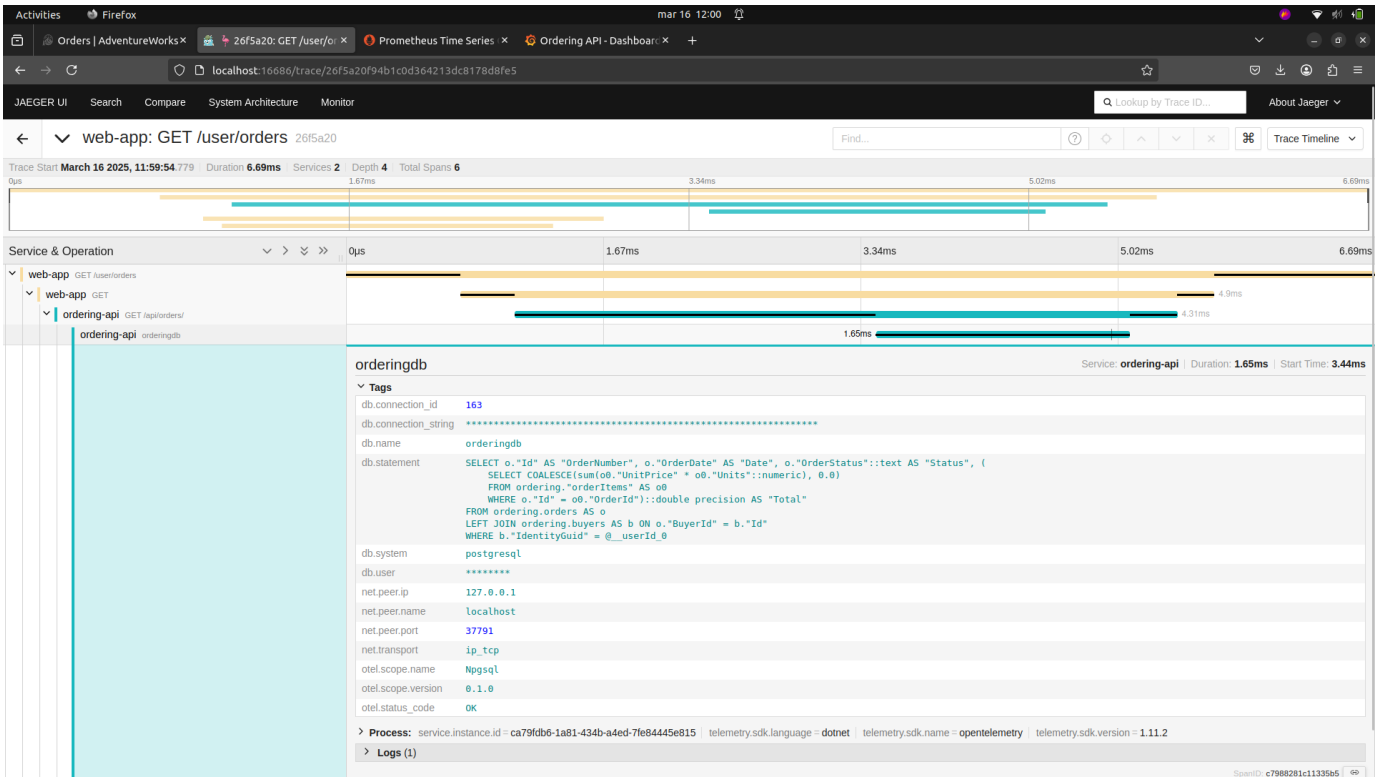

Fig. 9: Jaeger (2)

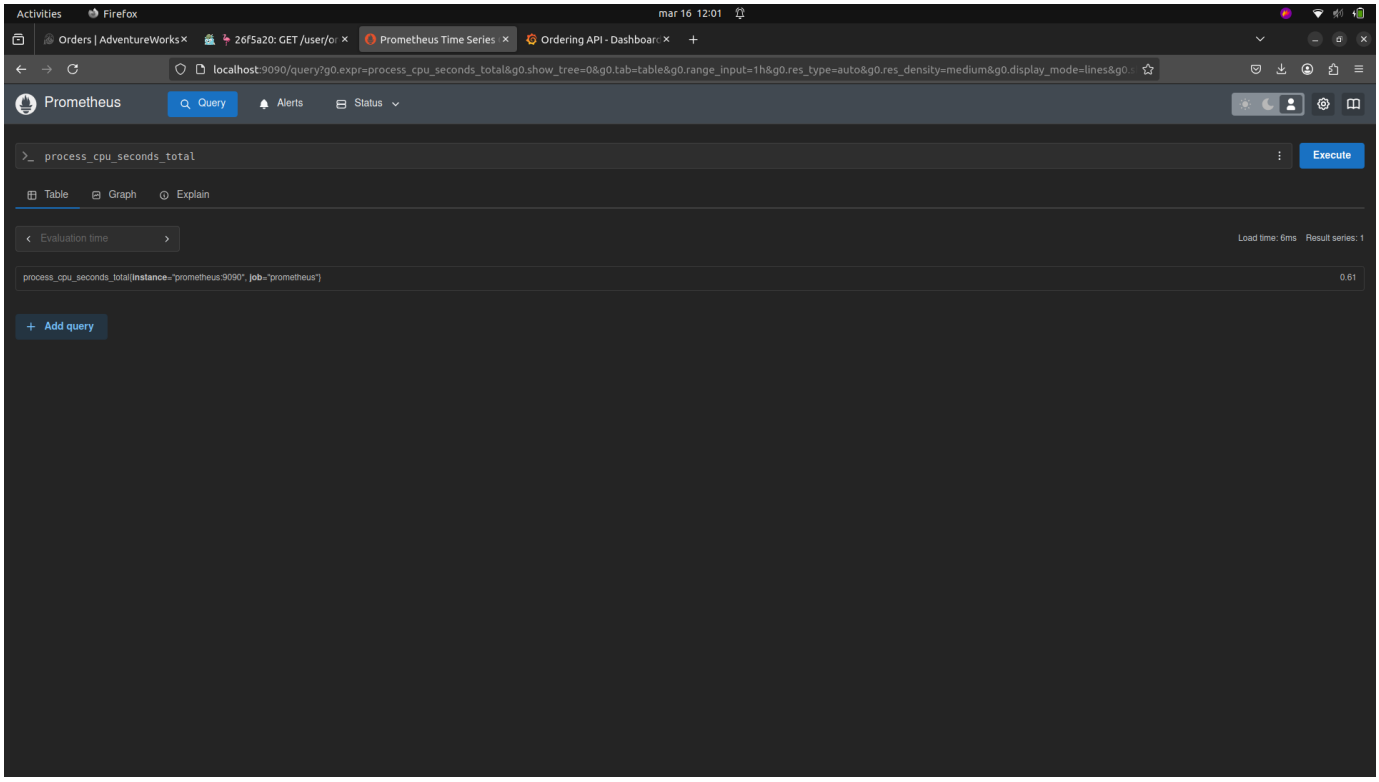

Fig. 10: Prometheus

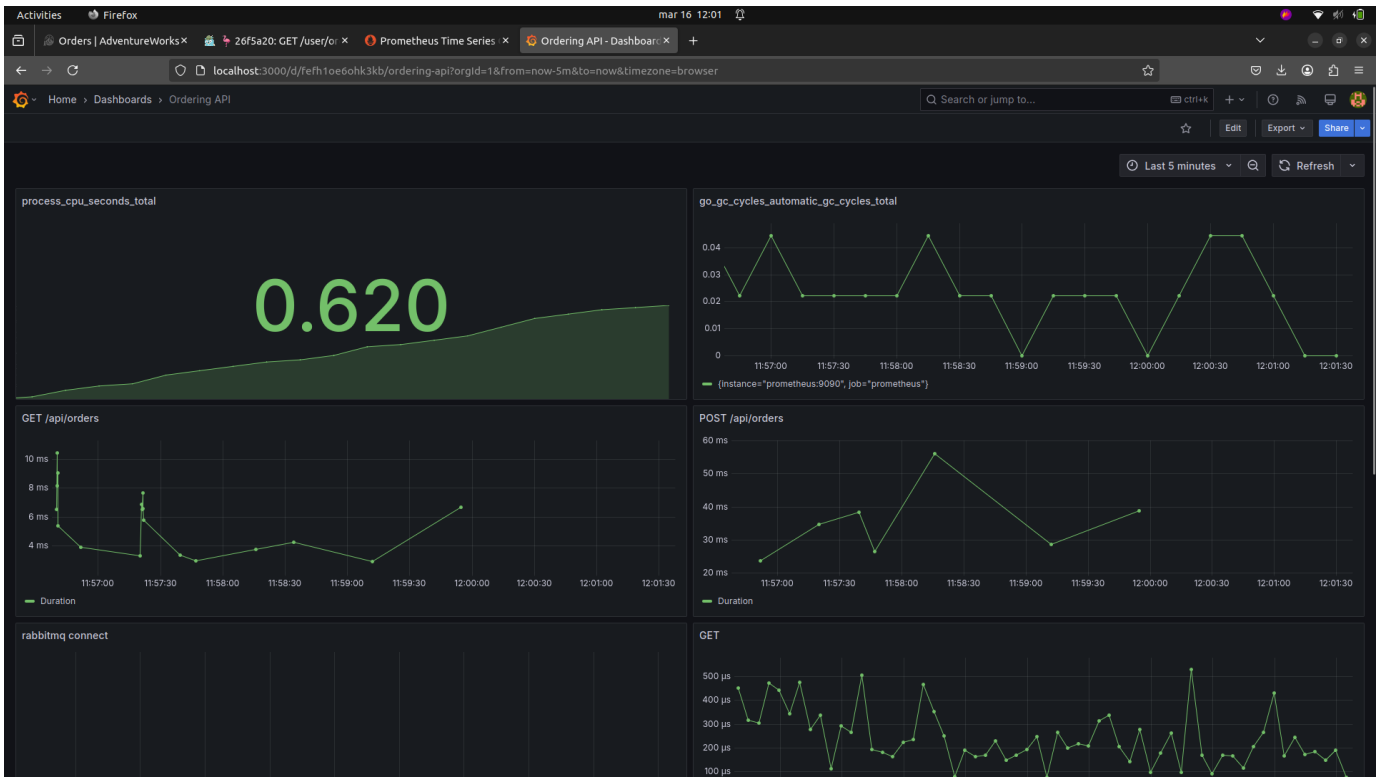

Fig. 11: Grafana (1)

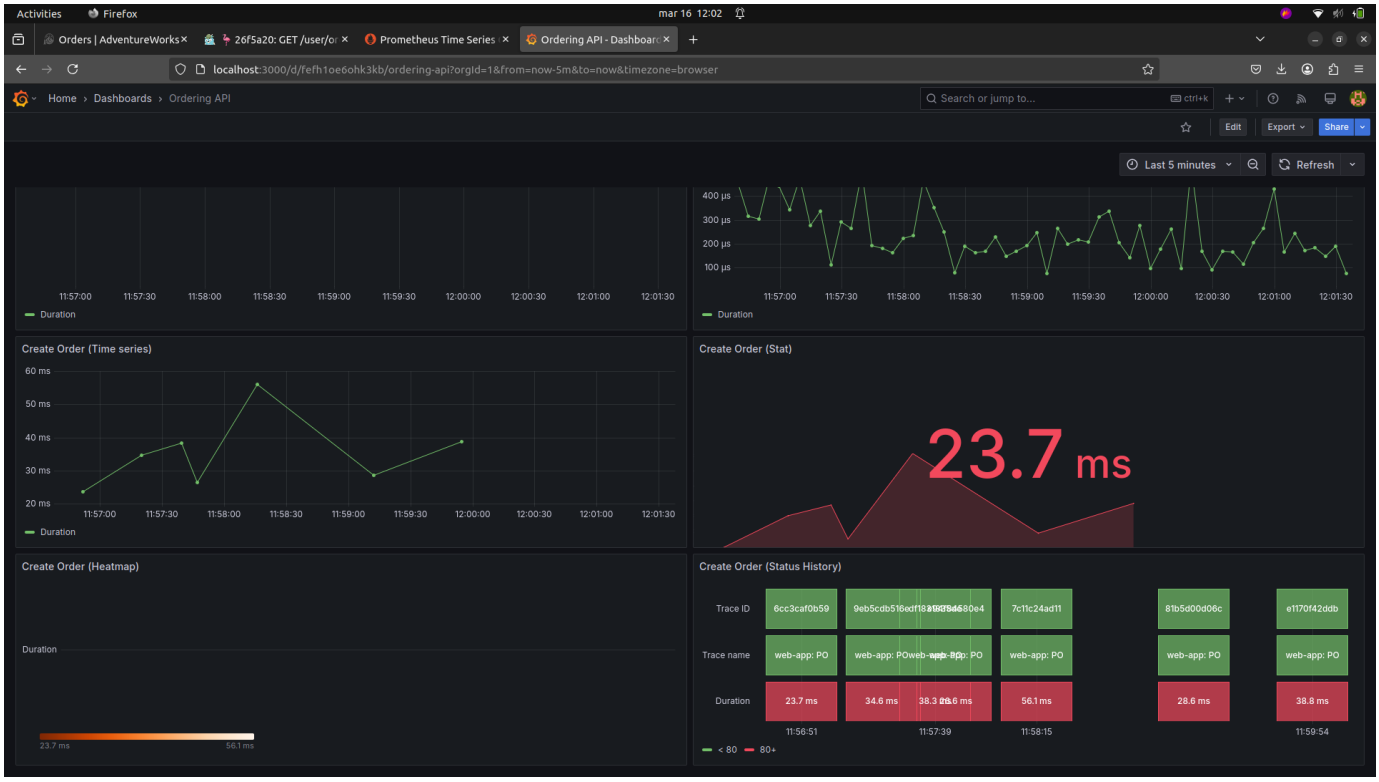

Fig. 12: Grafana (2)

Note: A demo can also be found in the GitHub repository, on he "*report_and_demo*" folder (demo.mp4).

IV Observability & Load Testing

IV-A Metrics Collection

The application was configured to collect various metrics using OpenTelemetry [1] (13):

```
otel.WithMetrics(metrics => metrics
        .AddAspNetCoreInstrumentation()
        .AddRuntimeInstrumentation()
        .AddPrometheusExporter()
        .AddMeter("Microsoft.AspNetCore.Hosting")
        .AddMeter("Microsoft.AspNetCore.Server.Kestrel")
        .AddMeter("System.Net.Http")
        .AddMeter("System.Net.NameResolution"));
```

Fig. 13: Metrics Collection Code

The metrics collected include:

- Infrastructure Metrics:
  - `ASP.NET` Core.
  - Runtime Metrics.
  - HTTP Client.
- Specific System Metrics:
  - `ASP.NET` Core Hosting.
  - Kestrel Web Service.
  - HTTP Client.
  - DNS Resolution.
- Application-Specific Metrics:
  - AI-Related Metrics.
  - Ordering API Metrics.

These metrics provide insights into HTTP requests, runtime behavior, network operations, ...

IV-B Load Testing

Load testing was implemented using *K6*, as specified in the "*loadTest.js*" (14):

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { fail } from 'k6';

export let options = {
    stages: [
        { duration: '30s', target: 10 }, // ramp up to 10 users
        { duration: '1m', target: 10 },  // stay at 10 users for 1 minute
        { duration: '30s', target: 0 },  // ramp down to 0 users
    ],
    insecureSkipTLSVerify: true, // Disable TLS verification
};

Tabnine | Edit | Test | Explain | Document
export default function () {
    // Navigate to the homepage
    let res = http.get('https://localhost:7298');
    check(res, { 'status was 200': (r) => r.status === 200 }) || fail('Failed to load homepage');


    // Log in as a user
    // ! Replace the URL with the login URL of the application
    let url = 'https://localhost:5243/Account/Login?ReturnUrl=%2Fconnect%2Fauthorize%2Fcallback%3Fr
    res = http.post(
        url,
        {
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded'
            },
        }
    );
    check(res, { 'login successful': (r) => r.status === 200 }) || fail('Failed to log in');

    sleep(1);
}
```

Fig. 14: Load Testing

Its results were (15):

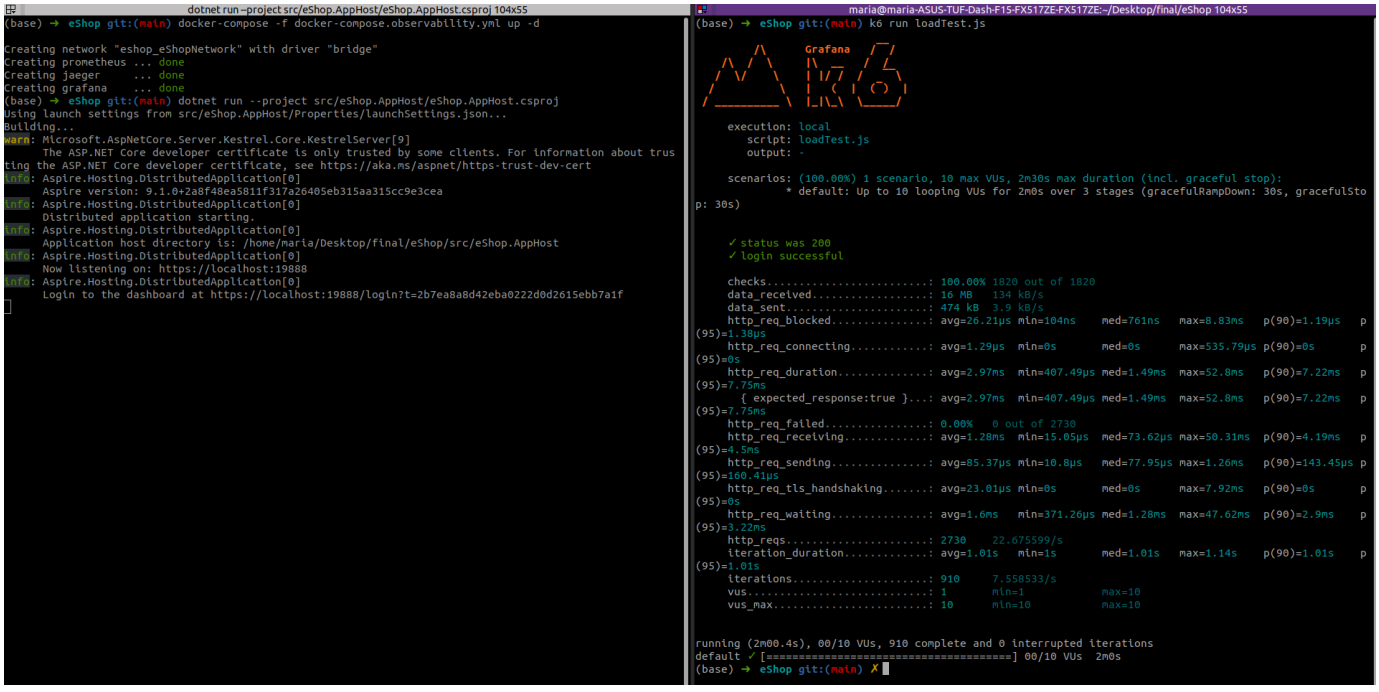

Fig. 15: Running K6

V How to run the program:

In the project's folder (**eShop**), run:

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

dotnet run --project src/eShop.AppHost/eShop.AppHost.csproj
```

For the load tests, run:

```
k6 run loadTest.js
```

VI Challenges Faced

Most of the difficulties I had came from understanding how the application itself worked. Many times, the application (*eShop* [2]), would simply fail unexpectedly without clear reasons. For instance, the basket service encountered errors even when no modifications were made to this part of the code. Restarting the application multiple times eventually resolved these issues.

Besides these problems with the application it self, some other challenges included:

- Configure consistent tracing.
- Ensure proper masking of sensitive information.
- Set up the observability tools and their integration.

VII A.I. Tools Used

A.I. assistance was used for:

- Researching **OpenTelemetry** [1] best practices.
- Debugging and troubleshooting implementation issues.

The following A.I. tools were utilized:

- *GitHub Copilot* [10].
- *Tabnine* [11].
- **ChatGPT** [12].

VIII Conclusions

Overall, I think this project successfully met all core requirements, except for the optional task of implementing column masking at the database layer.

VIII-A Personal Takeaways

Through this project, I gained knowledge in **OpenTelemetry**, observability, and distributed tracing. I also think that implementing real-time monitoring tools provided valuable experience in enhancing system visibility and debugging microservices architectures.

VIII-B Future Work

Future work on this project would include:

- Enhancing security mechanisms further, particularly in database encryption.
- Expanding load testing scenarios.
- Adding more metrics.
- Exploring additional tracing features.
- Refining dashboards.

References

[1] OpenTelemtry. Retrieved from: https://opentelemetry.io/
[2] eShop Repository. Retrieved from: https://github.com/dotnet/eShop/tree/main
[3] S.A. Course - Elearning. Retrieved from: https://elearning.ua.pt/course/view.php?id=5302
[4] S.A. 1st Individual Project Guidelines - Elearning. Retrieved from: https://docs.google.com/document/d/1A9Lr4qx8_phgnCYX-rWDPQkUhGacDsy4_P9mIpOh6Ew/edit?tab=t.0
[5] Jaeger. Retrieved from: https://www.jaegertracing.io/
[6] Prometheus. Retrieved from: https://prometheus.io/docs/introduction/overview/
[7] Grafana. Retrieved from: https://grafana.com/
[8] K6. Retrieved from: https://grafana.com/docs/k6/latest/
[9] ASP.NET Core. Retrieved from: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-9.0
[10] GitHub Copilot. Retrieved from: https://github.com/features/copilot
[11] Tabnine. Retrieved from: https://www.tabnine.com/
[12] ChatGPT. Retrieved from: https://chatgpt.com/