Using OpenTracing

Overview

OpenTracing is a vendor-neutral open standard for distributed tracing. Essentially, for Java-based projects the specification exists as a set of Java APIs which any distributed tracing solution is welcome to implement. There are available which are compatible with quite a few distributed tracing frameworks notably Zipkin, Lightstep, and Jaeger. Starting from release, Apache CXF fully supports integration (through cxf-integration-tracing-opentracing module) with any distributed tracer that provides OpenTracing Java API implementation.

The section dedicated to Apache HTrace has pretty good introduction into distributed tracing basics however OpenTracing specification abstracts a lot of things, outlining just a general APIs to denote the Span lifecycle and injection points to propagate the context across many distributed components. As such, the intrinsic details about HTTP headers f.e. becomes an integral part of the distributed tracer of your choice, out of reach for Apache CXF.

Distributed Tracing in Apache CXF using OpenTracing

Apache CXF is a very popular framework for building services and web APIs. No doubts, it is going to play even more important role in context of microservices architecture letting developers to quickly build and deploy individual JAX-RS/JAX-WS services. Distributed tracing is an essential technique to observe the application platform as a whole, breaking the request to individual service traces as it goes through and crosses the boundaries of threads, processes and machines.

The current integration of distributed tracing in Apache CXF supports OpenTracing Java API 0.30.0+ and provides full-fledged support of JAX-RS 2.x / JAX-WS applications. From high-level prospective, the JAX-RS integration consists of three main parts:

- TracerContext (injectable through @Context annotation)
- OpenTracingProvider (server-side JAX-RS provider) and OpenTracingClientProvider (client-side JAX-RS provider)
- OpenTracingFeature (server-side JAX-RS feature) to simplify the configuration and integration

Similarly, from high-level perspective, JAX-WS integration includes:

- OpenTracingStartInterceptor / OpenTracingStopInterceptor / OpenTracingFeature Apache CXF feature (server-side JAX-WS support)
- OpenTracingClientStartInterceptor / OpenTracingClientStopInterceptor / OpenTracingClientFeature Apache CXF feature (client-side JAX-WS support)

Apache CXF uses HTTP headers to hand off tracing context from the client to the service and from the service to service. Those headers are specific to distributing tracing framework you have picked and are not configurable at the moment (unless the framework itself has a way to do that).
By default, **OpenTracingClientProvider** will try to pass the currently active **span** through HTTP headers on each service invocation. If there is no active spans, the new span will be created and passed through HTTP headers on per-invocation basis. Essentially, for JAX-RS applications just registering **OpenTracingClientProvider** on the client and **OpenTracingProvider** on the server is enough to have tracing context to be properly passed everywhere. The only configuration part which is necessary are **span reporter(s)** and **sampler(s)** which are, not surprisingly, specific to distributing tracing framework you have chosen.

It is also worth to mention the way **Apache CXF** attaches the description to **spans**. With regards to the client integration, the description becomes a full URL being invoked prefixed by HTTP method, for example: GET `http://localhost:8282/books`. On the server side integration, the description becomes a relative JAX-RS resource path prefixed by HTTP method, f.e.: GET `books`, POST `book/123`.

### A Note on OpenTracing APIs

**OpenTracing** Java API is evolving very fast and, sadly but not surprisingly, often the changes being made are not backward compatible. The **Apache CXF 3.2.x** release branch stays on **OpenTracing Java API 0.30.0** as of now, while the **Apache CXF 3.3.x** is using **OpenTracing Java API 0.31.0**. There are quite many major differences between both APIs but **Apache CXF** is trying hard to smooth it over. It is worth to mention that **OpenTracing**-compatible clients and servers may not depend on the same APIs version, the only issue you will run into is related to compatibility of the provided Java clients for the tracer of your choice.

### OpenTracing API v0.30.0 and Apache CXF 3.2.x

#### Configuring Client

In this section and below, all the code snippets are going to be based on **Jaeger** distributed tracing framework (release 0.20.6+), although everything we are going to discuss is equally applicable to any other existing alternatives. Essentially, the only dependency **Apache CXF** integration relies on is the **Tracer** instance.

There are a couple of ways the JAX-RS client could be configured, depending on the client implementation. **Apache CXF** provides its own **WebClient** which could be configured just like that (in future versions, there would be a simpler ways to do that using client specific features):

```java
final Tracer tracer = new Configuration("web-client",
    new Configuration.SamplerConfiguration(ConstSampler.TYPE, 1), /* or any other Sampler */
    new Configuration.ReporterConfiguration(new HttpSender("http://localhost:14268/api/traces"))) /* or any other Sender */
    .getTracer();

Response response = WebClient.create("http://localhost:9000/catalog", Arrays.asList(new OpenTracingClientProvider(tracer)))
    .accept(MediaType.APPLICATION_JSON)
    .get();
```

The configuration based on using the standard JAX-RS **Client** is very similar:

```java
final Tracer tracer = new Configuration("jaxrs-client",
    new Configuration.SamplerConfiguration(ConstSampler.TYPE, 1), /* or any other Sampler */
    new Configuration.ReporterConfiguration(new HttpSender("http://localhost:14268/api/traces"))) /* or any other Sender */
    .getTracer();

final OpenTracingClientProvider provider = new OpenTracingClientProvider(tracer);
final Client client = ClientBuilder.newClient().register(provider);

final Response response = client
    .target("http://localhost:9000/catalog")
    .request()
    .accept(MediaType.APPLICATION_JSON)
    .get();
```

Alternatively, you may use **GlobalTracer** to pass the tracer around, for example:
final Tracer tracer = new Configuration("jaxrs-client",
        new Configuration.SamplerConfiguration(ConstSampler.TYPE, 1), /* or any other Sampler */
        new Configuration.ReporterConfiguration(new HttpSender("http://localhost:14268/api/traces")) /* or any
        other Sender */
    ).getTracer();

    // This method should only be called once during the application initialization phase.
    GlobalTracer.register(tracer);

    // No explicit Tracer instance is required, it will be picked off the GlobalTracer using get() method
    final OpenTracingClientProvider provider = new OpenTracingClientProvider();

Configuring Server

Server configuration is a bit simpler than the client one thanks to the feature class available, OpenTracingFeature. Depending on the way the Apache CXF is used to configure JAX-RS services, it could be part of JAX-RS application configuration, for example:

```java
@ApplicationPath( "/" )
public class CatalogApplication extends Application {
    @Override
    public Set<Object> getSingletons() {
        final Tracer tracer = new Configuration("tracer-server",
            new Configuration.SamplerConfiguration(ConstSampler.TYPE, 1), /* or any other Sampler */
            new Configuration.ReporterConfiguration(new HttpSender("http://localhost:14268/api/traces")) /* or any
            other Sender */
        ).getTracer();

        return new HashSet<>(
            Arrays.asList(
                new OpenTracingFeature(tracer)
            )
        );
    }
}
```

Or it could be configured using JAXRSServerFactoryBean as well, for example:

```java
final Tracer tracer = new Configuration("tracer-server",
    new Configuration.SamplerConfiguration(ConstSampler.TYPE, 1), /* or any other Sampler */
    new Configuration.ReporterConfiguration(new HttpSender("http://localhost:14268/api/traces")) /* or any
    other Sender */
).getTracer();

final JAXRSServerFactoryBean factory = RuntimeDelegate.getInstance().createEndpoint(/* application instance */,
    JAXRSServerFactoryBean.class);
factory.setProvider(new OpenTracingFeature(tracer));
...
return factory.create();
```

Alternatively, you may rely on GlobalTracer to pass the tracer around, so in this case the OpenTracingFeature will pick it up from there, for example:
Once the `span reporter` and `sampler` are properly configured, all generated `spans` are going to be collected and available for analysis and/or visualization.

Distributed Tracing In Action: Usage Scenarios

In the following subsections we are going to walk through many different scenarios to illustrate the distributed tracing in action, starting from the simplest ones and finishing with asynchronous JAX-RS services. All examples assume that configuration has been done (see please Configuring Client and Configuring Server sections above).

Example #1: Client and Server with default distributed tracing configured

In the first example we are going to see the effect of using default configuration on the client and on the server, with only `OpenTracingClientProvider` and `OpenTracingProvider` registered. The JAX-RS resource endpoint is pretty basic stubbed method:

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks() {
    return Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
}
```

The client is as simple as that:

```java
final Response response = client
    .target("http://localhost:8282/books")
    .request()
    .accept(MediaType.APPLICATION_JSON)
    .get();
```

The actual invocation of the request by the client (with service name `tracer-client`) and consequent invocation of the service on the server side (service name `tracer-server`) is going to generate the following sample traces (taken from Jaeger UI):

![Jaeger UI Trace](image)

The same trace will be looking pretty similar using traditional Zipkin UI frontend:
Example #2: Client and Server with nested trace

In this example server-side implementation of the JAX-RS service is going to call an external system (simulated as a simple delay of 500ms) within its own span. The client-side code stays unchanged.

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    try(final ActiveSpan scope = tracer.startSpan("Calling External System")) {
        // Simulating a delay of 500ms required to call external system
        Thread.sleep(500);
        return Arrays.asList(
            new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
        );
    }
}
```

The actual invocation of the request by the client (with service name `tracer-client`) and consequent invocation of the service on the server side (service name `tracer-server`) is going to generate the following sample traces (taken from Jaeger UI):

![Jaeger UI trace](image)

The same trace will be looking pretty similar using traditional Zipkin UI frontend:

![Zipkin UI trace](image)

Example #3: Client and Server trace with timeline

In this example server-side implementation of the JAX-RS service is going to add timeline to the active span. The client-side code stays unchanged.
@Produces( { MediaType.APPLICATION_JSON } )

public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    tracer.timeline("Preparing Books");
    // Simulating some work using a delay of 100ms
    Thread.sleep(100);
    return Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
}

The actual invocation of the request by the client (with service name tracer-client) and consequent invocation of the service on the server side (service name tracer-server) is going to generate the following sample traces (taken from Jaeger UI):

Please notice that timelines are treated as logs events in Jaeger.

The same trace will be looking pretty similar using traditional Zipkin UI frontend:

Example #4: Client and Server with binary annotations (key/value)

In this example server-side implementation of the JAX-RS service is going to add key/value annotations to the active span. The client-side code stays unchanged.
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    final Collection<Book> books = Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
    tracer.annotate("# of books", Integer.toString(books.size()));
    return books;
}

The actual invocation of the request by the client (with service name tracer-client) and consequent invocation of the service on the server side (service name tracer-server) is going to generate the following sample server trace properties (taken from Jaeger UI):

![Jaeger UI Trace](image)

The same trace will be looking pretty similar using traditional Zipkin UI frontend:

![Zipkin UI Trace](image)

Example #5: Client and Server with parallel trace (involving thread pools)

In this example server-side implementation of the JAX-RS service is going to offload some work into thread pool and then return the response to the client, simulating parallel execution. The client-side code stays unchanged.
```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    final Future<Book> book1 = executor.submit(
        tracer.wrap("Getting Book 1", new Traceable<Book>() {
            public Book call(final TracerContext context) throws Exception {
                // Simulating a delay of 100ms required to call external system
                Thread.sleep(100);
                return new Book("Apache CXF Web Service Development", 
                                "Naveen Balani, Rajeev Hathi");
            }
        })
    );
    final Future<Book> book2 = executor.submit(
        tracer.wrap("Getting Book 2", new Traceable<Book>() {
            public Book call(final TracerContext context) throws Exception {
                // Simulating a delay of 100ms required to call external system
                Thread.sleep(200);
                return new Book("Developing Web Services with Apache CXF and Axis2", 
                                "Kent Ka Iok Tong");
            }
        })
    );
    return Arrays.asList(book1.get(), book2.get());
}
```

The actual invocation of the request by the client (with service name `tracer-client`) and consequent invocation of the service on the server side (process name `tracer-server`) is going to generate the following sample traces (taken from Jaeger UI):

![Jaeger UI Trace](image)

The same trace will be looking pretty similar using traditional Zipkin UI frontend:

![Zipkin UI Trace](image)

Example #6: Client and Server with asynchronous JAX-RS service (server-side)
In this example, the server-side implementation of the JAX-RS service is going to be executed asynchronously. It poses a challenge from the tracing perspective as request and response are processed in different threads (in general). At the moment, Apache CXF does not support the transparent tracing spans management (except for default use case) but provides the simple ways to do that (by letting to transfer spans from thread to thread). The client-side code stays unchanged.

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public void getBooks(@Suspended final AsyncResponse response, @Context final TracerContext tracer) throws Exception {
    tracer.continueSpan(new Traceable<Future<Void>>() {
        public Future<Void> call(final TracerContext context) throws Exception {
            return executor.submit(
                tracer.wrap("Getting Book", new Traceable<Void>() {
                    public Void call(final TracerContext context) throws Exception {
                        Thread.sleep(50);
                        response.resume(
                            Arrays.asList(
                                new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
                            )
                        );
                        return null;
                    }
                })
            );
        }
    });
}
```

The actual invocation of the request by the client (with service name `tracer-client`) and consequent invocation of the service on the server side (service name `tracer-server`) is going to generate the following sample traces (taken from Jaeger UI):

![Jaeger UI Trace](image)

The same trace will be looking pretty similar using traditional Zipkin UI frontend:

![Zipkin UI Trace](image)

**Example #7: Client and Server with asynchronous invocation (client-side)**

In this example server-side implementation of the JAX-RS service is going to be the default one:
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks() {
    return Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
}

While the JAX-RS client implementation is going to perform the asynchronous invocation:

```java
final Future<Response> future = client
    .target("http://localhost:8282/books")
    .request()
    .accept(MediaType.APPLICATION_JSON)
    .async()
    .get();
```

In this respect, there is no difference from the caller prospective however a bit more work is going under the hood to transfer the active tracing span from JAX-RS client request filter to client response filter as in general those are executed in different threads (similarly to server-side asynchronous JAX-RS resource invocation). The actual invocation of the request by the client (with service name `tracer-client`) and consequent invocation of the service on the server side (service name `tracer-server`) is going to generate the following sample traces (taken from Jaeger UI):

![Jaeger UI Trace](image)

The same trace will be looking pretty similar using traditional Zipkin UI frontend:

![Zipkin UI Trace](image)

Distributed Tracing with OpenTracing and JAX-WS support

Distributed tracing in the Apache CXF is build primarily around JAX-RS 2.x implementation. However, JAX-WS is also supported but it requires to write some boiler-plate code and use OpenTracing Java API directly (the JAX-WS integration is going to be enhanced in the future). Essentially, from the server-side prospective the in/out interceptors, `OpenTracingStartInterceptor` and `OpenTracingStopInterceptor` respectively, should be configured as part of interceptor chains, either manually or using `OpenTracingFeature`. For example:

```java
final Tracer tracer = new Configuration("tracer",
    new Configuration.SamplerConfiguration(ConstSampler.TYPE, 1), /* or any other Sampler */
    new Configuration.ReporterConfiguration(new HttpSender("http://localhost:14268/api/traces")) /* or any other Sender */
).getTracer();

final JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
... sf.getFeatures().add(new OpenTracingFeature(tracer));
... sf.create();
```
Similarly to the server-side, client-side needs own set of out/in interceptors, `OpenTracingClientStartInterceptor` and `OpenTracingClientStopInterceptor` (or `OpenTracingClientFeature`). Please notice the difference from server-side: `OpenTracingClientStartInterceptor` becomes out-interceptor while `OpenTracingClientStopInterceptor` becomes in-interceptor. For example:

```java
final Tracer tracer = new Configuration("tracer",
   new Configuration.SamplerConfiguration(ConstSampler.TYPE, 1), /* or any other Sampler */
   new Configuration.ReporterConfiguration(new HttpSender("http://localhost:14268/api/traces")) /* or any other Sender */
).getTracer();

final JaxWsProxyFactoryBean sf = new JaxWsProxyFactoryBean();
...
sf.getFeatures().add(new OpenTracingClientFeature(tracer));
...
```

As it was mentioned before, you may use `GlobalTracer` utility class to pass the tracer around so, for example, any JAX-WS service will be able to retrieve the current tracer by invoking `GlobalTracer.get()` method.

**Distributed Tracing with OpenTracing and OSGi**

Most of the distributed tracers compatible with OpenTracing API could be deployed into OSGi container and as such, the integration is fully available for Apache CXF services running inside the container. For a complete example please take a look on `jax_rs_tracing_opentracing_osgi` sample project, but here is the typical OSGi Blueprint snippet in case of Jaeger.
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:cxf="http://cxf.apache.org/blueprint/core"
xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"

xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/v1.0.0
/blueprint.xsd


<bean id="tracingFeature" class="org.apache.cxf.tracing.opentracing.jaxrs.OpenTracingFeature">
  <argument index="0">
    <bean factory-ref="builder" factory-method="build" />
  </argument>
</bean>

<bean id="metrics" class="com.uber.jaeger.metrics.Metrics">
  <argument index="0">
    <bean class="com.uber.jaeger.metrics.StatsFactoryImpl">
      <argument index="0">
        <bean class="com.uber.jaeger.metrics.NullStatsReporter" />
      </argument>
    </bean>
  </argument>
</bean>

<bean id="builder" class="com.uber.jaeger.Tracer.Builder">
  <argument index="0" value="cxf-server" />
  <argument index="1">
    <bean class="com.uber.jaeger.reporters.RemoteReporter">
      <argument index="0" ref="sender" />
      <argument index="1" value="1000" />
      <argument index="2" value="100" />
      <argument index="3" ref="metrics" />
    </bean>
  </argument>
  <argument index="2">
    <bean class="com.uber.jaeger.samplers.ConstSampler">
      <argument index="0" value="true" />
    </bean>
  </argument>
</bean>

<bean id="sender" class="com.uber.jaeger.senders.HttpSender">
  <argument index="0" value="http://localhost:14268/api/traces" />
</bean>

<cxfs:bus>
  <cxfs:features>
    <cxfs:logging />
  </cxfs:features>
</cxfs:bus>

<jaxrs:server id="catalogServer" address="/">
  <jaxrs:serviceBeans>
    ...
  </jaxrs:serviceBeans>
  <jaxrs:providers>
    <ref component-id="tracingFeature" />
  </jaxrs:providers>
</jaxrs:server>
</blueprint>

Samples

- https://github.com/apache/cxf/tree/3.2.x-fixes/distribution/src/main/release/samples/jax_rs/tracing_opentracing
- https://github.com/apache/cxf/tree/3.2.x-fixes/distribution/src/main/release/samples/jax_rs/tracing_opentracing_camel
OpenTracing API v0.31.0 and Apache CXF 3.3.x

Configuring Client

In this section and below, all the code snippets are going to be based on Jaeger distributed tracing framework (release 0.30.3+), although everything we are going to discuss is equally applicable to any other existing alternatives. Essentially, the only dependency Apache CXF integration relies on is the Tracer instance. Jaeger uses service Java’s ServiceLoader mechanism to determine the instance of the tracer to use, so it is necessary to provide META-INF/services/io.jaegertracing.spi.SenderFactory binding, for example:

```java
io.jaegertracing.thrift.internal.senders.ThriftSenderFactory
```

Alternatively, you may just provide own implementation of the SenderConfiguration with the override getSender method, for example:

```java
final SenderConfiguration senderConfiguration = new SenderConfiguration() {
    @Override
    public Sender getSender() {
        return ...; /* the desired Sender implementation */
    }
}
```

There are a couple of ways the JAX-RS client could be configured, depending on the client implementation. Apache CXF provides its own WebClient which could be configured just like that (in future versions, there would be a simpler ways to do that using client specific features):

```java
final Tracer tracer = new Configuration("web-client")
    .withSampler(
        new SamplerConfiguration()
            .withType(ConstSampler.TYPE) /* or any other Sampler */
            .withParam(1)
    )
    .withReporter(
        new ReporterConfiguration()
            .withSender(
                new SenderConfiguration() /* or any other Sender configuration */
                    .withEndpoint("http://localhost:14268/api/traces")
            )
    )
    .getTracer();
Response response = WebClient
    .create("http://localhost:9000/catalog", Arrays.asList(new OpenTracingClientProvider(tracer)))
    .accept(MediaType.APPLICATION_JSON)
    .get();
```

The configuration based on using the standard JAX-RS Client is very similar:
final Tracer tracer = new Configuration("jaxrs-client")
    .withSampler(
        new SamplerConfiguration()
            .withType(ConstSampler.TYPE) /* or any other Sampler */
            .withParam(1)
    )
    .withReporter(
        new ReporterConfiguration()
            .withSender(
                new SenderConfiguration() /* or any other Sender configuration */
                    .withEndpoint("http://localhost:14268/api/traces")
            )
    )

final OpenTracingClientProvider provider = new OpenTracingClientProvider(tracer);
final Client client = ClientBuilder.newClient().register(provider);

final Response response = client
    .target("http://localhost:9000/catalog")
    .request()
    .accept(MediaType.APPLICATION_JSON)
    .get();

Alternatively, you may use GlobalTracer to pass the tracer around, for example:

final Tracer tracer = new Configuration("jaxrs-client")
    .withSampler(
        new SamplerConfiguration()
            .withType(ConstSampler.TYPE) /* or any other Sampler */
            .withParam(1)
    )
    .withReporter(
        new ReporterConfiguration()
            .withSender(
                new SenderConfiguration() /* or any other Sender configuration */
                    .withEndpoint("http://localhost:14268/api/traces")
            )
    )
    .getTracer();

// This method should only be called once during the application initialization phase.
GlobalTracer.register(tracer);

// No explicit Tracer instance is required, it will be picked off the GlobalTracer using get() method
final OpenTracingClientProvider provider = new OpenTracingClientProvider();

**Configuring Server**

Server configuration is a bit simpler than the client one thanks to the feature class available, OpenTracingFeature. Depending on the way the Apache CXF is used to configure JAX-RS services, it could be part of JAX-RS application configuration, for example:
@ApplicationPath( "/" )
public class CatalogApplication extends Application {
    @Override
    public Set<Object> getSingletons() {
        final Tracer tracer = new Configuration("tracer-server")
            .withSampler(
                new SamplerConfiguration()
                    .withType(ConstSampler.TYPE) /* or any other Sampler */
                    .withParam(1)
            )
            .withReporter(
                new ReporterConfiguration()
                    .withSender(
                        new SenderConfiguration() /* or any other Sender configuration */
                            .withEndpoint("http://localhost:14268/api/traces")
                    )
            )
            .getTracer();
        return new HashSet<>(
            Arrays.asList(
                new OpenTracingFeature(tracer)
            )
        );
    }
}

Or it could be configured using JAXRSServerFactoryBean as well, for example:

    final Tracer tracer = new Configuration("tracer-server")
            .withSampler(
                new SamplerConfiguration()
                    .withType(ConstSampler.TYPE) /* or any other Sampler */
                    .withParam(1)
            )
            .withReporter(
                new ReporterConfiguration()
                    .withSender(
                        new SenderConfiguration() /* or any other Sender configuration */
                            .withEndpoint("http://localhost:14268/api/traces")
                    )
            )
            .getTracer();

    final JAXRSServerFactoryBean factory = RuntimeDelegate.getInstance().createEndpoint(/* application instance */,
        JAXRSServerFactoryBean.class);
    factory.setProvider(new OpenTracingFeature(tracer));
    ...
    return factory.create();

Alternatively, you may rely on GlobalTracer to pass the tracer around, so in this case the OpenTracingFeature will pick it up from there, for example:

    @ApplicationPath( "/" )
public class CatalogApplication extends Application {
    @Override
    public Set<Object> getSingletons() {
        return new HashSet<>(
            Arrays.asList(
                new OpenTracingFeature()
                    .get() // No explicit Tracer instance is required, it will be picked off the GlobalTracer using get() method
            )
        );
    }
}
Once the **span reporter** and **sampler** are properly configured, all generated **spans** are going to be collected and available for analysis and/or visualization.

## Distributed Tracing In Action: Usage Scenarios

In the following subsections we are going to walk through many different scenarios to illustrate the distributed tracing in action, starting from the simplest ones and finishing with asynchronous JAX-RS services. All examples assume that configuration has been done (see please [Configuring Client](#) and Configuring Server sections above).

### Example #1: Client and Server with default distributed tracing configured

In the first example we are going to see the effect of using default configuration on the client and on the server, with only OpenTracingClientProvider and OpenTracingProvider registered. The JAX-RS resource endpoint is pretty basic stubbed method:

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks() {
    return Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
}
```

The client is as simple as that:

```java
final Response response = client
    .target("http://localhost:8282/books")
    .request()
    .accept(MediaType.APPLICATION_JSON)
    .get();
```

The actual invocation of the request by the client (with service name **tracer-client**) and consequent invocation of the service on the server side (service name **tracer-server**) is going to generate the following sample traces (taken from Jaeger UI):

![Jaeger UI](#)

### Example #2: Client and Server with nested trace

In this example server-side implementation of the JAX-RS service is going to call an external system (simulated as a simple delay of 500ms) within its own span. The client-side code stays unchanged.

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    try(final Scope scope = tracer.startSpan("Calling External System")) {
        // Simulating a delay of 500ms required to call external system
        Thread.sleep(500);

        return Arrays.asList(
            new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
        );
    }
}
```

The actual invocation of the request by the client (with service name **tracer-client**) and consequent invocation of the service on the server side (service name **tracer-server**) is going to generate the following sample traces (taken from Jaeger UI):

![Jaeger UI](#)
Example #3: Client and Server trace with timeline

In this example server-side implementation of the JAX-RS service is going to add timeline to the active span. The client-side code stays unchanged.

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    tracer.timeline("Preparing Books");
    // Simulating some work using a delay of 100ms
    Thread.sleep(100);
    return Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
}
```

The actual invocation of the request by the client (with service name tracer-client) and consequent invocation of the service on the server side (service name tracer-server) is going to generate the following sample traces (taken from Jaeger UI):

Example #4: Client and Server with annotations (key/value)

In this example server-side implementation of the JAX-RS service is going to add key/value annotations to the active span. The client-side code stays unchanged.

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    final Collection<Book> books = Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
    tracer.annotate("# of books", Integer.toString(books.size()));
    return books;
}
```

The actual invocation of the request by the client (with service name tracer-client) and consequent invocation of the service on the server side (service name tracer-server) is going to generate the following sample server trace properties (taken from Jaeger UI):
Example #5: Client and Server with parallel trace (involving thread pools)

In this example server-side implementation of the JAX-RS service is going to offload some work into thread pool and then return the response to the client, simulating parallel execution. The client-side code stays unchanged.

```java
@Produces({ MediaType.APPLICATION_JSON })
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    final Future<Book> book1 = executor.submit(
        tracer.wrap("Getting Book 1", new Traceable<Book>() {
            public Book call(final TracerContext context) throws Exception {
                // Simulating a delay of 100ms required to call external system
                Thread.sleep(100);
                return new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi");
            }
        })
    );
    final Future<Book> book2 = executor.submit(
        tracer.wrap("Getting Book 2", new Traceable<Book>() {
            public Book call(final TracerContext context) throws Exception {
                // Simulating a delay of 100ms required to call external system
                Thread.sleep(200);
                return new Book("Developing Web Services with Apache CXF and Axis2", "Kent Ka Iok Tong");
            }
        })
    );
    return Arrays.asList(book1.get(), book2.get());
}
```

The actual invocation of the request by the client (with service name `tracer-client`) and consequent invocation of the service on the server side (process name `tracer-server`) is going to generate the following sample traces (taken from Jaeger UI):

Example #6: Client and Server with asynchronous JAX-RS service (server-side)
In this example server-side implementation of the JAX-RS service is going to be executed asynchronously. It poses a challenge from the tracing prospective as request and response are processed in different threads (in general). At the moment, Apache CXF does not support the transparent tracing spans management (except for default use case) but provides the simple ways to do that (by letting to transfer spans from thread to thread). The client-side code stays unchanged.

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public void getBooks(@Suspended final AsyncResponse response, @Context final TracerContext tracer) throws Exception {
    tracer.continueSpan(new Traceable<Future<Void>>() {
        public Future<Void> call(final TracerContext context) throws Exception {
            return executor.submit(
                tracer.wrap("Getting Book", new Traceable<Void>() {
                    public Void call(final TracerContext context) throws Exception {
                        // Simulating a processing delay of 50ms
                        Thread.sleep(50);
                        response.resume(
                            Arrays.asList(
                                new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
                            )
                        );
                        return null;
                    }
                })
            );
        }
    });
}
```

The actual invocation of the request by the client (with service name `tracer-client`) and consequent invocation of the service on the server side (service name `tracer-server`) is going to generate the following sample traces (taken from Jaeger UI):

**Example #7: Client and Server with asynchronous invocation (client-side)**

In this example server-side implementation of the JAX-RS service is going to be the default one:

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks() {
    return Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
}
```

While the JAX-RS client implementation is going to perform the asynchronous invocation:

```java
final Future<Response> future = client
    .target("http://localhost:8282/books")
    .request()
    .accept(MediaType.APPLICATION_JSON)
    .async()
    .get();
```
In this respect, there is no difference from the caller prospective however a bit more work is going under the hood to transfer the active tracing span from JAX-RS client request filter to client response filter as in general those are executed in different threads (similarly to server-side asynchronous JAX-RS resource invocation). The actual invocation of the request by the client (with service name `tracer-client`) and consequent invocation of the service on the server side (service name `tracer-server`) is going to generate the following sample traces (taken from Jaeger UI):

Distributed Tracing with OpenTracing and JAX-WS support

Distributed tracing in the Apache CXF is build primarily around JAX-RS 2.x implementation. However, JAX-WS is also supported but it requires to write some boiler-plate code and use OpenTracing Java API directly (the JAX-WS integration is going to be enhanced in the future). Essentially, from the server-side prospective the in/out interceptors, `OpenTracingStartInterceptor` and `OpenTracingStopInterceptor` respectively, should be configured as part of interceptor chains, either manually or using OpenTracingFeature. For example:

```java
final Tracer tracer = new Configuration("tracer")
    .withSampler(
        new SamplerConfiguration()
            .withType(ConstSampler.TYPE) /* or any other Sampler */
            .withParam(1)
    )
    .withReporter(
        new ReporterConfiguration()
            .withSender(
                new SenderConfiguration() /* or any other Sender configuration */
                    .withEndpoint("http://localhost:14268/api/traces")
            )
    )
    .getTracer();

final JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
...
sf.getFeatures().add(new OpenTracingFeature(tracer));
...
sf.create();
```

Similarly to the server-side, client-side needs own set of out/in interceptors, `OpenTracingClientStartInterceptor` and `OpenTracingClientStopInterceptor` (or `OpenTracingClientFeature`). Please notice the difference from server-side: `OpenTracingClientStartInterceptor` becomes out-interceptor while `OpenTracingClientStopInterceptor` becomes in-interceptor. For example:

```java
final Tracer tracer = new Configuration("tracer")
    .withSampler(
        new SamplerConfiguration()
            .withType(ConstSampler.TYPE) /* or any other Sampler */
            .withParam(1)
    )
    .withReporter(
        new ReporterConfiguration()
            .withSender(
                new SenderConfiguration() /* or any other Sender configuration */
                    .withEndpoint("http://localhost:14268/api/traces")
            )
    )
    .getTracer();

final JaxWsProxyFactoryBean sf = new JaxWsProxyFactoryBean();
...
sf.getFeatures().add(new OpenTracingClientFeature(tracer));
...
sf.create();
```
As it was mentioned before, you may use GlobalTracer utility class to pass the tracer around so, for example, any JAX-WS service will be able to retrieve the current tracer by invoking GlobalTracer.get() method.

Distributed Tracing with OpenTracing and OSGi

Most of the distributed tracers compatible with OpenTracing API could be deployed into OSGi container and as such, the integration is fully available for Apache CXF services running inside the container. For a complete example please take a look on jax_rs_tracing_opentracing_osgi sample project, but here is the typical OSGi Blueprint snippet in case of Jaeger.
  <bean id="tracingFeature" class="org.apache.cxf.tracing.opentracing.jaxrs.OpenTracingFeature">
    <argument index="0">
      <bean factory-ref="withReporter" factory-method="getTracer" />
    </argument>
  </bean>
  <bean id="samplerBuilder" class="io.jaegertracing.Configuration.SamplerConfiguration" />
  <bean id="withType" factory-ref="samplerBuilder" factory-method="withType">
    <argument index="0" value="const"/>
  </bean>
  <bean id="sampler" factory-ref="withType" factory-method="withParam">
    <argument index="0">
      <bean class="java.lang.Integer">
        <argument value="1" /></bean>
    </argument>
  </bean>
  <bean id="senderBuilder" class="io.jaegertracing.Configuration.SenderConfiguration" />
  <bean id="sender" factory-ref="senderBuilder" factory-method="withEndpoint">
    <argument index="0" value="http://localhost:14268/api/traces"/>
  </bean>
  <bean id="reporterBuilder" class="io.jaegertracing.Configuration.ReporterConfiguration" />
  <bean id="reporter" factory-ref="reporterBuilder" factory-method="withSender">
    <argument index="0" ref="sender"/>
  </bean>
  <bean id="builder" class="io.jaegertracing.Configuration">
    <argument index="0" value="cxf-server" />
  </bean>
  <bean id="withSampler" factory-ref="builder" factory-method="withSampler">
    <argument index="0" ref="sampler"/>
  </bean>
  <bean id="withReporter" factory-ref="withSampler" factory-method="withReporter">
    <argument index="0" ref="reporter"/>
  </bean>
  <cxf:bus>
    <cxf:features>
      <cxf:logging />
    </cxf:features>
  </cxf:bus>
  <jaxrs:server id="catalogServer" address="/">
    <jaxrs:serviceBeans>
      ...
    </jaxrs:serviceBeans>
    <jaxrs:providers>
      <ref component-id="tracingFeature"/>
    </jaxrs:providers>
  </jaxrs:server>
</blueprint>
As of now, Jaeger tracer does not provide OSGi bundles and the service loader mechanism is not working very well. It is very likely that you may need to declare own sender configuration instance (overriding getSender method) or use system properties to pick the right one.

**Samples**

- https://github.com/apache/cxf/tree/master/distribution/src/main/release/samples/jax_rs/tracing_opentracing_camel

**Accessing OpenTracing APIs**

The Apache CXF abstracts as much of the tracer-specific APIs behind TracerContext as possible. However, sometimes there is a need to get access to OpenTracing APIs in order to leverages the rich set of available instrumentations. To make it possible, TracerContext has a dedicated unwrap method which returns underlying Tracer instance. The snippet below shows off how to use this API and use OpenTracing instrumentation for OpenFeign client.

```java
@GET
@Path(”/search”)  
@Produces(MediaType.APPLICATION_JSON)  
public JsonObject search(@QueryParam("q") final String query, @Context final TracerContext tracing) throws Exception {
    final GoogleBooksApi api = Feign.builder()
        .client(new TracingClient(new ApacheHttpClient(), tracing.unwrap(Tracer.class)))
        .target(GoogleBooksApi.class, "https://www.googleapis.com");

    final Response response = api.search(query);
    try (final Reader reader = response.body().asReader()) {
        return Json.createReader(reader).readObject();
    }
}
```