MicroProfile Telemetry Tracing

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

1.1-RC1, August 11, 2023: Draft

Table of Contents

Specification: MicroProfile Telemetry Tracing

Version: 1.1-RC1

Status: Draft

Release: August 11, 2023

Copyright

Eclipse Foundation Specification License

Disclaimers

DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders. :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).

The OpenTelemetry specification describes the cross-language requirements and expectations for all OpenTelemetry implementations. This specification is based on the Java implementation v1.28.0 of OpenTelemetry. An implementation of this MicroProfile Telemetry may consume a later patch release of the Java implementation as long as the required TCKs pass successfully.

Refer to the OpenTelemetry specification repo to understand some essential terms.

- 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.

Supported OpenTelemetry API Classes

Classes from the following API packages MUST be supported by implementations of this

specification, though this specification does not prevent additional API classes from being supported. Implementations are allowed to pull in a more recent patch version of the API classes.

Tracing APIs

- io.opentelemetry.api (except `GlobalOpenTelemetry`)
- io.opentelemetry.api.trace
- io.opentelemetry.api.common

NOTE	`io.opentelemetry.api` does depend on io.opentelemetry.api.metrics and io.opentelemetry.api.logs, though this specification only supports the parts that relate to Tracing.

Baggage APIs

- io.opentelemetry.api.baggage
- io.opentelemetry.api.baggage.propagation

Context API

- io.opentelemetry.context
- io.opentelemetry.context.propagation

Resource SDK

- io.opentelemetry.sdk.resources

Autoconfigure SPI

This is the programmatic interface that allows users to register extensions when using the SDK Autoconfigure Extension (which we use for configuration).

- io.opentelemetry.sdk.autoconfigure.spi
- io.opentelemetry.sdk.autoconfigure.spi.traces

The above packages have dependencies on the following packages which must be supported to the extent that they are required by the Autoconfigure SPI classes:

- io.opentelemetry.sdk.trace
- io.opentelemetry.sdk.trace.data
- io.opentelemetry.sdk.trace.export
- io.opentelemetry.sdk.trace.samplers
- io.opentelemetry.sdk.common

Annotations

- io.opentelemetry.instrumentation.annotations (`WithSpan` and `SpanAttribute` only)

Semantic Conventions

NOTE	These packages are not stable and may be subject to breaking changes in future releases.

- io.opentelemetry.semconv.trace.attributes
- io.opentelemetry.semconv.resource.attributes

Configuration

OpenTelemetry must be configured by MicroProfile Config following the semantics of configuration properties detailed in OpenTelemetry SDK Autoconfigure 1.28.0.

At minimum the following MicroProfile Config properties MUST be supported:

Property Name	Description
Global Configuration	
`otel.sdk.disabled`	Set to `false` to enable OpenTelemetry. Default value: `true`
Exporters configuration	
`otel.traces.exporter`	List of exporters to be used for tracing, separated by commas. `none` means no autoconfigured exporter. Values other than `none` and `otlp` might require additional libraries Default value: `otlp`
`otel.propagators`	The propagators to be used. Values other than `none`, `tracecontext` and `baggage` might require additional libraries Default value: `tracecontext, baggage`
Resource attributes	
`otel.resource.attributes`	Specify resource attributes in the following format: `key1=val1, key2=val2, key3=val3`
`otel.service.name`	Specify logical service name. Takes precedence over `service.name` defined with `otel.resource.attributes` Default value: application name (if applicable)

Property Name	Description
Batch Span Processor	
`otel.bsp.schedule.delay`	The interval, in milliseconds, between two consecutive exports. Default value: `5000`
`otel.bsp.max.queue.size`	The maximum queue size. Default value: `2048`
`otel.bsp.max.export.batch.size`	The maximum batch size. Default value: `512`
`otel.bsp.export.timeout`	The maximum allowed time, in milliseconds, to export data. Default value: `30000`
Sampler	
`otel.traces.sampler`	The sampler to use for tracing. Supported values are: • `always_on` • `always_off` • `traceidratio` • `parentbased_always_on` • `parentbased_always_off` • `parentbased_traceidratio` Support for other samplers might be added with additional libraries Default value: `parentbased_always_on`
`otel.traces.sampler.arg`	An argument to the configured tracer if supported, for example a ratio. Consult OpenTelemetry documentation for details.
OTLP Exporter	
`otel.exporter.otlp.protocol`	The transport protocol to use on OTLP trace, metric, and log requests. Options include grpc and http/protobuf. Default value: `grpc`

Property Name	Description
`otel.exporter.otlp.traces.protocol`	The transport protocol to use on OTLP trace requests. Options include grpc and http/protobuf. Default value: `grpc`
`otel.exporter.otlp.endpoint`	The OTLP traces, metrics, and logs endpoint to connect to. Must be a URL with a scheme of either http or https based on the use of TLS. If protocol is http/protobuf the version and signal will be appended to the path (e.g. v1/traces, v1/metrics, or v1/logs) Default value: `http://localhost:4317` when protocol is `grpc`, `http://localhost:4318/v1/{signal}` when protocol is `http/protobuf`
`otel.exporter.otlp.traces.endpoint`	The OTLP traces endpoint to connect to. Must be a URL with a scheme of either http or https based on the use of TLS. Default value: `http://localhost:4317` when protocol is `grpc`, and `http://localhost:4318/v1/traces` when protocol is `http/protobuf`
`otel.exporter.otlp.certificate`	The path to the file containing trusted certificates to use when verifying an OTLP trace, metric, or log server's TLS credentials. The file should contain one or more X.509 certificates in PEM format. By default the host platform's trusted root certificates are used.
`otel.exporter.otlp.traces.certificate`	The path to the file containing trusted certificates to use when verifying an OTLP trace server's TLS credentials. The file should contain one or more X.509 certificates in PEM format. By default the host platform's trusted root certificates are used.
`otel.exporter.otlp.client.key`	The path to the file containing private client key to use when verifying an OTLP trace, metric, or log client's TLS credentials. The file should contain one private key PKCS8 PEM format. By default no client key is used.

Property Name	Description
`otel.exporter.otlp.traces.client.key`	The path to the file containing private client key to use when verifying an OTLP trace client's TLS credentials. The file should contain one private key PKCS8 PEM format. By default no client key file is used.
`otel.exporter.otlp.client.certificate`	The path to the file containing trusted certificates to use when verifying an OTLP trace, metric, or log client's TLS credentials. The file should contain one or more X.509 certificates in PEM format. By default no chain file is used.
`otel.exporter.otlp.traces.client.certificate`	The path to the file containing trusted certificates to use when verifying an OTLP trace server's TLS credentials. The file should contain one or more X.509 certificates in PEM format. By default no chain file is used.
`otel.exporter.otlp.headers`	Key-value pairs separated by commas to pass as request headers on OTLP trace, metric, and log requests.
`otel.exporter.otlp.traces.headers`	Key-value pairs separated by commas to pass as request headers on OTLP trace requests.
`otel.exporter.otlp.compression`	The compression type to use on OTLP trace, metric, and log requests. Options include gzip. By default no compression will be used.
`otel.exporter.otlp.traces.compression`	The compression type to use on OTLP trace requests. Options include gzip. By default no compression will be used.
`otel.exporter.otlp.timeout`	The maximum waiting time, in milliseconds, allowed to send each OTLP trace, metric, and log batch. Default value: `10000`
`otel.exporter.otlp.traces.timeout`	The maximum waiting time, in milliseconds, allowed to send each OTLP trace batch. Default value: `10000`

If Environment Config Source is enabled for MicroProfile Config, then the environment variables as described by the OpenTelemetry SDK Autoconfigure are also supported.

Service Loader Support

Implementation will load additional configuration related components by means of service loader. This allows the application to define its own metadata and trace handling behavior. The following

components are supported

Component interface	Purpose
`ConfigurablePropagatorProvider`	Provides implementation for a name referred in `otel.propagators`
`ConfigurableSpanExporterProvider`	Provides implementation for a name referred in `otel.traces.exporter`
`ConfigurableSamplerProvider`	Provides implementation for a name referred in `otel.traces.sampler`
`AutoConfigurationCustomizerProvider`	Customizes configuration properties before they are applied to the SDK
`ResourceProvider`	Defines resource attributes describing the application

Behavior when multiple implementations are found for a given component name is undefined. Behavior when customizer changes other properties than those listed in the spec is also undefined.

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 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).

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 API. 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 recommended to only enable one of them, otherwise non-portable behaviour may occur.

Release Notes

This section documents the changes introduced by individual releases.

Release Notes for MicroProfile Telemetry 1.1

A full list of changes delivered in the 1.1 release can be found at MicroProfile Telemetry 1.1 Milestone.

Incompatible Changes

no

API/SPI Changes

Consume the OpenTelemetry Java release v1.28.0. The full comparison with the v1.19.0 supported by MicroProfile Telemetry 1.0 can be found here.

Other Changes

- Consume the lastest OpenTelemetry Tracing (88)
- Clarify which API classes must be available to users (91)
- Clarify the behaviour of Span and Baggage beans when the current span or baggage changes (90)
- TCK: Implement tests in a way that is not timestamp dependent (44)
- TCK: TCK RestClientSpanTest Span Name Doesn't Follow Semantic Conv (86)
- TCK: Adding missing TCKs (89)
- TCK: TCK cannot be run using the Arquillian REST protocol (72)