# MicroProfile Telemetry

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

2.0-RC1, June 27, 2024: Draft

# Table of Contents

```
Specification: MicroProfile Telemetry
Version: 2.0-RC1
Status: Draft
Release: June 27, 2024
```

# Copyright



# Eclipse Foundation Specification License - v1.1



## Disclaimers

FOUNDATION AISBL WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation AISBL 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.

The OpenTelemetry specification describes the cross-language requirements and expectations for all OpenTelemetry implementations. This specification is based on the Java implementation v1.39.0 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 Logging integrations of OpenTelemetry are out of scope of this specification. Implementations are free to provide support for Logging if desired. |

# SDK integration

Implementations SHALL provide integration by appropriately configuring one or more compatible OpenTelemetry SDK instances for the application runtime. Regardless of signal type being considered — traces, logs or metrics — the following requirements SHALL be met.

## Enabling OpenTelemetry support

By default, MicroProfile Telemetry is deactivated.

In order to enable any aspects of integration, the configuration `otel.sdk.disabled=false` MUST be specified.

If the `otel.sdk.disabled=false` configuration setting is visible to the runtime at initialization time then the runtime MUST provide a SINGLE OpenTelemetry SDK instance which MUST be used by the runtime and all applications.

If the `otel.sdk.disabled=false` configuration setting is only visible to the application(s) at initialization time, then runtime telemetry MUST be disabled and the application(s) MUST be configured using the visible `otel.*` properties.

Implementations that do not support use of MicroProfile Config during runtime initialization may use the `OTEL_SDK_DISABLED` environment variable or `otel.sdk.disabled` Java system property to specify this setting.

At application initialization time runtimes MUST use configuration sources available via MicroProfile Config for configuration.

Runtimes MAY provide additional means of configuring per-application SDKs or runtime extensions.

**NOTE**

This is a breaking change for runtimes that can simultaneously run multiple applications. In MicroProfile Telemetry 1.1 the `OTEL_SDK_DISABLED` environment variable could be used to indicate whether to enable or disable the application instance(s) of OpenTelemetry. Users that want to enable/disable individual application instances can do so using any MicroProfile Config configuration source that is only visible to applications. Users that want to enable an OpenTelemetry instance to be used by the runtime and all applications need to provide any OpenTelemetry configuration settings in a way that is visible to the runtime at initialization time (for example, using environment variables).

**IMPORTANT**

This is a deviation from the OpenTelemetry Specification that specifies the `otel.sdk.disabled` configuration property officially, where OpenTelemetry is activated by default!

But in fact, it will be activated only by adding its dependency to the application or platform project. To be able to add MicroProfile Telemetry 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.

# Configuration

OpenTelemetry MUST be configured by MicroProfile Config following the semantics of configuration properties of OpenTelemetry SDK Autoconfigure extension.

Full list of required configuration property names are listed in Configuration.

### OTLP support

OpenTelemetry data can be exported in various ways. Implementation MUST support exporting data via OTLP protocol and relevant configuration properties for OTLP exporter.

### Service Providers support

Additional OpenTelemetry SDK components can be integrated by means of Java Service Loader mechanism.

Full list of supported service providers is listed in Service Loader Support.

# Access to OpenTelemetry API

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

- `io.opentelemetry.api.OpenTelemetry`

Implementations MAY support:

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

To obtain the access to `OpenTelemetry` instance. The consumer MUST use the exact same instrumentation name and version used by the implementation. Failure to do so, MAY result in a different tracing and metrics components to be used.

Later sections provide more beans for particular signal types.

### API classes

In order to provide integration with OpenTelemetry the implementations SHALL make a number of OpenTelemetry packages available to applications. The full list of packages is listed in Supported OpenTelemetry API Classes.

# Tracing

In the observability, Tracing is used to diagnose problems. Tracing instrumentation is used to generate traces.

# Tracing Instrumentation

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 Trace 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.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()`

# Trace Semantic Conventions

The Semantic Conventions for HTTP Spans MUST be followed by any compatible implementation.

| | |
|---|---|
| **NOTE** | This is a breaking change from MicroProfile Telemetry 1.1 due to stabilization of HTTP semantic conventions in OpenTelemetry. Changes to attributes are described in HTTP semantic convention stability migration guide. |

Semantic Conventions distinguish several Requirement Levels for attributes. All Span attributes marked as `Required` and `Conditionally Required` MUST be present in the context of the Span where they are defined. Any other attribute is optional. Implementations MAY also add their own attributes, or provide means of configuring `Opt-In` attribute emission.

## 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`

# Routing Traces

OpenTelemetry can be enabled selectively for each application, or globally for the runtime and all applications as described in Enabling OpenTelemetry support. Traces and spans may be emitted by applications or on behalf of a component in the runtime. For example, spans created by an app to track the execution of a database call are application spans, whereas spans created to track the execution of a call to the runtime's `/health` endpoint are runtime spans.

For spans that originate from an application:

- if the OpenTelemetry SDK instance is shared by the runtime and applications then application spans should be routed to this instance

- if an OpenTelemetry SDK instance is enabled for the application that is creating spans then spans from that application should be routed to this instance
- if no OpenTelemetry SDK instance is enabled for the application that is creating spans then spans from that application should be discarded (typically by sending the request to a noop OpenTelemetry SDK instance)

For spans that originate from the runtime:

- if the OpenTelemetry SDK instance is shared by the runtime and applications then runtime spans should be routed to this instance
- if no OpenTelemetry SDK instance is shared by the runtime and applications then spans from the runtime should be discarded (typically by sending the request to a noop OpenTelemetry SDK instance)

# Tracing Enablement

Tracing is activated whenever Microprofile Telemetry is enabled, as described in Enabling OpenTelemetry support.

# 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.

# Metrics

Metrics are captured measurements of applications' and runtime's behavior. An application may provide metrics of its own in addition to the metrics provided by the runtime.

Implementations are required to capture certain required metrics such as JVM performance counters and HTTP request processing times. Custom metrics can be defined by utilizing Metrics API as following example demostrates:

```
class WithCounter {
    @Inject
    Meter meter;

    private LongCounter counter;

    @PostConstruct
    public void init() {
        counter = meter
                    .counterBuilder("new_subscriptions")
                    .setDescription("Number of new subscriptions")
                    .setUnit("1")
                    .build();
    }

    void subscribe(String plan) {
        counter.add(1,
            Attributes.of(AttributeKey.stringKey("plan"), plan));
    }
}
```

In this example `Meter` is used to define an instrument, in this case a Counter and application code then can record measurement values along with additional attributes. Measurement aggregations are computed separately for each unique combination of attributes.

## Routing Metrics

OpenTelemetry can be enabled selectively for each application, or globally for the runtime and all applications as described in Enabling OpenTelemetry support. Metrics may be registered by applications or on behalf of a component in the runtime. For example, a counter metric that is registered by an application to track the number of cars driving over a bridge is an application metric, whereas a gauge tracking the amount of memory used by the JVM is a runtime metric.

For metrics that are registered by an application:

- if the OpenTelemetry SDK instance is shared by the runtime and applications then application-registered metrics should be routed to this instance
- if an OpenTelemetry SDK instance is enabled for the application that is registering a metric then

that metric should be routed to this instance

- if no OpenTelemetry SDK instance is enabled for the application that is registering a metric then that metric should be discarded (typically by sending the registration request to a noop OpenTelemetry SDK instance)

For metrics that originate from the runtime:

- if the OpenTelemetry SDK instance is shared by the runtime and applications then runtime-registered metrics should be routed to this instance
- if no OpenTelemetry SDK instance is shared by the runtime and applications then runtime-registered metrics from the runtime should be discarded (typically by sending the registration request to a noop OpenTelemetry SDK instance)

# Access to the OpenTelemetry Metrics API

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

- `io.opentelemetry.api.metrics.Meter`

# Required Metrics

The following metrics MUST be provided by runtimes. These are as defined in the OpenTelemetry Semantic Conventions v1.26.0

All attributes that are listed as required and stable in the OpenTelemetry Semantic Conventions MUST be included.

All attributes that are listed as conditionally required and stable in the OpenTelemetry Semantic Conventions MUST be included when the condition described in the OpenTelemetry Semantic Conventions is satisfied.

All attributes that are listed as recommended and stable in the OpenTelemetry Semantic Conventions SHOULD be included if they are readily available and can be efficiently populated.

All attributes that are listed as opt-in and stable in the OpenTelemetry Semantic Conventions MUST NOT be included unless the implementation provides a means for users to configure which opt-in attributes to enable. This requirement is based on OpenTelemetry Semantic Conventions documentation indicating that opt-in attributes MUST NOT be included unless the user has a way to choose if they are enabled/disabled.

Attribute values and usage guidelines as defined in the semantic conventions document MUST be followed.

| Metric Name | Type | Attributes |
|---|---|---|
| **HTTP Server** | | |

| Metric Name | Type | Attributes |
| --- | --- | --- |
| `http.server.request.duration` | Histogram | required attributes<br>• `http.request.method`<br>• `url.scheme`<br>conditionally required<br>• `error.type`<br>• `http.response.status_code`<br>• `http.route`<br>• `network.protocol.name`<br>recommended<br>• `network.protocol.version`<br>opt-in<br>• `server.address`<br>• `server.port` |
| **JVM Memory** | | |
| `jvm.memory.used` | UpDownCounter | recommended<br>• `jvm.memory.pool.name`<br>• `jvm.memory.type` |
| `jvm.memory.committed` | UpDownCounter | recommended<br>• `jvm.memory.pool.name`<br>• `jvm.memory.type` |
| `jvm.memory.limit` | UpDownCounter | recommended<br>• `jvm.memory.pool.name`<br>• `jvm.memory.type` |
| `jvm.memory.used_after_last_gc` | UpDownCounter | recommended<br>• `jvm.memory.pool.name`<br>• `jvm.memory.type` |
| **JVM Garbage Collection** | | |
| `jvm.gc.duration` | Histogram | recommended<br>• `jvm.gc.action`<br>• `jvm.gc.name` |

| Metric Name | Type | Attributes |
|---|---|---|
| **JVM Threads** | | |
| `jvm.thread.count` | UpDownCounter | recommended<br>• `jvm.thread.daemon`<br>• `jvm.thread.state` |
| **JVM Classes** | | |
| `jvm.class.loaded` | Counter | |
| `jvm.class.unloaded` | Counter | |
| `jvm.class.count` | UpDownCounter | |
| **JVM CPU** | | |
| `jvm.cpu.time` | Counter | |
| `jvm.cpu.count` | UpDownCounter | |
| `jvm.cpu.recent_utilization` | Gauge | |

# Metrics Enablement

Metrics are activated whenever Microprofile Telemetry is enabled, as described in Enabling OpenTelemetry support.

# Logs

The OpenTelemetry Logs bridge API exists to enable bridging logs from other log frameworks (e.g. SLF4J, Log4j, JUL, Logback, etc) into OpenTelemetry. It does not define new Log APIs and the Logs bridge APIs in OpenTelemetry are not for application but for runtime to bridge log frameworks. Therefore, this specification does not expose any Log APIs.

## Routing Logs

OpenTelemetry can be enabled selectively for each application, or globally for the runtime and all applications as described in Enabling OpenTelemetry support. Logs may be emitted by applications or on behalf of a component in the runtime. For example, logs written from a RESTful web service that is part of a banking application are application logs, whereas logs written from the kernel of a runtime before any application has started are runtime logs.

For logs that originate from an application:

- if the OpenTelemetry SDK instance is shared by the runtime and applications then application logs should be routed to this instance
- if an OpenTelemetry SDK instance is enabled for the application that is logging then logs from that application should be routed to this instance
- if no OpenTelemetry SDK instance is enabled for the application that is logging then logs from that application should be discarded (typically by sending the logging request to a noop OpenTelemetry SDK instance)

For logs that originate from the runtime:

- if the OpenTelemetry SDK instance is shared by the runtime and applications then runtime logs should be routed to this instance
- if no OpenTelemetry SDK instance is shared by the runtime and applications then logs from the runtime should be discarded (typically by sending the logging request to a noop OpenTelemetry SDK instance)

## Logs Enablement

Logging is activated whenever Microprofile Telemetry is enabled, as described in Enabling OpenTelemetry support.

# Configuration

OpenTelemetry MUST be configured by MicroProfile Config following the semantics of configuration properties detailed in OpenTelemetry SDK Autoconfigure 1.39.0. Following properties MUST be supported:

## Required Configuration Properties

| Property Name | Description |
|---|---|
| **Global Configuration** | |
| `otel.sdk.disabled` | Set to `false` to enable OpenTelemetry.<br><br>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`, `otlp` or `console` might require additional libraries. Implementations of the `otlp` and `console` exporters MUST be from the OpenTelemetry SDK.<br><br>Default value: `otlp` |
| `otel.metrics.exporter` | List of exporters to be used for metrics, separated by commas. `none` means no autoconfigured exporter. Values other than `none`, `otlp` or `console` might require additional libraries. Implementations of the `otlp` and `console` exporters MUST be from the OpenTelemetry SDK.<br><br>Default value: `otlp` |
| `otel.logs.exporter` | List of exporters to be used for logs, separated by commas. `none` means no autoconfigured exporter. Values other than `none`, `otlp` or `console` might require additional libraries. Implementations of the `otlp` and `console` exporters MUST be from the OpenTelemetry SDK.<br><br>Default value: `otlp` |

| Property Name | Description |
| --- | --- |
| `otel.propagators` | The propagators to be used. Values other than `none`, `tracecontext` and `baggage` might require additional libraries<br><br>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`<br><br>Default value: application name (if applicable) |
| **Batch Span Processor** | |
| `otel.bsp.schedule.delay` | The interval, in milliseconds, between two consecutive exports.<br><br>Default value: `5000` |
| `otel.bsp.max.queue.size` | The maximum queue size.<br><br>Default value: `2048` |
| `otel.bsp.max.export.batch.size` | The maximum batch size.<br><br>Default value: `512` |
| `otel.bsp.export.timeout` | The maximum allowed time, in milliseconds, to export data.<br><br>Default value: `30000` |
| **Sampler** | |

| Property Name | Description |
| --- | --- |
| `otel.traces.sampler` | The sampler to use for tracing. Supported values are:<br>• `always_on`<br>• `always_off`<br>• `traceidratio`<br>• `parentbased_always_on`<br>• `parentbased_always_off`<br>• `parentbased_traceidratio`<br>Support for other samplers might be added with additional libraries<br>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.<br>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)<br>Default value: `http://localhost:4317` when protocol is `grpc`, `http://localhost:4318/v1/{signal}` 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.<br>By default the host platform's trusted root certificates are used. |

| Property Name | Description |
| --- | --- |
| 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.<br><br>By default no client key 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.headers | Key-value pairs separated by commas to pass as request headers on OTLP trace, metric, and log requests. |
| otel.exporter.otlp.compression | The compression type to use on OTLP trace, metric, and log requests. Options include gzip.<br><br>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.<br><br>Default value: 10000 |
| otel.exporter.otlp.metrics.temporality.preference | The preferred output aggregation temporality.<br><br>• CUMULATIVE: all instruments will have cumulative temporality.<br>• DELTA: counter (sync and async) and histograms will be delta, up down counters (sync and async) will be cumulative.<br>• LOWMEMORY: sync counter and histograms will be delta, async counter and up down counters (sync and async) will be cumulative.<br><br>Default value: CUMULATIVE. |
| otel.exporter.otlp.metrics.default.histogram.aggregation | The preferred default histogram aggregation. Options include BASE2_EXPONENTIAL_BUCKET_HISTOGRAM and EXPLICIT_BUCKET_HISTOGRAM.<br><br>Default value: EXPLICIT_BUCKET_HISTOGRAM. |

| Property Name | Description |
|---|---|
| `otel.metrics.exemplar.filter` | The filter for exemplar sampling. Can be `ALWAYS_OFF`, `ALWAYS_ON` or `TRACE_BASED`.<br><br>Default value: `TRACE_BASED` |
| `otel.metric.export.interval` | The interval, in milliseconds, between the start of two export attempts. |
| **Batch log record processor** | |
| `otel.blrp.schedule.delay` | The interval, in milliseconds, between two consecutive exports.<br><br>Default value: `1000` |
| `otel.blrp.max.queue.size` | The maximum batch size.<br><br>Default value: `512` |
| `otel.blrp.max.export.batch.size` | The maximum queue size.<br><br>Default value: `2048` |
| `otel.blrp.export.timeout` | The maximum allowed time, in milliseconds, to export data.<br><br>Default value: `30000` |

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

# Optional Configuration Properties

An implementation MAY support additional configuration properties. Notable examples include per-signal configuration of exporters:

| Property Name | Description |
|---|---|
| **OTLP Exporter** | |
| `otel.exporter.otlp.traces.protocol` | The transport protocol to use on OTLP trace requests. Options include grpc and http/protobuf.<br><br>Default value: `grpc` |
| `otel.exporter.otlp.metrics.protocol` | The transport protocol to use on OTLP metric requests. Options include grpc and http/protobuf.<br><br>Default value: `grpc` |

| Property Name | Description |
| --- | --- |
| `otel.exporter.otlp.logs.protocol` | The transport protocol to use on OTLP log requests. Options include grpc and http/protobuf.<br><br>Default value: `grpc` |
| `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.<br><br>Default value: `http://localhost:4317` when protocol is `grpc`, and `http://localhost:4318/v1/traces` when protocol is `http/protobuf` |
| `otel.exporter.otlp.metrics.endpoint` | The OTLP metrics endpoint to connect to. MUST be a URL with a scheme of either http or https based on the use of TLS.<br><br>Default value: `http://localhost:4317` when protocol is `grpc`, and `http://localhost:4318/v1/metrics` when protocol is `http/protobuf` |
| `otel.exporter.otlp.logs.endpoint` | The OTLP logs endpoint to connect to. MUST be a URL with a scheme of either http or https based on the use of TLS.<br><br>Default value: `http://localhost:4317` when protocol is `grpc`, and `http://localhost:4318/v1/logs` when protocol is `http/protobuf` |
| `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.<br><br>By default the host platform's trusted root certificates are used. |
| `otel.exporter.otlp.metrics.certificate` | The path to the file containing trusted certificates to use when verifying an OTLP metric server's TLS credentials. The file SHOULD contain one or more X.509 certificates in PEM format.<br><br>By default the host platform's trusted root certificates are used. |

| Property Name | Description |
| --- | --- |
| otel.exporter.otlp.logs.certificate | The path to the file containing trusted certificates to use when verifying an OTLP log server's TLS credentials. The file SHOULD contain one or more X.509 certificates in PEM format.<br><br>By default the host platform's trusted root certificates are used. |
| 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.<br><br>By default no client key file is used. |
| otel.exporter.otlp.metrics.client.key | The path to the file containing private client key to use when verifying an OTLP metric client's TLS credentials. The file SHOULD contain one private key PKCS8 PEM format.<br><br>By default no client key file is used. |
| otel.exporter.otlp.logs.client.key | The path to the file containing private client key to use when verifying an OTLP log client's TLS credentials. The file SHOULD contain one private key PKCS8 PEM format.<br><br>By default no client key 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.<br><br>By default no chain file is used. |
| otel.exporter.otlp.metrics.client.certificate | The path to the file containing trusted certificates to use when verifying an OTLP metric server's TLS credentials. The file SHOULD contain one or more X.509 certificates in PEM format.<br><br>By default no chain file is used. |

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

# Service Loader Support

Implementation will load additional configuration related components by means of service loader. This allows the application or runtime extender to define their own metadata and trace / metrics /

log 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 |
| `ConfigurableMetricExporterProvider` | Provides implementation for a name referred in `otel.metrics.exporter` |
| `ConfigurableLogRecordExporterProvider` | Provides implementation for a name referred in `otel.logs.exporter` |

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.

# Supported OpenTelemetry API Classes

Classes from the following API packages MUST be available to applications by implementations of this specification, though this specification does not prevent additional API classes from being available. Implementations are allowed to pull in a more recent patch version of the API classes.

## OpenTelemetry API

**Common API**

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

**Tracing API**

- io.opentelemetry.api.trace

**Baggage API**

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

**Metrics API**

- io.opentelemetry.api.metrics

## Context API

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

## Resource SDK

- io.opentelemetry.sdk.resources

## Metrics SDK

- io.opentelemetry.sdk.metrics

## 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
- io.opentelemetry.sdk.autoconfigure.spi.metrics

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
- io.opentelemetry.sdk.metrics
- io.opentelemetry.sdk.metrics.data
- io.opentelemetry.sdk.metrics.export

# Tracing Annotations

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

# Release Notes

This section documents the changes introduced by individual releases.

# Release Notes for MicroProfile Telemetry 2.0

A full list of changes delivered in the 2.0 release can be found at MicroProfile Telemetry 2.0 Milestone.

## Incompatible Changes

- The Semantic Conventions for HTTP Spans differ from the conventions used with MicroProfile Telemetry 1.1 due to stabilization of HTTP semantic conventions in OpenTelemetry. Changes to attributes are described in HTTP semantic convention stability migration guide.
- For runtimes that can simultaneously run multiple applications, in MicroProfile Telemetry 1.1 the `OTEL_SDK_DISABLED` environment variable could be set to `false` to enable all applications to use separate OpenTelemetry SDK instances. Setting `OTEL_SDK_DISABLED` to `false` in MicroProfile Telemetry 2.0 results in a single OpenTelemetry SDK instance being created for shared use between the runtime and applications. To enable all applications to use separate OpenTelemetry SDK instances in MicroProfile Telemetry 2.0, do not set the `OTEL_SDK_DISABLED` environment variable and set `otel.sdk.disabled` to `false` in a `microprofile-config.properties` file packaged with each application or using any other MicroProfile Config source that is only visible to applications.

## API/SPI Changes

- Consume the OpenTelemetry Java release v1.39.0. The full comparison with the v1.29.0 supported by MicroProfile Telemetry 1.1 can be found here.
- Adopt OpenTelemetry Metrics API (141, 149)

## Other Changes

- Consume the latest OpenTelemetry API (150)
- Adopt OpenTelemetry Logging (146)
- Provide a way to specify runtime configuration for OpenTelemetry (169)
- Specify metrics provided by platform (151)
- TCK: Test required metrics present (143)
- TCK: support Meter injection (145)
- TCK: remove the dependency on Jakarta Concurrency (137)

# 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

None.

## API/SPI Changes

Consume the OpenTelemetry Java release v1.29.0. The full comparison with the v1.19.0 supported by MicroProfile Telemetry 1.0 can be found here.

## Other Changes

- Consume the latest 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 (lhttps://github.com/eclipse/microprofile-telemetry/issues/90[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)
- Typos in spec document (80)