Eclipse MicroProfile OpenTracing

Steve Fontes, Heiko W. Rupp, Pavol Loffay

2.0-RC2, August 03, 2020

Table of Contents

Specification: Eclipse MicroProfile OpenTracing

Version: 2.0-RC2

Status: Draft

Release: August 03, 2020

Chapter 1. Introduction

Distributed tracing allows you to trace the flow of a request across service boundaries. This is particularly important in a microservices environment where a request typically flows through multiple services. To accomplish distributed tracing, each service must be instrumented to log messages with a correlation id that may have been propagated from an upstream service. A common companion to distributed trace logging is a service where the distributed trace records can be stored. See also examples on opentracing.io. The storage service for distributed trace records can provide features to view the cross service trace records associated with particular request flows.

It will be useful for services written in the MicroProfile framework to be able to integrate well with a distributed trace system that is part of the larger microservices environment. This specification defines an API and MicroProfile behaviors that allow services to easily participate in an environment where distributed tracing is enabled.

This specification specifically addresses the problem of making it easy to instrument services with distributed tracing function, given an existing distributed tracing system in the environment.

This specification specifically does not address the problem of defining, implementing, or configuring the underlying distributed tracing system. The proposal assumes an environment where all services use a common OpenTracing implementation (all Zipkin compatible, all Jaeger compatible, ...).

The information about a Span that is propagated between services is typically called a SpanContext. It is not the intent of this specification to define the exact format for how SpanContext information is stored or propagated. Our use case is for applications running in an environment where all applications use the same Tracer implementation, and microservices that require explicit tracing logic use the OpenTracing API. Work on defining standard wire protocols and consistent APIs for handling trace (and metric) data is being done at OpenCensus. The OpenCensus API appears very similar to OpenTracing, but support for OpenCensus Tracers will require a separate MicroProfile specification.

Chapter 2. Rationale

In order for a distributed tracing system to be effective and usable, two things are required

1. The different services in the environment must agree on the mechanism for transferring correlation ids across services.
2. The different services in the environment should produce their trace records in format that is consumable by the storage service for distributed trace records.

Without the first, some services will not be included in the trace records associated with a request. Without the second, custom code would need to be written to present the information about a full request flow.

There are existing distributed tracing systems that provide a server for distributed trace record storage and viewing, and application libraries for instrumenting microservices. The problem is that the different distributed tracing systems use implementation specific mechanisms for propagating correlation IDs and for formatting trace records, so once a microservice chooses a distributed tracing implementation library to use for its instrumentation, all other microservices in the environment are locked into the same choice.

The OpenTracing project's purpose is to provide a standard API for instrumenting microservices for distributed tracing. If every microservice is instrumented for distributed tracing using the OpenTracing API, then (as long as an implementation library exists for the microservice's language), the microservice can be configured at deploy time to use a common system implementation to perform the log record formatting and cross service correlation id propagation. The common implementation ensures that correlation ids are propagated in a way that is understandable to all services, and log records are formatted in a way that is understandable to the server for distributed trace record storage.

In order to make MicroProfile distributed tracing friendly, it will be useful to allow distributed tracing to be enabled on any MicroProfile application, without having to explicitly add distributed tracing code to the application.

In order to make MicroProfile as flexible as possible for adding distributed trace log records, MicroProfile should expose whatever objects are necessary for an application to use the OpenTracing API.

Chapter 3. Architecture

This specification defines an easy way to allow an application running in a MicroProfile container to take advantage of distributed tracing by using an OpenTracing Tracer implementation. This document and implementations MUST comply with OpenTracing specification and semantic conventions if it is not defined otherwise. The currently used OpenTracing API version is 0.33.0.

There are two operation modes

- Without instrumentation of application code
- With explicit code instrumentation

3.1. Enabling distributed tracing with no code instrumentation

The MicroProfile implementation will allow JAX-RS applications to participate in distributed tracing, without requiring developers to add any distributed tracing code into their applications, and without requiring developers to know anything about the distributed tracing environment that their JAX-RS application will be deployed into.

1. The MicroProfile implementation must provide a mechanism to configure an `io.opentracing.Tracer` implementation for use by each JAX-RS application.
2. The MicroProfile implementation must provide a mechanism to automatically extract SpanContext information from any incoming JAX-RS request.
3. The MicroProfile implementation must provide a mechanism to automatically start a Span for any incoming JAX-RS request, and finish the Span when the request completes.
4. The MicroProfile implementation must provide a mechanism to automatically inject SpanContext information into any outgoing JAX-RS request.
5. The MicroProfile implementation must provide a mechanism to automatically start a Span for any outgoing JAX-RS request, and finish the Span when the request completes.

Correct parent child relationships between incoming requests and outgoing requests are handled automatically, as long as the outgoing requests occur on the same thread as the incoming request. If outgoing requests are performed on a different thread than the incoming request, it is the developers responsibility to propagate the Tracer context between threads.

3.1.1. Tracer configuration

An implementation of an `io.opentracing.Tracer` must be made available to each application. Each application will have its own Tracer instance. The Tracer must be configurable outside of the application to match the distributed tracing environment where the application is deployed. For example, it should be possible to take the exact same application and deploy it to an environment where Zipkin is in use, and to deploy the application without modification to a different environment where Jaeger is in use, and the application should report Spans correctly in either environment.

3.1.2. Span creation for inbound requests

When a request arrives at a JAX-RS endpoint, configured Tracer instance is used to extract a SpanContext from the inbound request. The extracted context is used as a child of reference for a new Span created for this endpoint.

Server Span name

The default operation name of the new Span for the incoming request is

```
<HTTP method>:<package name>.<class name>.<method name>
```

The operation name can be configured via key `mp.opentracing.server.operation-name-provider`. The implementation has to provide two operation name providers:

- `class-method` - the provider for the default operation name.
- `http-path` - the operation name has the following form `<HTTP method>:<@Path value of endpoint's class>/<@Path value of endpoint's method>`. For example if the class is annotated with `@Path("service")` and method `@Path("endpoint/{id: \\d+}")` then the operation name is `GET:/service/endpoint/{id: \\d+}`.

If no operation name provider is specified then `class-method` is used.

Server Span tags

Spans created for incoming requests will have the following tags added by default:

- `Tags.SPAN_KIND = Tags.SPAN_KIND_SERVER`
- `Tags.HTTP_METHOD`
- `Tags.HTTP_URL`
- `Tags.HTTP_STATUS`
- `Tags.COMPONENT = "jaxrs"`
- `Tags.ERROR` (if true)

`Tags.SPAN_KIND` MUST be specified at Span start time.

`Tags.ERROR tag` SHOULD be added to a Span on failed operations for any server error (5xx) codes. If there is an exception object available the implementation SHOULD also add logs `event=error` and `error.object=<error object instance>` to the active span.

3.1.3. Span creation and injection for outbound requests

Tracing of client requests is supported for `javax.ws.rs.client.Client` and MicroProfile Rest Client.

When a request is sent from a traced client, a new Span is created and its SpanContext is injected in the outbound request for propagation downstream. The new Span will be a child of the active Span if an active Span exists. The new Span will be finished when the outbound request is completed.

JAX-RS Client

Tracing in `javax.ws.rs.client.Client` has to be explicitly enabled by invoking `org.eclipse.microprofile.opentracing.ClientTracingRegistrar.configure(ClientBuilder clientBuilder)`. The implementation might enable client tracing globally, in this case explicit configuration has no effect.

MicroProfile Rest Client

Tracing for this client is by default globally enabled and it can be disabled by specifying `@Traced(false)` on the client interface or method. When it is specified on the client's interface tracing is disabled for all methods.

Note that integration with MicroProfile Rest Client is not mandatory for vendors not implementing the client specification.

Client Span name

The default operation name of the new Span for the outgoing request is

```
<HTTP method>
```

Client Span tags

Spans created for outgoing requests will have the following tags added by default:

- `Tags.SPAN_KIND = Tags.SPAN_KIND_CLIENT`
- `Tags.HTTP_METHOD`
- `Tags.HTTP_URL`
- `Tags.HTTP_STATUS`
- `Tags.COMPONENT = "jaxrs"`
- `Tags.ERROR` (if true)

`Tags.SPAN_KIND` MUST be specified at Span start time.

`Tags.ERROR` tag SHOULD be added to a Span on failed operations for any client error (4xx) codes. If there is an exception object available the implementation SHOULD also add logs `event=error` and `error.object=<error object instance>` to the active span.

Disabling server side tracing

Server side tracing can be disabled by specifying a skip pattern which is used to match with HTTP path `UriInfo.getPath()`. If the regex matches with HTTP path then tracing for the given server request is disabled even if the method or class is annotated with `@Traced`. The configuration does not disable any outbound request made from the disabled server endpoint.

The skip pattern is specified as a string with key `mp.opentracing.server.skip-pattern` which has to be compliant with `java.util.regex.Pattern`. An example skip pattern might be `mp.opentracing.server.skip-pattern=/foo|/bar.*`

The endpoints defined in the following MicroProfile specifications are always excluded from tracing.

- MicroProfile Health - `/health`
- MicroProfile Metrics - `/metrics`, `/metrics/base/.*`, `/metrics/vendor/.*` and `/metrics/application/.*`
- MicroProfile OpenAPI - `/openapi`

3.2. Enabling explicit distributed tracing code instrumentation

An annotation is provided to define explicit Span creation. This works on top of the "no-action" setup described in Enabling distributed tracing with no code instrumentation.

- `@Traced`: Specify a class or method to be traced.

3.2.1. The traced annotation

The `@Traced` annotation, applies to a class or a method. When applied to a class, the `@Traced` annotation is applied to all methods of the class. If the annotation is applied to a class and method then the annotation applied to the method takes precedence. The annotation starts a Span at the beginning of the method, and finishes the Span at the end of the method.

The `@Traced` annotation has two optional arguments.

- `value=[true|false]`. Defaults to true. If `@Traced` is specified at the class level, then `@Traced(false)` is used to annotate specific methods to disable creation of a Span for those methods. By default all JAX-RS endpoint methods are traced. To disable Span creation of a specific JAX-RS endpoint, the `@Traced(false)` annotation can be used.

  When the `@Traced(false)` annotation is used for a JAX-RS endpoint method, the upstream SpanContext will not be extracted. Any Spans created, either automatically for outbound requests, or explicitly using an injected Tracer, will not have an upstream parent Span in the Span hierarchy.

- `operationName=<Name for the Span>`. Default is `""`. If the `@Traced` annotation finds the `operationName` as `""`, the default operation name is used. For a JAX-RS endpoint method (see Server Span name). If the annotated method is not a JAX-RS endpoint, the default operation name of the new Span for the method is `<package name>.<class name>.<method name>`. If `operationName` is specified on a class, that `operationName` will be used for all methods of the class unless a method explicitly overrides it with its own `operationName`.

Any exceptions thrown by non JAX-RS components must be logged to the span corresponding to the ongoing invocation. The span must be annotated with the following data:

- `Tags.ERROR = true` - added as span tag.
- `event = Tags.ERROR.getKey()` and `error.object = <exception>` logged to span in a single log fields map. The `exception` is the thrown exception object.

Example:

```
@InterceptorBinding
@Target({ TYPE, METHOD })
@Retention(RUNTIME)
public @interface Traced {
    @Nonbinding
    boolean value() default true;
    @Nonbinding
    String operationName() default "";
}
```

3.2.2. Access to the configured tracer

This proposal also specifies that the underlying OpenTracing Tracer object configured instance is available for developer use. The MicroProfile implementation will make the configured Tracer available with CDI injection.

The configured Tracer object is accessed by injecting the Tracer class that has been configured for the particular application for this environment. Each application gets a different Tracer instance.

Example:

```
@Inject
io.opentracing.Tracer configuredTracer;
```

The Tracer object enables support for the more complex tracing requirements, such as creating spans inside business methods.

Access to the Tracer also allows tags, logs and baggage to be added to Spans with, for example:

```
configuredTracer.activeSpan().setTag(...);
configuredTracer.activeSpan().log(...);
configuredTracer.activeSpan().setBaggage(...);
```

Chapter 4. Configuration

MicroProfile OpenTracing project leverages MicroProfile Config specification to provide a consistent means for all supported configuration options. It means that vendor implementations must also be compliant with the Config specification.

All configuration keys supported by this project start with `mp.opentracing`. Refer to the Config specification for precedence of config sources and replacement of illegal characters such as `.` and `-` to `_` when using environmental variables.

4.1. Configuration items

Configuration key	Description
`mp.opentracing.server.skip-pattern`	Specifies a skip pattern to avoid tracing of selected REST endpoints. See Disabling server side tracing.
`mp.opentracing.server.operation-name-provider`	Specifies operation name provider for server spans. Possible values are `http-path` and `class-method`. See Server Span name.

Chapter 5. Impact on existing code

`@Traced` annotations can be added to existing code. A configured Tracer object can be accessed with CDI injection.

Chapter 6. Alternatives considered

Current mechanisms require a decision at development time about the distributed trace system that will be used. This feature allows the decision to be made at the operational environment level.

Chapter 7. Changelog

Release 2.0

Incompatible changes

Update OpenTracing API to 0.33.0 (#177)

The following APIs were removed:

- `Scope = ScopeManager.active()`: no alternative, the reference `Scope` has to be kept explicitly since the scope was created.
- `Scope = ScopeManager.activate(Span, boolean)`: no alternative auto-finishing has been removed.
- `Span = Scope.span()`: use `ScopeManager.activeSpan()` or hold the reference to `Span` explicitly since the span was started.
- `Scope =SpanBuilder.startActive()`: use `Tracer.activateSpan(Span)` instead.
- `Span = Tracer.startManual()`: use `Tracer.start()` instead.
- `AutoFinishScopeManager`: no alternative, auto-finishing has been removed.

Other changes

- Exclude transitive dependency on `javax.el-api` (#196)
- Make OSGI dependency provided (#190)
- Remove OpenTracing API from WAR in TCK (#183)
- Update Arquillian version in TCK to 1.6.0 (#168)
- Use Jakarta EE 8 APIs instead of Java EE 7 and remove dependency on Jackson (#162)

Release 1.3.1

- Add return MIME type to Rest Client interface in TCK (#145)

Release 1.3

- Instrument MicroProfile Rest Client 1.2 (#102)
- Clarify http-path when path contains regular expressions (#136)

Release 1.2.1

- Split tree equals in TCK and remove logs from server spans (#132)
- Remove servlets from TCK and use context root to test MP Metrics, OpenAPI, Health URLs (#127)
- Clarified default skip pattern value (#126)

- Added HTTP method to http-path operation name (#125)
- Added tests for `ClientTracingRegistrar` (#105)
- Update metadata in `pom.xml` (#101)
- Renamed test class to match OpenTracing (#106)

Release 1.2

- By default will not trace endpoints associated with MicroProfile Metrics, Health and OpenAPI (#95)
- Added logging exceptions thrown by explicit instrumentation (`@Traced`)(#94)
- Added server operation name provider (#90)
- Removed `geronimo-atinject` and `javax.annotation` dependencies for API module (#92)
- Added support for server side skip pattern (#86)

Release 1.1

- Added component tag to server and client spans (#70)
- Explicitly enabled tracing for JAX-RS clients (#64)
- Updated OpenTracing API from 0.30.0 to 0.31.0. Note that these two versions contain breaking changes (#67).