Eclipse MicroProfile OpenTracing

Steve Fontes, Heiko W. Rupp, Pavol Loffay

1.0-RC1, December 13, 2017

Table of Contents

Specification: Eclipse MicroProfile OpenTracing

Version: 1.0-RC1

Status: Draft

Release: December 13, 2017

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.io 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 common formats for SpanContexts and implementations that use the common format is being done at open-census. The open-census API appears very similar to opentracing, but support for open-census Tracers will probably 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.io 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.io 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.io 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.io Tracer implementation.

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. SpanContext extraction

When a request arrives at a JAX-RS endpoint, an attempt is made to use the configured Tracer to extract a SpanContext from the arriving request. If a SpanContext is extracted, it is used as the parent for the new Span that is created for the endpoint.

3.1.3. Span creation for inbound requests

When a request arrives at a JAX-RS endpoint, a new Span is created. The new Span will be a child of the SpanContext extracted from the incoming request, if the extracted SpanContext exists.

Server Span name

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

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

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.ERROR (if true)

Tags.SPAN_KIND MUST be specified at Span start time.

An 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.

3.1.4. Span creation and injection for outbound requests

When a request is sent from a JAX-RS javax.ws.rs.client.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.

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.ERROR (if true)

Tags.SPAN_KIND MUST be specified at Span start time.

An 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.

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 operationName 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>
```

Example:

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

3.2.2. io.opentracing.Tracer access

This proposal also specifies that the underlying opentracing.io 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;
```

Access to the configured Tracer gives full access to opentracing.io functions.

The Tracer object enables support for the more complex tracing requirements, such as when a Span is started in one method, and finished in another.

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. Impact on existing code

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

Chapter 5. 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.