Quarkus - Using OpenTracing

This guide explains how your Quarkus application can utilize opentracing to provide distributed tracing for interactive web applications.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+
- Docker

Architecture

In this guide, we create a straightforward REST application to demonstrate distributed tracing.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can skip right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `opentracing-quickstart` directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.2.0.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=opentracing-quickstart \
    -DclassName="org.acme.opentracing.TracedResource" \
    -Dpath="/hello" \
    -Dextensions="opentracing"
cd opentracing-quickstart
```

This command generates the Maven project with a REST endpoint and imports the `smallrye-`

`opentracing` extension, which includes the OpenTracing support and the default Jaeger tracer.

Examine the JAX-RS resource

Open the `src/main/java/org/acme/opentracing/TracedResource.java` file and see the following content:

```java
package org.acme.opentracing;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class TracedResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

Notice that there is no tracing specific code included in the application. By default, requests sent to this endpoint will be traced without any code changes being required. It is also possible to enhance the tracing information. For more information on this, please see the MicroProfile OpenTracing specification.

Create the configuration

There are two ways to configure the Jaeger tracer within the application.

The first approach is by providing the properties within the `src/main/resources/application.properties` file:

```
quarkus.jaeger.service-name=myservice ①
quarkus.jaeger.sampler-type=const ②
quarkus.jaeger.sampler-param=1 ③
```

① If the `quarkus.jaeger.service-name` property (or `JAEGER_SERVICE_NAME` environment variable) is not provided then a "no-op" tracer will be configured, resulting in no tracing data being reported to the backend.

② Setup a sampler, that uses a constant sampling strategy.

③ Sample all requests. Set sampler-param to somewhere between 0 and 1, e.g. 0.50, if you do not wish to sample all requests.

The second approach is to supply the properties as environment variables. These can be specified as `jvm.args` as shown in the following section.

Run the application

The first step is to start the tracing system to collect and display the captured traces:

```
docker run -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 -p 5775:5775/udp -p
6831:6831/udp -p 6832:6832/udp -p 5778:5778 -p 16686:16686 -p
14268:14268 -p 9411:9411 jaegertracing/all-in-one:latest
```

Now we are ready to run our application. If using `application.properties` to configure the tracer:

```
./mvnw compile quarkus:dev
```

or if configuring the tracer via environment variables:

```
./mvnw compile quarkus:dev -Djvm.args="
-DJAEGER_SERVICE_NAME=myservice -DJAEGER_SAMPLER_TYPE=const
-DJAEGER_SAMPLER_PARAM=1"
```

Once both the application and tracing system are started, you can make a request to the provided endpoint:

```
$ curl http://localhost:8080/hello
hello
```

When the first request has been submitted, the Jaeger tracer within the app will be initialized:

```
2019-10-16 09:35:23,464 INFO  [io.jae.Configuration] (executor-
thread-1) Initialized tracer=JaegerTracer(version=Java-0.34.0,
serviceName=myservice, reporter=RemoteReporter(sender=UdpSender(),
closeEnqueueTimeout=1000), sampler=ConstSampler(decision=true,
tags={sampler.type=const, sampler.param=true}),
tags={hostname=localhost.localdomain, jaeger.version=Java-0.34.0,
ip=127.0.0.1}, zipkinSharedRpcSpan=false,
expandExceptionLogs=false, useTraceId128Bit=false)
```

Then visit the Jaeger UI to see the tracing information.

Hit CTRL+C to stop the application.

Additional instrumentation

The OpenTracing API Contributions project offers additional instrumentation that can be used to add tracing to a large variety of technologies/components.

The instrumentation documented in this section has been tested with Quarkus and works in both standard and native mode.

JDBC

The JDBC instrumentation will add a span for each JDBC queries done by your application, to enable it, add the following dependency to your pom.xml:

```
<dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-jdbc</artifactId>
</dependency>
```

As it uses a dedicated JDBC driver, you must configure your datasource and Hibernate to use it.

```
# add ':tracing' to your database URL
quarkus.datasource.url=jdbc:tracing:postgresql://localhost:5432/mydatabase
# use the 'TracingDriver' instead of the one for your database
quarkus.datasource.driver=io.opentracing.contrib.jdbc.TracingDriver
# configure Hibernate dialect
quarkus.hibernate-orm.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Jaeger Configuration Reference

⬛ Configuration property fixed at build time - ⚙ Configuration property overridable at runtime

Configuration property	Type	Default
⬛ quarkus.jaeger.enabled Defines if the Jaeger extension is enabled.	boolean	true
⚙ quarkus.jaeger.endpoint The traces endpoint, in case the client should connect directly to the Collector, like http://jaeger-collector:14268/api/traces	URI	

⚙ `quarkus.jaeger.auth-token` Authentication Token to send as "Bearer" to the endpoint	string	
⚙ `quarkus.jaeger.user` Username to send as part of "Basic" authentication to the endpoint	string	
⚙ `quarkus.jaeger.password` Password to send as part of "Basic" authentication to the endpoint	string	
⚙ `quarkus.jaeger.agent-host-port` The hostname and port for communicating with agent via UDP	host:port	
⚙ `quarkus.jaeger.reporter-log-spans` Whether the reporter should also log the spans	boolean	
⚙ `quarkus.jaeger.reporter-max-queue-size` The reporter's maximum queue size	int	
⚙ `quarkus.jaeger.reporter-flush-interval` The reporter's flush interval	Duration ❓	
⚙ `quarkus.jaeger.sampler-type` The sampler type (const, probabilistic, ratelimiting or remote)	string	
⚙ `quarkus.jaeger.sampler-param` The sampler parameter (number)	BigDecimal	
⚙ `quarkus.jaeger.sampler-manager-host-port` The host name and port when using the remote controlled sampler	host:port	
⚙ `quarkus.jaeger.service-name` The service name	string	

⚙ `quarkus.jaeger.tags` A comma separated list of name = value tracer level tags, which get added to all reported spans. The value can also refer to an environment variable using the format ${envVarName:default}, where the :default is optional, and identifies a value to be used if the environment variable cannot be found	string	
⚙ `quarkus.jaeger.propagation` Comma separated list of formats to use for propagating the trace context. Defaults to the standard Jaeger format. Valid values are jaeger and b3	string	
⚙ `quarkus.jaeger.sender-factory` The sender factory class name	string	
⚙ `quarkus.jaeger.log-trace-context` Whether the trace context should be logged.	boolean	`true`

ℹ️ *About the Duration format*

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, PT is implicitly prepended to the value to obtain a standard `java.time.Duration` format.