Assignment 02.02: Integrating OpenTelemetry & Security in eShop

Guilherme Vieira, 108671

March 16, 2025

1 Introduction

This report documents the implementation of OpenTelemetry tracing and security measures in the eShop application, a modern e-commerce system built using ASP.NET Core and microservices architecture. The project focused on instrumenting a single feature (the ordering flow) with end-to-end tracing while ensuring sensitive data such as personal information and payment details are properly masked or excluded from telemetry and logs. The implementation also includes a basic Grafana dashboard setup for visualizing traces and metrics.

2 Project Overview

The eShop application is structured as a set of microservices, each handling specific bounded contexts of the e-commerce domain. The main microservices include Catalog, Basket, and Ordering. For this assignment, I focused on implementing OpenTelemetry tracing on the Ordering service, specifically on the order creation flow, which involves several steps from draft creation to order submission, payment processing, and shipping.

3 Technical Implementation

3.1 Feature Selection and Instrumentation Approach

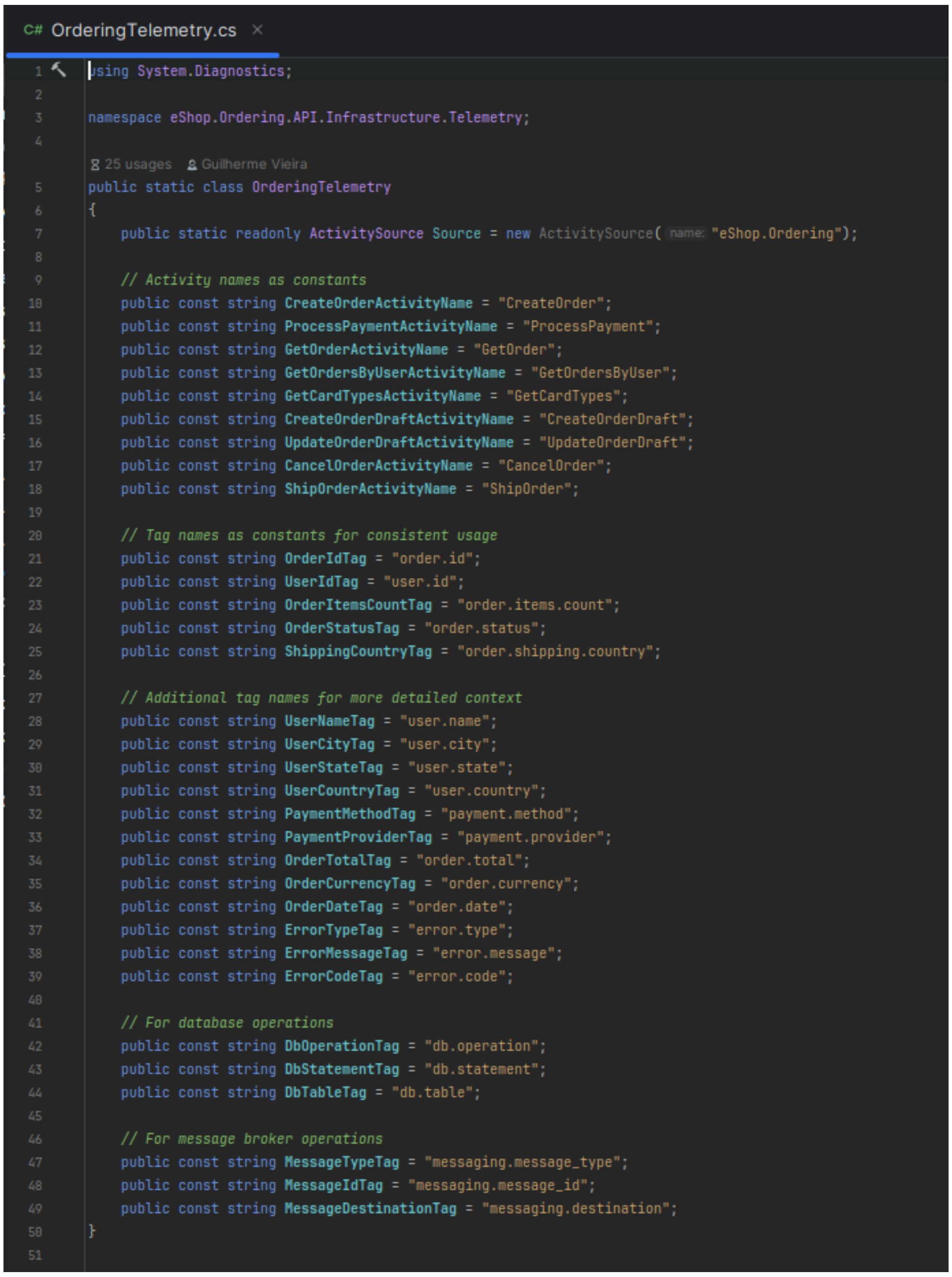
I selected the "Place an Order" flow as the primary feature to instrument with OpenTelemetry. The implementation approach involved adding instrumentation at multiple levels:

- Creating custom Activity sources for detailed tracing
- Adding span creation around key operations
- Implementing metrics collection for important business events
- Redacting sensitive information from traces and logs

3.2 OpenTelemetry Tracing Implementation

It's important to note that the eShop application already had basic telemetry implemented as part of its service defaults. For the Ordering API, we redirected the OpenTelemetry Protocol (OTLP) to our custom observability stack, which allowed us to enhance and customize the telemetry while maintaining compatibility with the existing infrastructure.

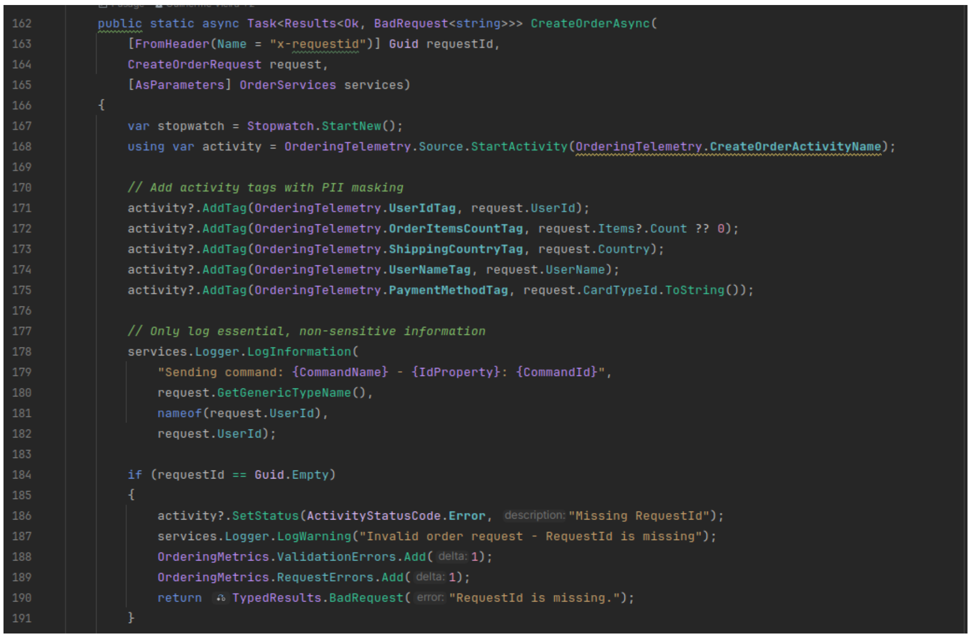
The core of our implementation centers around establishing proper tracing infrastructure in the Ordering.API service. A dedicated static class, OrderingTelemetry, was created to define an ActivitySource and various activity names as constants for consistent usage throughout the codebase.



```csharp
using System.Diagnostics;

namespace eShop.Ordering.API.Infrastructure.Telemetry;

public static class OrderingTelemetry
{
    public static readonly ActivitySource Source = new ActivitySource( name: "eShop.Ordering");

    // Activity names as constants
    public const string CreateOrderActivityName = "CreateOrder";
    public const string ProcessPaymentActivityName = "ProcessPayment";
    public const string GetOrderActivityName = "GetOrder";
    public const string GetOrdersByUserActivityName = "GetOrdersByUser";
    public const string GetCardTypesActivityName = "GetCardTypes";
    public const string CreateOrderDraftActivityName = "CreateOrderDraft";
    public const string UpdateOrderDraftActivityName = "UpdateOrderDraft";
    public const string CancelOrderActivityName = "CancelOrder";
    public const string ShipOrderActivityName = "ShipOrder";

    // Tag names as constants for consistent usage
    public const string OrderIdTag = "order.id";
    public const string UserIdTag = "user.id";
    public const string OrderItemsCountTag = "order.items.count";
    public const string OrderStatusTag = "order.status";
    public const string ShippingCountryTag = "order.shipping.country";

    // Additional tag names for more detailed context
    public const string UserNameTag = "user.name";
    public const string UserCityTag = "user.city";
    public const string UserStateTag = "user.state";
    public const string UserCountryTag = "user.country";
    public const string PaymentMethodTag = "payment.method";
    public const string PaymentProviderTag = "payment.provider";
    public const string OrderTotalTag = "order.total";
    public const string OrderCurrencyTag = "order.currency";
    public const string OrderDateTag = "order.date";
    public const string ErrorTypeTag = "error.type";
    public const string ErrorMessageTag = "error.message";
    public const string ErrorCodeTag = "error.code";

    // For database operations
    public const string DbOperationTag = "db.operation";
    public const string DbStatementTag = "db.statement";
    public const string DbTableTag = "db.table";

    // For message broker operations
    public const string MessageTypeTag = "messaging.message_type";
    public const string MessageIdTag = "messaging.message_id";
    public const string MessageDestinationTag = "messaging.destination";
}
```

Figure 1: OrderingTelemetry Implementation

Tracing was implemented by wrapping key API endpoints with custom activities. For example, the CreateOrder endpoint now creates a span using OrderingTelemetry.Source.StartActivity(), which allows tracking of the entire order creation process. Each activity is enriched with relevant tags such as order ID, user ID, and order status, while ensuring that sensitive information is properly masked.



```csharp
public static async Task<Results<Ok, BadRequest<string>>> CreateOrderAsync(
    [FromHeader(Name = "x-requestid")] Guid requestId,
    CreateOrderRequest request,
    [AsParameters] OrderServices services)
{
    var stopwatch = Stopwatch.StartNew();
    using var activity = OrderingTelemetry.Source.StartActivity(OrderingTelemetry.CreateOrderActivityName);

    // Add activity tags with PII masking
    activity?.AddTag(OrderingTelemetry.UserIdTag, request.UserId);
    activity?.AddTag(OrderingTelemetry.OrderItemsCountTag, request.Items?.Count ?? 0);
    activity?.AddTag(OrderingTelemetry.ShippingCountryTag, request.Country);
    activity?.AddTag(OrderingTelemetry.UserNameTag, request.UserName);
    activity?.AddTag(OrderingTelemetry.PaymentMethodTag, request.CardTypeId.ToString());

    // Only log essential, non-sensitive information
    services.Logger.LogInformation(
        "Sending command: {CommandName} - {IdProperty}: {CommandId}",
        request.GetGenericTypeName(),
        nameof(request.UserId),
        request.UserId);

    if (requestId == Guid.Empty)
    {
        activity?.SetStatus(ActivityStatusCode.Error, description: "Missing RequestId");
        services.Logger.LogWarning("Invalid order request - RequestId is missing");
        OrderingMetrics.ValidationErrors.Add( delta: 1);
        OrderingMetrics.RequestErrors.Add( delta: 1);
        return  TypedResults.BadRequest( error: "RequestId is missing.");
    }
```

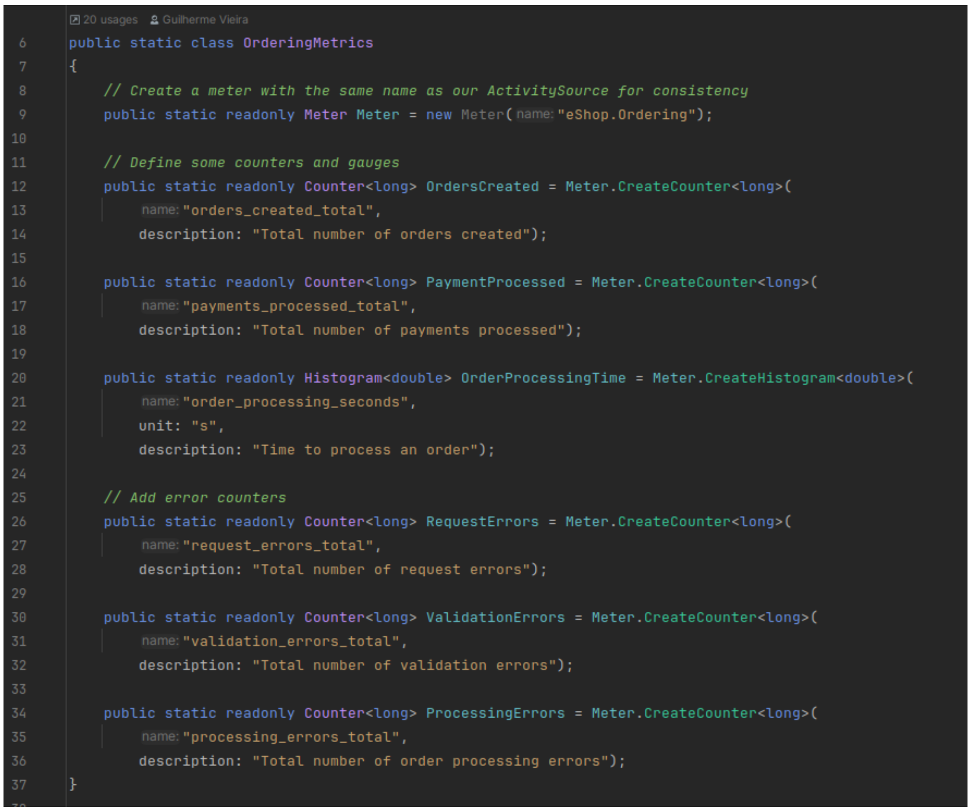
Figure 2: CreateOrder Activity Implementation

The implementation also extends to capturing meaningful events within each operation. For instance, when an order fails validation, the activity status is set to Error with an appropriate message, which provides valuable context for troubleshooting.

3.3 Metrics Collection

Beyond tracing, a comprehensive metrics collection system was implemented using OpenTelemetry's Meter API to provide quantitative insights into the ordering process. The metrics strategy focuses on four main categories of information: order processing volume, payment processing statistics, timing data, and error tracking.

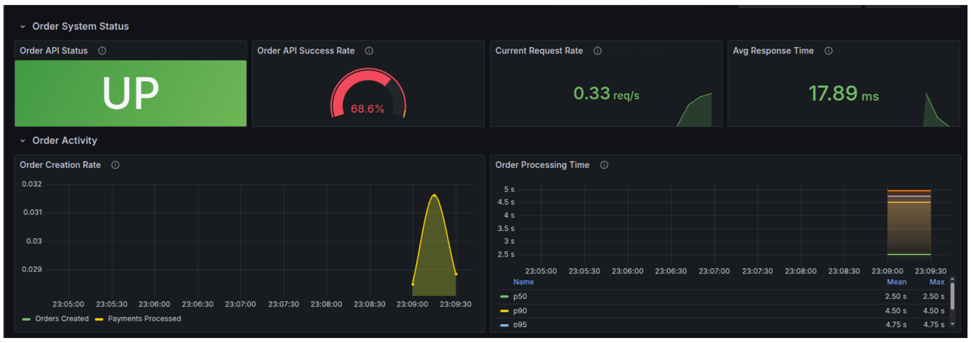
For volume tracking, counters were implemented to track order creation counts over time, allowing for analysis of order patterns and trends. Payment processing metrics include successful payment counts and breakdowns by payment method, providing insights into the financial aspects of the system. Order processing time is captured using histograms that record the duration of each step in the order flow, enabling detailed performance analysis. Finally, error metrics are categorized by type (validation, processing, request) to facilitate targeted troubleshooting and improvement efforts.



```
20 usages  Guilherme Vieira
public static class OrderingMetrics
{
    // Create a meter with the same name as our ActivitySource for consistency
    public static readonly Meter Meter = new Meter( name: "eShop.Ordering");

    // Define some counters and gauges
    public static readonly Counter<long> OrdersCreated = Meter.CreateCounter<long>(
         name: "orders_created_total",
        description: "Total number of orders created");

    public static readonly Counter<long> PaymentProcessed = Meter.CreateCounter<long>(
         name: "payments_processed_total",
        description: "Total number of payments processed");

    public static readonly Histogram<double> OrderProcessingTime = Meter.CreateHistogram<double>(
         name: "order_processing_seconds",
        unit: "s",
        description: "Time to process an order");

    // Add error counters
    public static readonly Counter<long> RequestErrors = Meter.CreateCounter<long>(
         name: "request_errors_total",
        description: "Total number of request errors");

    public static readonly Counter<long> ValidationErrors = Meter.CreateCounter<long>(
         name: "validation_errors_total",
        description: "Total number of validation errors");

    public static readonly Counter<long> ProcessingErrors = Meter.CreateCounter<long>(
         name: "processing_errors_total",
        description: "Total number of order processing errors");
}
```

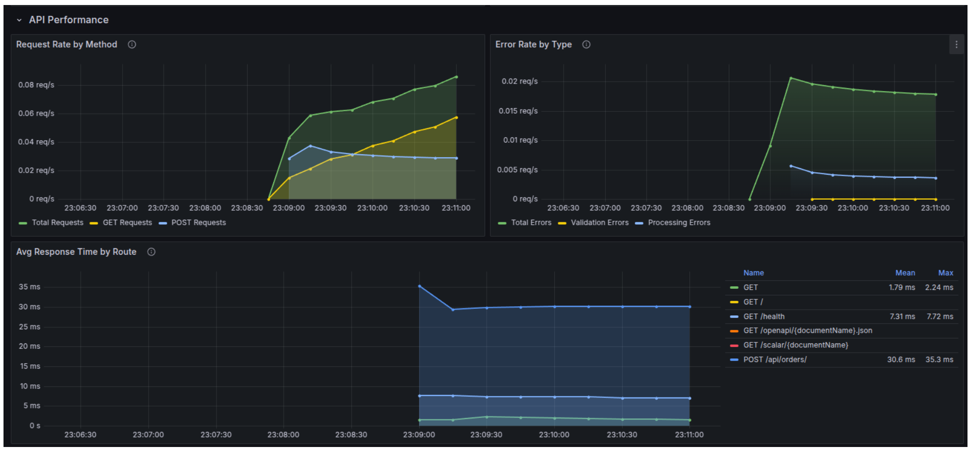
Figure 3: OrderingMetrics Implementation

These metrics are collected using OpenTelemetry's Meter API and exported to Prometheus for storage and querying. The Grafana dashboards then visualize this data through various panels including time-series graphs, counters, and heatmaps. The implementation includes customized Grafana panels that present both real-time operational data and aggregated historical trends.



Name	Mean	Max
p50	2.50 s	2.50 s
p90	4.50 s	4.50 s
p95	4.75 s	4.75 s

Figure 4: Order Activity Metrics Dashboard



	Name	Mean	Max
🟩	GET	1.79 ms	2.24 ms
🟨	GET /		
🟦	GET /health	7.31 ms	7.72 ms
🟧	GET /openapi/{documentName}.json		
🟥	GET /scalar/{documentName}		
🟦	POST /api/orders/	30.6 ms	35.3 ms

Figure 5: API Performance Metrics Dashboard

The metrics implementation provides both business-level and technical visibility into the ordering process, making it possible to track trends, identify issues at a glance, and drill down into specific problem areas when needed. This dual-purpose approach ensures that the telemetry system serves both operational and business intelligence needs.

3.4 Data Protection and PII Handling

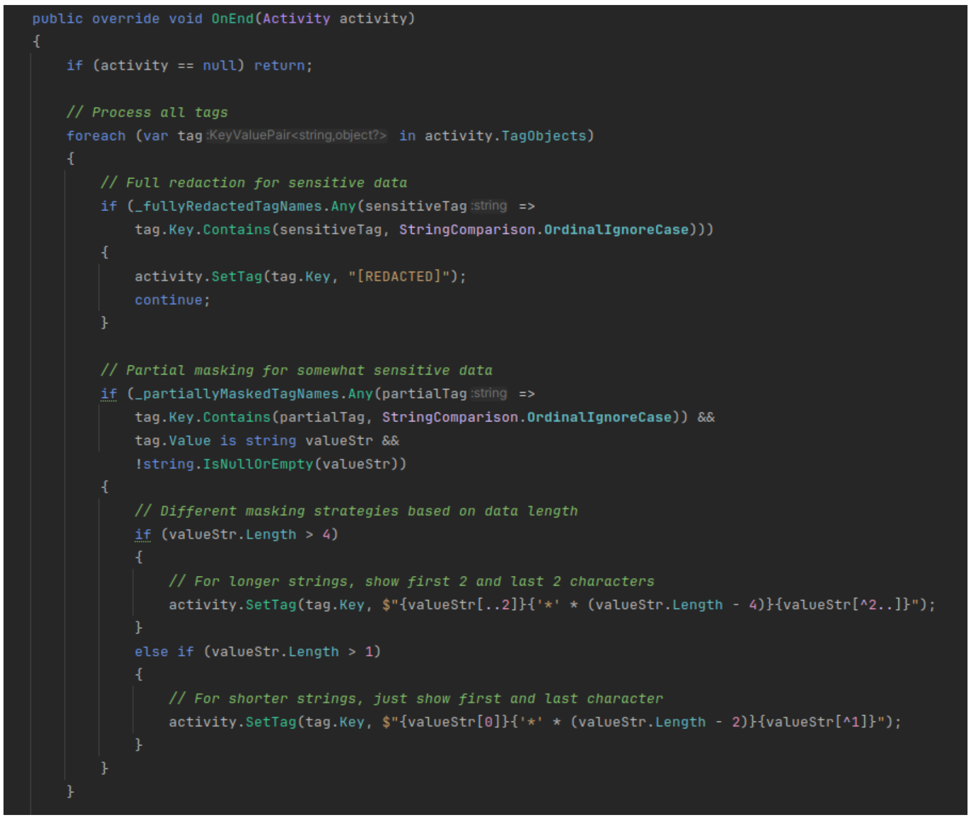
A critical aspect of the implementation was ensuring that Personally Identifiable Information (PII) and sensitive data are properly protected in logs and telemetry. It's worth noting that the eShop application already had some level of data protection implemented natively. For instance, credit card numbers were already being masked in the original code. Our implementation built upon this foundation and expanded the protection to cover more scenarios and data types. This was accomplished through multiple mechanisms:

3.4.1 PII Redaction Processor

A custom PiiRedactionProcessor was implemented as an OpenTelemetry processor that intercepts activities before they are exported. This processor identifies sensitive fields based on predefined lists of tag names and applies appropriate redaction:

- Full redaction for highly sensitive data like credit card numbers, passwords, and tokens
- Partial masking for less sensitive data like user names and addresses, showing only portions of the original value (e.g., first and last characters)

This approach ensures that sensitive information is never exported to external systems in plaintext while preserving enough context for operational purposes.



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

    // Process all tags
    foreach (var tag :KeyValuePair<string,object?> in activity.TagObjects)
    {
        // Full redaction for sensitive data
        if (_fullyRedactedTagNames.Any(sensitiveTag :string =>
            tag.Key.Contains(sensitiveTag, StringComparison.OrdinalIgnoreCase)))
        {
            activity.SetTag(tag.Key, "[REDACTED]");
            continue;
        }

        // Partial masking for somewhat sensitive data
        if (_partiallyMaskedTagNames.Any(partialTag :string =>
            tag.Key.Contains(partialTag, StringComparison.OrdinalIgnoreCase)) &&
            tag.Value is string valueStr &&
            !string.IsNullOrEmpty(valueStr))
        {
            // Different masking strategies based on data length
            if (valueStr.Length > 4)
            {
                // For longer strings, show first 2 and last 2 characters
                activity.SetTag(tag.Key, $"{valueStr[..2]}{'*' * (valueStr.Length - 4)}{valueStr[^2..]}");
            }
            else if (valueStr.Length > 1)
            {
                // For shorter strings, just show first and last character
                activity.SetTag(tag.Key, $"{valueStr[0]}{'*' * (valueStr.Length - 2)}{valueStr[^1]}");
            }
        }
    }
```

Figure 6: PII Redactor Implementation (Only Tags code)

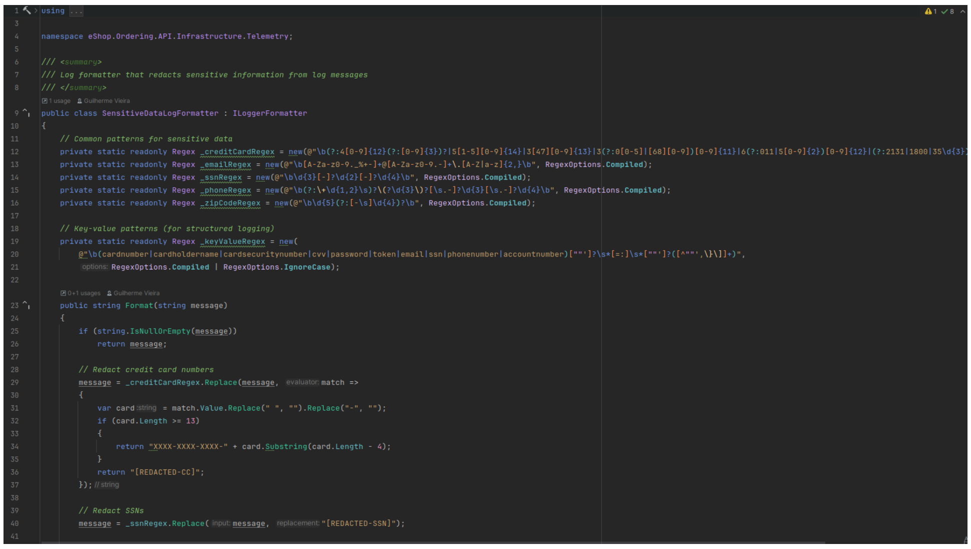
This image only shows the tags redaction process, the events part can be seen in the code.

3.4.2 Sensitive Data Log Filter

Complementary to the trace processor, a SensitiveDataLogFormatter was implemented to filter sensitive information from logs. This formatter uses regular expressions to identify and redact patterns that might contain PII, such as:

- Credit card numbers
- Email addresses
- Social security numbers
- Phone numbers
- Postal codes

The formatter applies different redaction strategies based on the type of data, such as showing only the last four digits of a credit card number or partially masking email addresses to preserve some recognition value.



```csharp
using ...

namespace eShop.Ordering.API.Infrastructure.Telemetry;

/// <summary>
/// Log formatter that redacts sensitive information from log messages
/// </summary>
public class SensitiveDataLogFormatter : ILoggerFormatter
{
    // Common patterns for sensitive data
    private static readonly Regex _creditCardRegex = new(@"\b(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|3[47][0-9]{13}|3(?:0[0-5]|[68][0-9])[0-9]{11}|6(?:011|5[0-9]{2})[0-9]{12}|(?:2131|1800|35\d{3})
    private static readonly Regex _emailRegex = new(@"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b", RegexOptions.Compiled);
    private static readonly Regex _ssnRegex = new(@"\b\d{3}[-]?\d{2}[-]?\d{4}\b", RegexOptions.Compiled);
    private static readonly Regex _phoneRegex = new(@"\b(?:\+\d{1,2}\s)?\(?\d{3}\)?[\s.-]?\d{3}[\s.-]?\d{4}\b", RegexOptions.Compiled);
    private static readonly Regex _zipCodeRegex = new(@"\b\d{5}(?:[-\s]\d{4})?\b", RegexOptions.Compiled);

    // Key-value patterns (for structured logging)
    private static readonly Regex _keyValueRegex = new(
        @"\b(cardnumber|cardholdername|cardsecuritynumber|cvv|password|token|email|ssn|phonenumber|accountnumber)[""']?\s*[=:]\s*[""']?([^""',\}\]]+)",
        RegexOptions.Compiled | RegexOptions.IgnoreCase);

    public string Format(string message)
    {
        if (string.IsNullOrEmpty(message))
            return message;

        // Redact credit card numbers
        message = _creditCardRegex.Replace(message, match =>
        {
            var card = match.Value.Replace(" ", "").Replace("-", "");
            if (card.Length >= 13)
            {
                return "XXXX-XXXX-XXXX-" + card.Substring(card.Length - 4);
            }
            return "[REDACTED-CC]";
        });

        // Redact SSNs
        message = _ssnRegex.Replace(message, "[REDACTED-SSN]");

```

Figure 7: Some part of the Sensitive Data Log Filter Implementation

3.4.3 Modified Logging Practices

The implementation also includes changes to existing logging statements to avoid capturing sensitive data. For example, the CreateOrderDraftAsync method was modified to log only essential, non-sensitive information rather than the entire command object, which might contain payment details or personal information.

3.5 Integration with Observability Stack

The implementation includes configuration for exporting telemetry data to an observability stack. A comprehensive observability pipeline was established with three key components: Jaeger for distributed tracing, Prometheus for metrics collection, and Grafana for visualization and dashboarding.

The OpenTelemetry configuration in the Ordering service was configured to export traces to Jaeger via the OTLP protocol endpoint (configurable via environment variables). This setup allows for detailed visualization of request flows, including parent-child relationships between spans and the timing of each operation in the order processing pipeline. Jaeger's UI provides powerful filtering capabilities to isolate specific traces based on tags like order ID or error status.

For metrics, Prometheus was configured as the collection backend. The Ordering service exposes a metrics endpoint that Prometheus scrapes at regular intervals. This approach allows for efficient collection of time-series data like order counts, processing times, and error rates without significantly impacting application performance.

Grafana serves as the visualization layer, connecting to both Jaeger and Prometheus data sources. Custom dashboards were created to provide both business and technical insights:

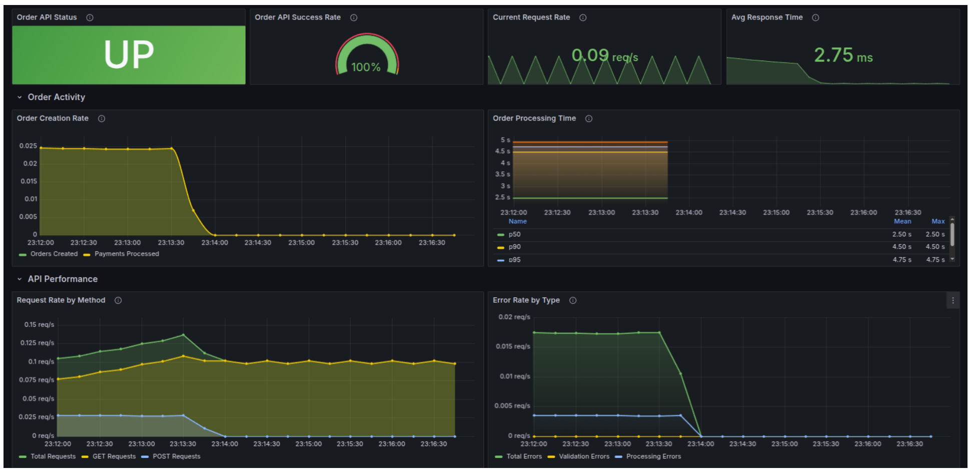
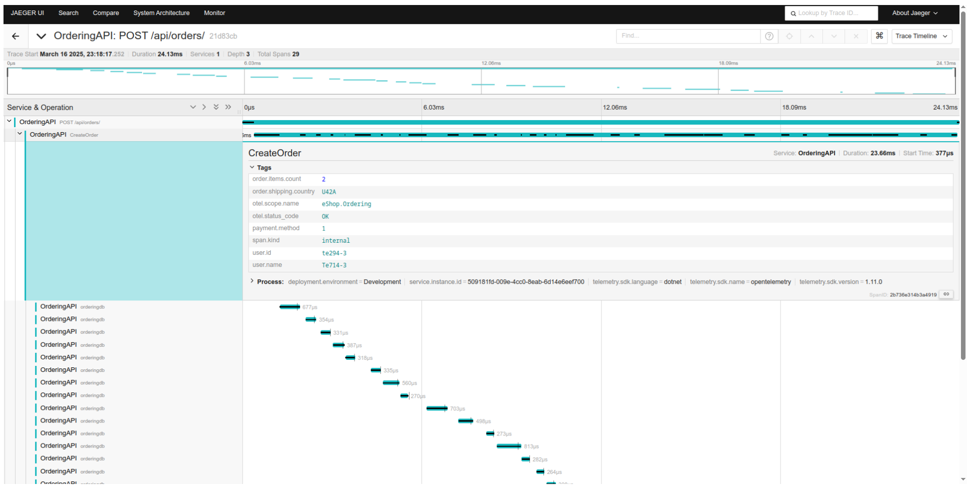


Figure 8: Grafana Overview



OrderingAPI: POST /api/orders/ 21d83cb

Trace Start **March 16 2025, 23:18:17**.252 | Duration **24.13ms** | Services **1** | Depth **3** | Total Spans **29**

Service & Operation

OrderingAPI POST /api/orders/

OrderingAPI CreateOrder

CreateOrder — Service: **OrderingAPI** | Duration: **23.66ms** | Start Time: **377μs**

Tags

order.items.count	2
order.shipping.country	U42A
otel.scope.name	eShop.Ordering
otel.status_code	OK
payment.method	1
span.kind	internal
user.id	te294-3
user.name	Te714-3

Process: deployment.environment = Development | service.instance.id = 509181fd-009e-4cc0-8eab-6d14e6eef700 | telemetry.sdk.language = dotnet | telemetry.sdk.name = opentelemetry | telemetry.sdk.version = 1.11.0

SpanID: 2b736e314b3a4919

Figure 9: Jaeger Trace View of Order Processing

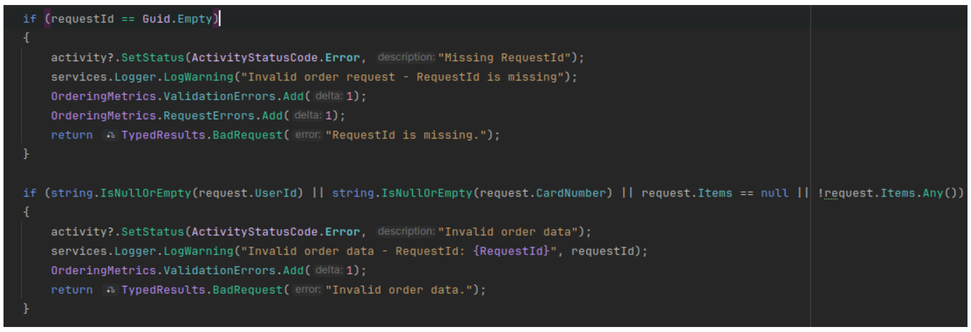
Additionally, a RequestMetricsMiddleware was implemented to track HTTP requests and responses, capturing key performance metrics and error rates at the API level. This middleware automatically records request durations, counts requests by status code category, and maintains running averages of performance metrics. These metrics are then exported to Prometheus and visualized in Grafana dashboards, providing real-time visibility into API performance and health.

4 Security Considerations

The security aspects of the implementation extend beyond just data protection. Several security best practices were incorporated. Regarding database column masking, there was an attempt to implement this feature natively within the application, unfortunately unsuccessfully due to many changes required to the codebase to do a proper implementation.

4.1 Input Validation and Error Handling

The implementation enhances input validation for the order creation process, checking for essential fields and rejecting incomplete requests early. When validation fails, appropriate error status codes are returned, and the failures are logged without exposing sensitive details.



```csharp
if (requestId == Guid.Empty)
{
    activity?.SetStatus(ActivityStatusCode.Error, description: "Missing RequestId");
    services.Logger.LogWarning("Invalid order request - RequestId is missing");
    OrderingMetrics.ValidationErrors.Add( delta: 1);
    OrderingMetrics.RequestErrors.Add( delta: 1);
    return TypedResults.BadRequest( error: "RequestId is missing.");
}

if (string.IsNullOrEmpty(request.UserId) || string.IsNullOrEmpty(request.CardNumber) || request.Items == null || !request.Items.Any())
{
    activity?.SetStatus(ActivityStatusCode.Error, description: "Invalid order data");
    services.Logger.LogWarning("Invalid order data - RequestId: {RequestId}", requestId);
    OrderingMetrics.ValidationErrors.Add( delta: 1);
    return TypedResults.BadRequest( error: "Invalid order data.");
}
```

Figure 10: Enhanced Input Validation Implementation

4.2 Proper Error Messages

Care was taken to ensure that error messages do not leak sensitive information. Error responses include general error descriptions without revealing internal system details or sensitive data patterns.

4.3 Rate Limiting Considerations

While not explicitly implemented, the metric collection infrastructure lays the groundwork for identifying abnormal request patterns, which could be extended to implement rate limiting and anomaly detection as future enhancements.

5 Challenges and Solutions

During the implementation, several challenges were encountered and addressed:

5.1 Balancing Detail vs. Privacy

One significant challenge was finding the right balance between capturing detailed operational information and protecting user privacy. The solution involved categorizing data fields based on sensitivity levels and applying appropriate masking strategies. For example, user IDs are partially masked to retain some recognizability while protecting the full identifier.

5.2 Maintaining Performance

Adding comprehensive tracing and metrics collection introduced some performance overhead. To minimize this impact, the implementation focuses on key business operations rather than instrumenting every method call. Additionally, the PII redaction processor was optimized to use efficient string operations and avoid unnecessary processing.

5.3 Consistency Across Services

While the implementation focused primarily on the Ordering service, ensuring consistency with other services posed a challenge. This was addressed by defining clear naming conventions for activities, tags, and metrics, which can be adopted by other services in the future.

6 Usage of Generative AI Tools

In developing this implementation, I utilized Claude with MCP (Model Control Protocol) tools to assist with various aspects of the development process. The integration with JetBrains Rider using the MCP server plugin made the development workflow seamless.

Claude was particularly helpful throughout the development process. It assisted in generating initial boilerplate code for telemetry classes, which saved significant time and ensured consistency in the implementation. During the PII handling phase, Claude provided valuable guidance on best practices, helping me design effective redaction strategies that balanced privacy with operational needs. For performance-critical sections, Claude suggested several optimizations that improved the efficiency of the telemetry system without compromising its functionality. Throughout the entire development cycle, Claude also reviewed the implementation for potential security issues, helping to identify and address vulnerabilities before they could become problems in production.

The AI assistance significantly accelerated the development process while maintaining code quality and security. I found that Claude's suggestions for security patterns and telemetry practices were well-aligned with industry standards, though I still needed to adapt them to the specific context of the eShop application.

Working with Claude through MCP in Rider provided a more integrated experience than traditional chat interfaces, allowing for contextual code understanding and suggestions. This integration made it easier to implement complex patterns like the PII redaction processor and sensitive data log formatter.

7 Final Thoughts

The assignment provided a valuable opportunity to explore the intersection of observability and security in a microservices architecture. Implementing OpenTelemetry in a production-like application highlighted the importance of thoughtful instrumentation that balances operational visibility with data protection.

7.1 Accomplishments

This assignment resulted in several significant accomplishments. The end-to-end tracing for the ordering flow was successfully implemented, providing visibility into each step of the order processing pipeline from creation to completion. A comprehensive PII protection system was created for both logs and telemetry, ensuring sensitive user data remains protected while maintaining sufficient operational visibility for troubleshooting. The implementation established meaningful business and technical metrics that provide insights into the ordering system's health and performance, enabling proactive monitoring and issue detection. Finally, the telemetry system was seamlessly integrated with industry-standard observability tools like OTLP and Prometheus, allowing for sophisticated visualization and analysis of the collected data through Grafana dashboards.

7.2 Areas for Future Improvement

While the current implementation meets the assignment requirements, several areas could be enhanced in future iterations. The telemetry instrumentation could be extended to cover cross-service interactions, especially between the Ordering and Basket services, providing a more complete picture of the entire user journey across the application. To further strengthen security, role-based access controls could be implemented for sensitive telemetry data, ensuring that only authorized personnel can access potentially sensitive operational information. The rich metrics collected through this implementation could serve as a foundation for implementing anomaly detection capabilities, automatically identifying unusual patterns that might indicate performance issues or security incidents. Additionally, developing automated tests specifically designed to detect PII leakage in telemetry and logs would provide ongoing assurance that sensitive data remains protected as the application evolves and changes over time.

7.3 Lessons Learned

The assignment reinforced the importance of designing observability with security in mind from the beginning. Retrofitting privacy controls onto existing telemetry can be challenging, whereas integrating them from the start ensures a more cohesive approach.

I also gained valuable insights into the practical challenges of implementing OpenTelemetry in a microservices environment. Context propagation across service boundaries proved to be more complex than anticipated, requiring careful consideration of headers and correlation IDs. Maintaining consistent naming conventions for metrics, traces, and tags across different services demanded thoughtful planning and documentation. The setup of the observability stack itself—Jaeger, Prometheus, and Grafana—required significant configuration to achieve the desired level of integration.

One significant challenge that remains unresolved was implementing database-level masking for sensitive data. Despite multiple attempts, I was unable to successfully implement column-level masking within the database itself. This was particularly challenging due to the extensive changes required to the existing codebase and data access layer. The application uses Entity Framework Core with a complex domain model, and implementing proper database masking would have required substantial modifications to the data persistence layer. This experience highlighted the importance of designing data protection mechanisms at the database level early in the application development lifecycle, as retrofitting such changes to an existing system with established data patterns proved to be prohibitively complex within the scope of this assignment.

Working with Grafana for visualization highlighted the importance of dashboard design for effective operational monitoring. Simply collecting metrics and traces is insufficient; presenting them in an intuitive, actionable format requires thought about the dashboard user's needs and workflows. The Grafana dashboards created for this project evolved through several iterations to achieve the right balance of high-level overview and detailed drill-down capabilities.

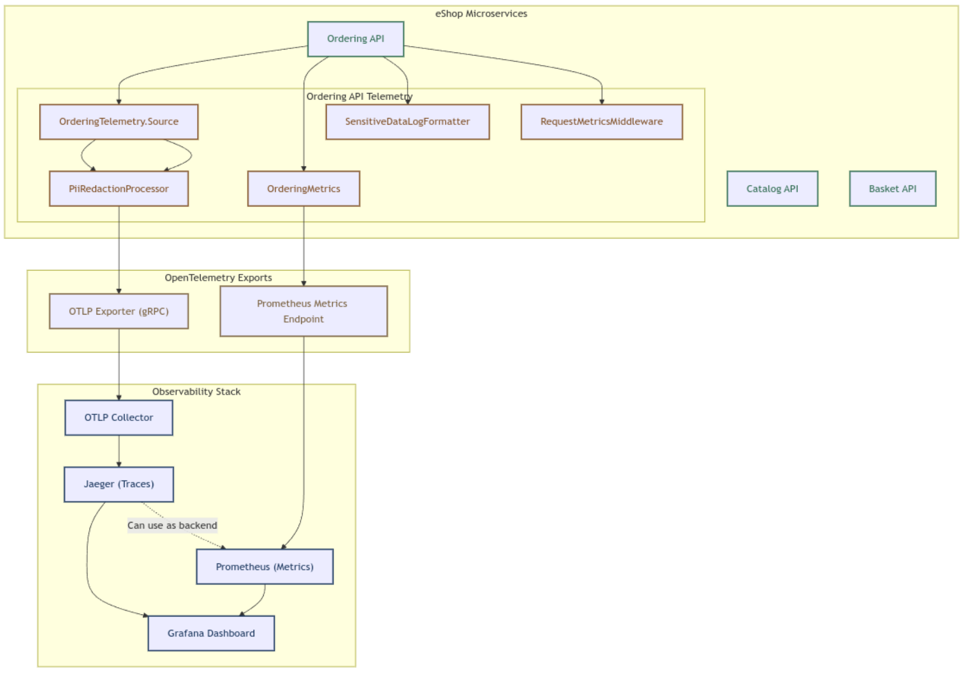


Figure 11: eShop Observability Stack Architecture

Overall, this project demonstrated that effective observability and strong security measures are not opposing goals but complementary aspects of a well-designed system. By thoughtfully implementing both, we can create applications that are both operationally transparent and respectful of user privacy. The combination of tracing, metrics, and visualization tools, when properly implemented with security in mind, creates a powerful platform for understanding system behavior without compromising sensitive information.