



Observability In Jakarta EE Applications with MicroProfile Telemetry



Power Up Your Jakarta EE

User Guide

Contents

Guide Updated: **November 2024**

Introduction	1
What Is Observability	1
The Three Pillars of Observability	1
Logs	2
Metrics	2
Traces	2
Telemetry Data Collection Options	2
OpenTelemetry Overview	2
Unified Standard for Telemetry Data	2
Vendor-Neutral Approach	3
MicroProfile Telemetry Specification	3
Standards-based Approach for Jakarta EE	3
Integration with OpenTelemetry	3
Focus on Tracing (Current Spec Scope)	3
MicroProfile Telemetry Deep Dive	4
The Setup	4
Automatic Instrumentation	4
Manual Instrumentation	5
Agent-based Instrumentation	6
MicroProfile Telemetry In Action	6
Basic Setup and Configuration	6
Automatic Tracing	7
Manual Instrumentation	7
Advanced Configuration	8
Best Practices and Patterns	10
Span Naming Conventions	10
Attribute Management	11
Error Handling	11
Performance Considerations	12
Security Considerations	12
Conclusions	13

Introduction

Understanding the behavior and health of software systems has become more critical—and more challenging—than ever before. As Jakarta EE applications grow in complexity, traditional monitoring approaches fall short of providing the deep insights needed to maintain reliability and performance. This guide introduces you to modern observability practices in Jakarta EE applications, with a specific focus on MicroProfile Telemetry.

Through detailed examples and best practices, you'll learn how to leverage both automatic and manual instrumentation techniques, configure telemetry collection for different environments, as well as implement secure and performant observability solutions. By the end of this guide, you'll have a solid understanding of how to make your Jakarta EE applications more observable and manageable in production environments.

What Is Observability

Observability is the ability to understand a system's internal state by examining its outputs. Unlike traditional monitoring, which tells you when something is wrong, observability helps you understand why something is wrong. It enables teams to:

- Debug production issues efficiently
- Understand system behavior in real-time
- Make data-driven decisions about performance and reliability
- Reduce Mean Time To Resolution (MTTR)
- Proactively identify potential issues before they impact users

Observability is built on three core pillars, together acting as the cornerstone of collecting detailed insights into the running of modern software systems.

The Three Pillars of Observability

Modern observability is built upon three fundamental pillars that work together to provide a complete view of your software system's health and behavior. Each pillar serves a distinct purpose and provides unique insights, yet they are most powerful when used in combination to create a comprehensive observability strategy. Understanding and implementing these pillars effectively is critical for maintaining reliable and performant systems.

Logs

Logs are time stamped records of discrete events that occur within a system. They provide detailed context about specific occurrences. When properly implemented, logs become tools for various operational and business purposes.

Metrics

Metrics are numerical measurements collected over time. They provide quantitative data on system performance and behaviour.

Traces

Traces track the journey of requests as they flow through your application, showing the relationships between the various services and components. This guide is mainly about tracing with MicroProfile Telemetry.

Telemetry Data Collection Options

The landscape of telemetry data collection has evolved significantly, with two major approaches emerging as primary options for Jakarta EE applications: OpenTelemetry and MicroProfile Telemetry. Both options offer full-featured APIs for collecting telemetry data in your applications. Their differences lie in the scope of the underlying projects.

OpenTelemetry Overview

OpenTelemetry has emerged as the de facto standard for telemetry data collection in cloud-native applications. It represents a significant advancement in observability by providing:

Unified Standard for Telemetry Data

- A single, coherent set of APIs, libraries, agents and instrumentation
- Consistent data collection across all three observability pillars (logs, metrics and traces)
- Standardised data formats and semantic conventions
- Support for multiple programming languages and platforms
- Built-in context propagation across service boundaries

Vendor-Neutral Approach

- Open governance under the Cloud Native Computing Foundation (CNCF)
- Wide industry adoption and support
- Pluggable architecture supporting multiple backends
- No vendor lock-in for telemetry data
- Compatible with popular observability platforms (Jaeger, Zipkin, Prometheus, etc.)

MicroProfile Telemetry Specification

The MicroProfile Telemetry specification provides a standards-based approach that is specifically designed for Jakarta EE applications, offering a bridge between traditional enterprise Java applications and modern observability practices.

Standards-based Approach for Jakarta EE

- Native integration with Jakarta EE programming model
- Consistent with Jakarta EE principles and patterns
- Declarative configuration through annotations
- CDI-based integration
- Compatible with existing Jakarta EE applications

Integration with OpenTelemetry

- Built on top of OpenTelemetry APIs
- Uses OpenTelemetry's core functionality
- Provides Jakarta EE-specific conventions
- Automatic context propagation in Jakarta EE environments
- Seamless integration with OpenTelemetry collectors

Focus on Tracing (Current Spec Scope)

- Emphasis on distributed tracing as initial implementation
- Support for common Jakarta EE use cases
- Automatic instrumentation of Jakarta EE components
- Trace context propagation across Jakarta EE boundaries
- Extension points for custom instrumentation

The key differences between these approaches can be found in their scope and target audience:

- OpenTelemetry provides a comprehensive, language-agnostic solution that is suitable for any application architecture
- MicroProfile Telemetry focuses specifically on Jakarta EE applications, providing integrated observability features that align with Jakarta EE development practices

Both options provide solutions for implementing observability in Jakarta EE applications, and they can be used together in many cases. OpenTelemetry offers broader ecosystem compatibility, while MicroProfile Telemetry provides a more integrated experience for Jakarta EE developers. This guide focuses on MicroProfile Telemetry in Jakarta EE applications.

MicroProfile Telemetry Deep Dive

MicroProfile Telemetry provides three distinct approaches to instrumenting your Jakarta EE applications for observability: automatic, manual and agent-based instrumentation. Each approach serves different needs and can be used independently or in combination to achieve comprehensive system observability. Before looking at the approaches, let us first start with the setup.

The Setup

To start with MicroProfile Telemetry tracing on the Jakarta EE Platform, add the following dependencies to your application.

```
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-api</artifactId>
    <version>1.29.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>io.opentelemetry.instrumentation</groupId>
    <artifactId>opentelemetry-instrumentation-annotations</
artifactId>
    <version>2.5.0</version>
    <scope>provided</scope>
</dependency>
```

Automatic Instrumentation

The automatic instrumentation capability is perhaps the most straightforward way to begin collecting telemetry data in your Jakarta EE applications. It provides built-in support for Jakarta REST and MicroProfile REST Client without requiring any code changes. When enabled, your REST endpoints (both server and client-side) are automatically traced, with trace context properly propagated between services. This instrumentation follows OpenTelemetry semantic conventions and integrates with existing Request Tracing Services.

Manual Instrumentation

When you need more fine-grained control over your telemetry data collection, manual instrumentation may be more suited to address your needs. This option offers two primary approaches. The first is the `@WithSpan` annotation, which can be added to any CDI-aware bean methods. This declarative approach allows you to name spans, configure their kind, and capture method parameters as span attributes. For example:

```
@ApplicationScoped
class OrderService {
    @WithSpan("process-order")
    void processOrder(@SpanAttribute("orderId") String orderId) {
        // Processing logic here
    }
}
```

For situations requiring even more control, you can directly interact with the OpenTelemetry Tracer API. This approach gives you complete control over span lifecycle and attributes. The Tracer can be injected via CDI:

```
@RequestScoped
class OrderProcessor {
    @Inject
    Tracer tracer;

    void processComplexOrder() {
        Span span = tracer.spanBuilder("complex-order")
            .setSpanKind(SpanKind.INTERNAL)
            .setAttribute("complexity", "high")
            .startSpan();
        try {
            // Order processing logic
        } finally {
            span.end();
        }
    }
}
```

You can also access and manipulate the current span, either through CDI injection or the `Span.current()` API. This is particularly useful when you need to add attributes to spans created by the automatic instrumentation.

Agent-based Instrumentation

For teams looking to add observability without modifying application code, MicroProfile Telemetry implementations can support agent-based instrumentation. This approach uses a Java agent that attaches to the running JVM and automatically instruments a wide range of common libraries and frameworks. The agent can be configured through properties and can work alongside other instrumentation approaches. While optional, it provides a powerful way to gather telemetry data with minimal setup overhead, when implemented according to the OpenTelemetry Java Instrumentation project guidelines.

This flexibility in instrumentation approaches gives you the flexibility to choose the most appropriate method for your specific needs, whether you're just starting with observability or need detailed control over your telemetry data collection.

MicroProfile Telemetry In Action

Getting started with MicroProfile Telemetry in your Jakarta EE applications involves a few key setup steps, followed by configuration choices that determine how your telemetry data is collected and exported. Let's explore the implementation process from basic setup through advanced configuration options.

Basic Setup and Configuration

MicroProfile Telemetry is disabled by default in Jakarta EE applications to prevent any unexpected impact on your systems. To begin collecting telemetry data, you'll need to explicitly enable it through configuration. The most fundamental configuration property is `otel.sdk.disabled`, which must be set to `false` in any Config Source (mostly the `microprofile-config.properties` file) to activate telemetry collection:

```
otel.sdk.disabled=false
```

You'll also want to configure where your telemetry data is sent. By default, MicroProfile Telemetry uses the [OTLP protocol](#) to export traces to <http://localhost:4317>. For development purposes, you can quickly set up a local OpenTelemetry backend to receive these traces through the [docker-otel-lgtm project](#):


```
docker run \  
  --name lgtm \  
  -p 3000:3000 \  
  -p 4317:4317 \  
  -p 4318:4318 \  
  --rm \  
  -ti \  
  -v $PWD/container/grafana:/data/grafana \  
  -v $PWD/container/prometheus:/data/prometheus \  
  -v $PWD/container/loki:/loki \  
  -e GF_PATHS_DATA=/data/grafana \  
  docker.io/grafana/otel-lgtm:latest
```

Automatic Tracing

Once enabled, MicroProfile Telemetry automatically instruments your Jakarta REST endpoints and client calls. This means your application immediately begins collecting distributed traces without any code changes. For instance, when a REST endpoint processes a request, you'll see spans that include important metadata, such as HTTP methods, status codes and request paths.

The automatic instrumentation follows OpenTelemetry semantic conventions, ensuring your traces are consistent with industry standards. You can configure how these spans are named through the `payara.telemetry.span-convention` property, with options ranging from OpenTelemetry's conventions to MicroProfile-specific naming patterns.

Manual Instrumentation

While automatic instrumentation covers many common scenarios, you'll often want to add custom instrumentation to capture business-specific operations. MicroProfile Telemetry integrates with Jakarta CDI, making this straightforward. Here's how you might instrument a business service:

```
@ApplicationScoped  
public class OrderService {  
    @Inject  
    Tracer tracer; // Inject the OpenTelemetry tracer  
  
    @WithSpan("process-order") // Create named spans easily  
    public void processOrder(@SpanAttribute("orderId") String orderId) {  
        // Method implementation  
    }  
}
```

```
}

public void complexOperation() {
    Span span = tracer.spanBuilder("complex-operation")
        .setSpanKind(SpanKind.INTERNAL)
        .startSpan();
    try (var scope = span.makeCurrent()) {
        // Operation logic
        span.setAttribute("business.value", "important-data");
    } finally {
        span.end();
    }
}
}
```

Advanced Configuration

As your observability needs to grow, MicroProfile Telemetry offers various configuration options to fine-tune your telemetry collection. Sampling is a core aspect of production deployments, letting you control how much telemetry data you collect. The default sampler (`parentbased_always_on`) captures all traces, but you can adjust this using the `otel.traces.sampler` property:

```
# Sample 10% of traces
otel.traces.sampler=traceidratio
otel.traces.sampler.arg=0.1
```

The above configuration tells the runtime to sample 10% of traces. Consult the spec for the full list of supported samplers.

Resource attributes help identify your service in the observability backend. You can set these through configuration:

```
otel.service.name=order-service
otel.resource.attributes=deployment.environment=production,team=commerce
```

This configuration assigns the name `order-service` to the instrumented service, ensuring that any telemetry data (like traces or metrics) is associated with that service. It also provides additional context via resource attributes, specifying that the service is running in the production environment

(`deployment.environment=production`) and is managed by the commerce team (`team=commerce`). These attributes help in organising, filtering and analysing telemetry data across different services or environments.

For performance optimization, you can configure how spans are batched and exported:

```
# Adjust batch export settings
otel.bsp.schedule.delay=5000
otel.bsp.max.queue.size=2048
otel.bsp.max.export.batch.size=512
otel.bsp.export.timeout=30000
```

This configuration adjusts the **batch span processor (bsp)** settings, which controls how telemetry data (spans, metrics) is buffered and exported. The `otel.bsp.schedule.delay=5000` sets the delay (in milliseconds) between export attempts to 5 seconds, ensuring regular batching of data. `otel.bsp.max.queue.size=2048` defines the maximum buffer size for spans, allowing up to 2048 spans to be queued before they are exported. `otel.bsp.max.export.batch.size=512` limits the number of spans sent in each batch to 512, promoting efficient data exports. Lastly, `otel.bsp.export.timeout=30000` sets the export timeout to 30 seconds, meaning that if the export process fails to complete within this time frame, it will be aborted, ensuring export operations remain timely.

Export configuration is important for production environments, especially when sending data to secure endpoints:

```
otel.exporter.otlp.endpoint=https://telemetry.production.com:4317
otel.exporter.otlp.protocol=grpc
otel.exporter.otlp.headers=api-key=secret,tenant=team1
```

The `otel.exporter.otlp.endpoint=https://telemetry.production.com:4317` specifies the target endpoint (URL and port) where the telemetry data will be sent. In this case, data are sent to a server at `telemetry.production.com` on port 4317. The `otel.exporter.otlp.protocol=grpc` sets the communication protocol to **gRPC**, a high-performance RPC (Remote Procedure Call) framework that is commonly used for efficient and reliable data exchange. The `otel.exporter.otlp.headers=api-key=secret,tenant=team1` sets custom headers, including an **API key** (`api-key=secret`) used for authentication and a **tenant identifier** (`tenant=team1`) to segregate data by team or project. These configurations ensure secure, structured, and authenticated telemetry data exports from applications.

Through these configuration options, MicroProfile Telemetry provides a flexible foundation for implementing observability in your Jakarta EE applications. Whether you're just starting with basic tracing or need sophisticated telemetry collection in production, the specification offers the tools needed to understand your distributed systems better.

Best Practices and Patterns

When implementing telemetry in Jakarta EE applications using MicroProfile Telemetry, following established best practices ensures your observability data are both useful and maintainable. These practices help create consistent, performant and secure telemetry implementation across your services.

Span Naming Conventions

Effective span naming is important for creating traces that tell a clear story about your system's behaviour. MicroProfile Telemetry provides several built-in span naming conventions through the `payara.telemetry.span-convention` property. For REST endpoints, you can choose conventions that best fit your observability strategy:

```
# OpenTelemetry semantic conventions (default)
GET /app/rest/resource/method

# Class and method based
GET:example.Resource.restMethod

# HTTP path based
GET:/resource/method
```

Choose a convention that aligns with how your team thinks about and debugs the application. For custom spans, create names that are both descriptive and consistent. Include the operation type and the entity being operated on, such as:

```
@WithSpan("order.validate") // Instead of just "validate"
@WithSpan("payment.process") // Instead of "processPayment"
```

Attribute Management

Attributes provide context to your traces, but collecting too many can impact performance and cost. Follow these guidelines for effective attribute management:

```
@ApplicationScoped
public class OrderService {
    @Inject
    Span span;

    public void processOrder(Order order) {
        // Add high-value business context
        span.setAttribute("order.id", order.getId());
        span.setAttribute("order.type", order.getType());

        // Conditionally add detailed attributes
        if (span.isRecording()) {
            // Add more detailed attributes only if the span is sampled
            span.setAttribute("order.items.count", order.getItems().
size());
            span.setAttribute("order.total", order.calculateTotal());
        }
    }
}
```

Error Handling

Proper error handling in traces helps quickly identify and debug issues. MicroProfile Telemetry integrates with Jakarta EE's exception handling mechanisms:

```
@ApplicationScoped
public class OrderProcessor {
    @WithSpan
    public void processOrder(String orderId) {
        try {
            // Processing logic
        } catch (OrderNotFoundException e) {
            Span.current()
                .setStatus(StatusCode.ERROR)
                .recordException(e)
                .setAttribute("error.type", "order_not_found");
            throw e; // Rethrow to maintain application flow
        }
    }
}
```

```
    }  
  }  
}
```

Performance Considerations

While telemetry provides valuable insights, it's important to minimise its performance impact. Consider these practices:

Use sampling judiciously in production environments. For instance, you might want to sample 10% of the available traces instead of everything:

```
# Sample based on parent trace decision  
otel.traces.sampler=parentbased_traceidratio  
otel.traces.sampler.arg=0.1
```

Configure appropriate batch settings for your environment:

```
# Adjust based on your application's traffic patterns  
otel.bsp.schedule.delay=5000  
otel.bsp.max.queue.size=2048
```

Check span recording status before computing expensive attributes:

```
if (span.isRecording()) {  
    span.setAttribute("computed.value", expensiveComputation());  
}
```

Security Considerations

Telemetry data often contains sensitive information. Implement these security practices:

Configure secure transport for telemetry data, at least in production:

```
# Use HTTPS for telemetry export
otel.exporter.otlp.endpoint=https://telemetry.secure.com:4317
otel.exporter.otlp.certificate=/path/to/cert.pem
```

Be mindful of sensitive data in span attributes:

```
public void processPayment(PaymentDetails payment) {
    // Only log non-sensitive payment information
    span.setAttribute("payment.type", payment.getType());
    span.setAttribute("payment.status", payment.getStatus());
    // Don't log card numbers, tokens, or other sensitive data
}
```

Use headers for authentication and tenant isolation:

```
otel.exporter.otlp.headers=authorization=Bearer ${TOKEN},tenant=${TENANT_ID}
```

Following these best practices, you can create a telemetry implementation that provides useful insights while maintaining security, performance and maintainability. Remember that telemetry should enhance your application's observability without compromising its core functionality or security.

Conclusions

Implementing effective observability in Jakarta EE applications is no longer optional in today's complex distributed systems – it's a must to maximizing seamless performance, identifying bottlenecks early, and maintaining system resilience. MicroProfile Telemetry provides a standards-based, flexible approach that bridges the gap between traditional enterprise Java applications and modern observability practices.

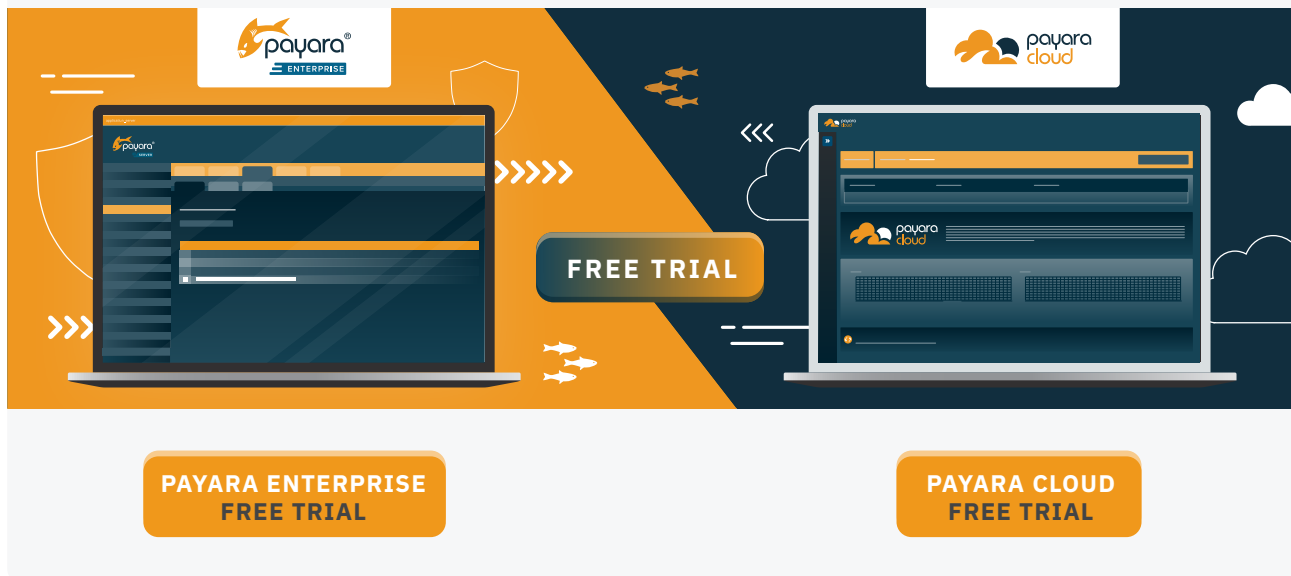
Throughout this guide, we explored how MicroProfile Telemetry offers multiple instrumentation approaches—automatic, manual and agent-based—each serving different needs and use cases. We looked at how it seamlessly integrates with the Jakarta EE programming model while leveraging the power and standardization of OpenTelemetry.

Key takeaways from this guide include:

1. Start with automatic instrumentation for quick wins, then gradually add manual instrumentation for business-specific insights
2. Follow established naming conventions and attribute management practices to maintain clarity and consistency
3. Configure appropriate sampling and batching settings to balance observability needs with performance
4. Implement proper security measures when handling and transmitting telemetry data
5. Use resource attributes effectively to provide context for your services

Remember that observability is a journey, not a destination. Begin with basic tracing and gradually expand your telemetry implementation as your needs grow. By following the practices and patterns outlined in this guide, you'll be well-equipped to build and maintain observable Jakarta EE applications that are easier to debug, monitor and maintain in production environments.

Interested in Payara? *Try Before You Buy*



The banner features two laptops. The left laptop displays the Payara Enterprise interface, and the right laptop displays the Payara Cloud interface. Above the left laptop is the 'payara ENTERPRISE' logo, and above the right laptop is the 'payara cloud' logo. A central orange button with white text reads 'FREE TRIAL'. Below the banner, there are two orange buttons: 'PAYARA ENTERPRISE FREE TRIAL' on the left and 'PAYARA CLOUD FREE TRIAL' on the right. The background is a dark blue gradient with orange fish icons and white arrows indicating a flow from left to right.



sales@payara.fish



UK: +44 800 538 5490
Intl: +1 888 239 8941



www.payara.fish

Payara Services Ltd 2024 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ