MicroProfile Telemetry Tracing

MicroProfile Telemetry Team (Roberto Cortez, Emily Jiang, Bruno Baptista, Jan Westerkamp, Felix Wong, Yasmin Aumeeruddy)

1.0, November 09, 2022: Final

Table of Contents

```
Specification: MicroProfile Telemetry Tracing

Version: 1.0

Status: Final

Release: November 09, 2022
```

Copyright

Eclipse Foundation Specification License

Disclaimers

 :sectnums:

Introduction

In cloud-native technology stacks, distributed and polyglot architectures are the norm. Distributed architectures introduce a variety of operational challenges including how to solve availability and performance issues quickly. These challenges have led to the rise of observability.

Telemetry data is needed to power observability products. Traditionally, telemetry data has been provided by either open-source projects or commercial vendors. With a lack of standardization, the net result is the lack of data portability and the burden on the user to maintain the instrumentation.

The OpenTelemetry project solves these problems by providing a single, vendor-agnostic solution.

Architecture

OpenTelemetry is a set of APIs, SDKs, tooling and integrations that are designed for the creation and management of telemetry data such as traces, metrics, and logs.

This specification defines the behaviors that allow MicroProfile applications to easily participate in an environment where distributed tracing is enabled via OpenTelemetry (a merger between OpenTracing and OpenCensus).

This document and implementations **MUST** comply with the following OpenTelemetry 1.13 specifications:

- OpenTelemetry Overview
- Tracing API
- Baggage API
- Context API
- Resource SDK

IMPORTANT	The Metrics and Logging integrations of OpenTelemetry are out of scope of this specification. Implementations are free to provide support for both Metrics and Logging if desired.

This specification supports the following three types of instrumentation:

- Automatic Instrumentation
- Manual Instrumentation
- Agent Instrumentation

Automatic Instrumentation

Jakarta RESTful Web Services (server and client), and MicroProfile REST Clients are automatically enlisted to participate in distributed tracing without code modification as specified in the Tracing API.

These should follow the rules specified in the Semantic Conventions section.

Manual Instrumentation

Explicit manual instrumentation can be added into a MicroProfile application in the following ways:

@WithSpan

Annotating a method in any Jakarta CDI aware beans with the `io.opentelemetry.instrumentation.annotations.WithSpan` annotation. This will create a new Span

and establish any required relationships with the current Trace context.

Method parameters can be annotated with the `io.opentelemetry.instrumentation.annotations.SpanAttribute` annotation to indicate which method parameters should be part of the Trace.

Example:

```
@ApplicationScoped
class SpanBean {
    @WithSpan
    void span() {

    }

    @WithSpan("name")
    void spanName() {

    }

    @WithSpan(kind = SpanKind.SERVER)
    void spanKind() {

    }

    @WithSpan
    void spanArgs(@SpanAttribute(value = "arg") String arg) {

    }
}
```

Obtain a SpanBuilder

By obtaining a `SpanBuilder` from the current `Tracer` and calling `io.opentelemetry.api.trace.Tracer.spanBuilder(String)`. In this case, it is the developer's responsibility to ensure that the `Span` is properly created, closed, and propagated.

Example:

```
@RequestScoped
@Path("/")
public class SpanResource {
    @Inject
    Tracer tracer;

    @GET
    @Path("/span/new")
    public Response spanNew() {
        Span span = tracer.spanBuilder("span.new")
                .setSpanKind(SpanKind.INTERNAL)
                .setParent(Context.current().with(this.span))
                .setAttribute("my.attribute", "value")
                .startSpan();

        span.end();

        return Response.ok().build();
    }
}
```

NOTE	Start and end a new `Span` will add a child `Span` to the current one enlisted by the automatic instrumentation of Jakarta REST applications.

Obtain the current Span

By obtaining the current `Span` to add attributes. The Span lifecycle is managed by the implementation.

Example:

```
@RequestScoped
@Path("/")
public class SpanResource {
    @GET
    @Path("/span/current")
    public Response spanCurrent() {
        Span span = Span.current();
        span.setAttribute("my.attribute", "value");
        return Response.ok().build();
    }
}
```

Or with CDI:

```
@RequestScoped
@Path("/")
public class SpanResource {
    @Inject
    Span span;

    @GET
    @Path("/span/current")
    public Response spanCurrent() {
        span.setAttribute("my.attribute", "value");
        return Response.ok().build();
    }
}
```

Agent Instrumentation

Implementations are free to support the OpenTelemetry Agent Instrumentation. This provides the ability to gather telemetry data without code modifications by attaching a Java Agent JAR to the running JVM.

If an implementation of MicroProfile Telemetry Tracing provides such support, it must conform to the instructions detailed in the OpenTelemetry Java Instrumentation project, including:

- Agent Configuration
- Suppressing Instrumentation

Both Agent and MicroProfile Telemetry Tracing Instrumentation (if any), must coexist with each other.

Access to the OpenTelemetry Tracing API

An implementation of MicroProfile Telemetry Tracing must provide the following CDI beans for supporting contextual instance injection:

- `io.opentelemetry.api.OpenTelemetry`
- `io.opentelemetry.api.trace.Tracer`
- `io.opentelemetry.api.trace.Span`
- `io.opentelemetry.api.baggage.Baggage`

Calling the OpenTelemetry API directly must work in the same way and yield the same results:

- `io.opentelemetry.api.trace.Span.current()`
- `io.opentelemetry.api.baggage.Baggage.current()`

Implementations MAY support

- `io.opentelemetry.api.GlobalOpenTelemetry.get()`

To obtain the `Tracer` with the OpenTelemetry API, the consumer must use the exact same instrumentation name and version used by the implementation. Failure to do so, may result in a different `Tracer` and incorrect handling of the OpenTelemetry data.

Configuration

OpenTelemetry must be configured by MicroProfile Config following the configuration properties detailed in:

- OpenTelemetry SDK Autoconfigure (excluding properties related to Metrics and Logging)
- Manual Instrumentation

An implementation may opt to not support a subset of configuration properties related to a specific configuration. For instance, `otel.traces.exporter` is required but if the implementation does not support `jaeger` as a valid exporter, then all configuration properties referring to `otel.tracer.jaeger.*` are not required.

Semantic Conventions

The Trace Semantic Conventions for Spans and Attributes must be followed by any compatible implementation.

All attributes marked as `required` must be present in the context of the Span where they are defined. Any other attribute is optional. Implementations can also add their own attributes.

MicroProfile Attributes

Other MicroProfile specifications can add their own attributes under their own attribute name following the convention `mp.[specification short name].[attribute name]`.

Implementation libraries can set the library name using the following property:

`mp.telemetry.tracing.name`

Tracing Enablement

By default, MicroProfile Telemetry Tracing is deactivated.

In order to enable any of the tracing aspects, the configuration `otel.sdk.disabled=false` must be specified in any of the configuration sources available via MicroProfile Config.

IMPORTANT

This is a deviation from the OpenTelemetry Specification version 1.14.0 that specifies this configuration property officially, where OpenTelemetry is activated by default!

But in fact, it will be activated only by adding it's dependency to the application or platform project. To be able to add MicroProfile Telemetry Tracing to MicroProfile implementations by default without side effects, this deviating behaviour has been defined here (see also MicroProfile Telemetry and MicroProfile OpenTracing).

The original definition for this configuration property and the corresponding `OTEL_SDK_DISABLED` environment variable is specified in the OpenTelemetry Environment Variable Specification version 1.14.0 and it's General SDK Configuration.

This property is read once when the application is starting. Any changes afterwards will not take effect unless the application is restarted.

MicroProfile OpenTracing

MicroProfile Telemetry Tracing supersedes MicroProfile OpenTracing. Even if the end goal is the same, there are some considerable differences:

- Different API (between OpenTracing and OpenTelemetry)
- No `@Traced` annotation
- No specific MicroProfile configuration
- No customization of Span name through MicroProfile API
- Differences in attribute names and mandatory ones

For these reasons, the MicroProfile Telemetry Tracing specification does not provide any migration path between both projects. While it is certainly possible to achieve a migration path at the code level and at the specification level (at the expense of not following the main OpenTelemetry specification), it is unlikely to be able to achieve the same compatibility at the data layer. Regardless, implementations are still free to provide migration paths between MicroProfile OpenTracing and MicroProfile Telemetry Tracing.

If a migration path is provided, the bridge layer provided by OpenTelemetry should be used. This bridge layer implements OpenTracing APIs using OpenTelemetry APIs (more details can be found from OpenTracing Compatbility. The bridge layer takes OpenTelemetry Tracer and exposes as OpenTracing Tracer. See the example below.

```
//From the global OpenTelemetry configuration
Tracer tracer1 = OpenTracingShim.createTracerShim();
//From a provided OpenTelemetry instance oTel
Tracer tracer2 = OpenTracingShim.createTracerShim(oTel);
```

Afterwards, you can then register the tracer as the OpenTracing Global Tracer:

```
GlobalTracer.registerIfAbsent(tracer);
```

MicroProfile Telemetry and MicroProfile OpenTracing

If MicroProfile Telemetry and MicroProfile OpenTracing are both present in one application, it is advised only to enable one of them. Otherwise, no portable behaviour may occur.