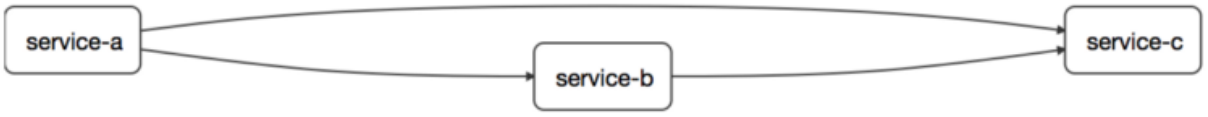
# Kibana + Zipkin

For our projects with lots of JAVA microservices, we use Zipkin to get insights into the calls that are made and where the bottlenecks might be. Once we identified an interesting trace, we would like to know what the services were doing at the time. However, we were initially missing the ability to link the Zipkin traces to the log messages of the microservices.

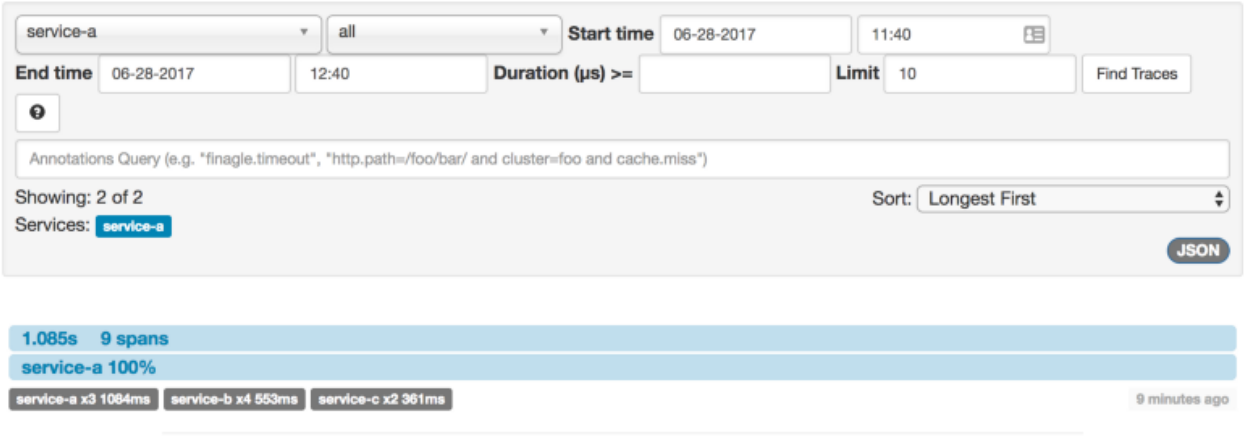
Our JAVA microservices create JSON formatted log lines using the logback LogStash encoder. They are picked up by Fluentd, put in Elasticsearch and accessed via Kibana. In this post, we want to show how to add the Zipkin traceId and spanId to the JSON formatted log lines via the Mapped Diagnostic Context (MDC). When this is done, one can select a trace from Zipkin, put the traceId in Kibana for search, and pull up all the logging associated to that particular trace.

To demonstrate how to do this, we created a sample project at merapar/zipkin-trace. It contains three microservices: a, b and c. Service a will be our entry point by means of an HTTP web service. Service a can call service b via gRPC or service c via HTTP. Service b can call service c via either HTTP or gRPC. See the following image as generated by Zipkin.

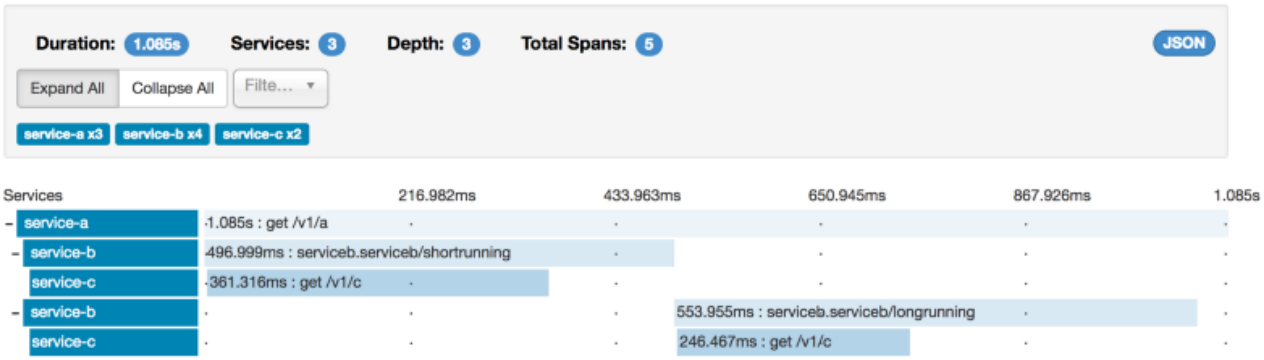



The reason for the different communication methods is to show that the propagation of the Zipkin traceId and spanId works across different protocols.

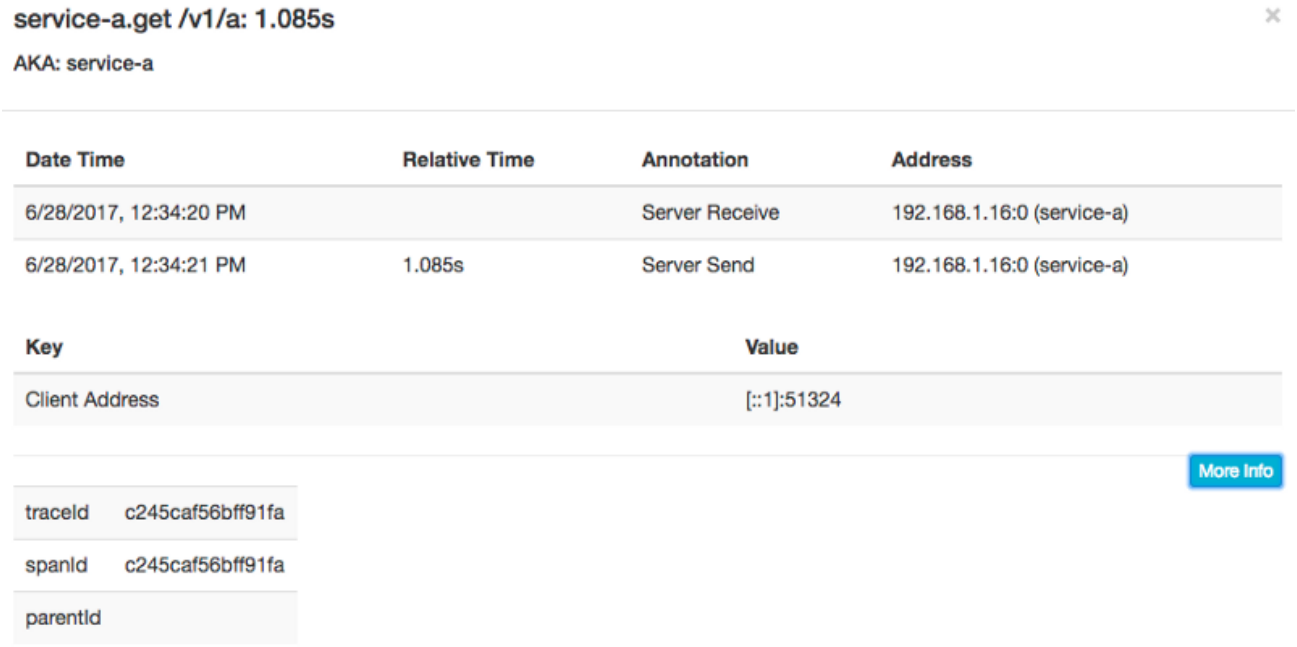
What we want to achieve is the following. If we start the three services and call http://localhost:8080/v1/a , we can see in Zipkin UI:




If we expand the trace, we get:

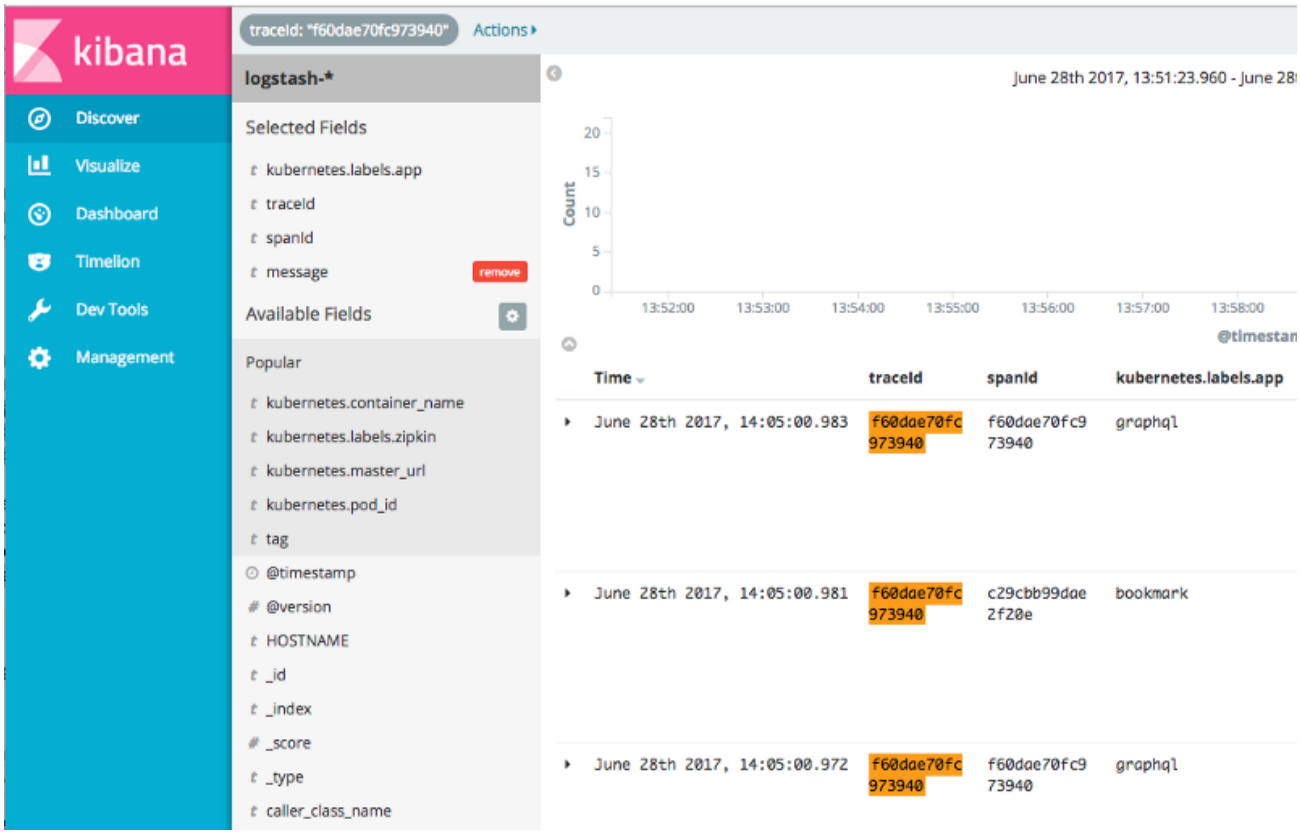



And after clicking on a span, we get another window with a ‘More Info’ button, which will reveal the traceId.




The traceId c245caf56bff91fa can then be searched for in Kibana. As an example, below is a screen-shot of Kibana using another traceId from our live system:

So we are able to find the logging that belongs to a particular Zipkin trace in Kibana, using the Zipkin traceId as search parameter. In the next section we will describe how to configure this.




The first thing that needs to be done is to setup Zipkin. We will use the openzipkin/brave library to do this. You will need at least version 4.3.4 for this to work. In common/tracing of the common project, we created the following files:

- ZipkinBraveConfiguration.java
- ZipkinGrpcServerBuilderConfigurer.java
- ZipkinHttpClientParser.java
- ZipkinHttpServerParser.java
- ZipkinWebConfiguration.java
- RestTemplateConfiguration.java
- GrpcServerInterceptorWrapper.java

We will not go over everything in the files. The key to get the spanId and traceId put in the MDC is in this section of the ZipkinBraveConfiguration.java file:

```
@Bean
public Tracing tracing() {
   val tracing = Tracing.newBuilder()
         .localServiceName(serviceName)
         .currentTraceContext(MDCCurrentTraceContext.create());

   if (zipkinEnabled) {
      tracing.reporter(reporter());
   }
   return tracing.build();
```

```
}
```

The Brave library provides an MDCCurrentTraceContext which can be set on the tracing object. We use the MDCCurrentTraceContext, because we use SLF4J as our logging framework. If you are using Log4j-2 directly, you can use ThreadContextCurrentTraceContext instead.

We are also using LogNet/grpc-spring-boot-starter, to start our gRPC services. The GrpcServerInterceptorWrapper.java file wraps Brave's serverInterceptor, so that it can be used when defining the gRPC service with the @GRpcService annotation:

```
@Slf4j
@Component
@GRpcService(interceptors = GrpcServerInterceptorWrapper.class)
public class GrpcServiceBImpl extends ServiceBGrpc.ServiceBImplBase {
...
```

Furthermore, we also like to use our own Executor to handle the incoming gRPC calls. In the ZipkinGrpcServerBuilderConfigurer.java file we set the executor for the LogNet GRpcService:

```
@Component
public class ZipkinGrpcServerBuilderConfigurer extends GRpcServerBuilderConfigurer {

   @Autowired
   private Executor executor;

   @Override
   public void configure(ServerBuilder<?> serverBuilder) {
      serverBuilder.executor(executor);
   }
}
```

With the configuration done, everything is in place to:

- Intercept incoming HTTP requests (See ZipkinWebConfiguration.java)
- Intercept outgoing HTTP requests (See RestTemplateConfiguration.java)
- Intercept incoming gRPC requests (See ZipkinBraveConfiguration.java and GrpcServerInterceptorWrapper.java)
- Intercept outgoing gRPC requests (See ZipkinBraveConfiguration.java)
- Expose the traceId and spanId in the MDC for logging.

Here is an example of the output of the test project. We configured our logback.xml to output log lines to console as follows:

```
<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
   <encoder>
      <pattern>%d [%X{traceId}/%X{spanId}] [%thread] %-5level %logger{36} - %msg%n</pattern>
      <charset>utf8</charset>
   </encoder>
</appender>
```

Now if we start the three services and call http://localhost:8080/v1/a , we can see in the logging of service a:

2017–06–28 12:34:20,038 [**c245caf56bff91fa/c245caf56bff91fa**] [http-nio-8080-exec-2] INFO zipkintrace.servicea.web.AController—Received /a request: A:HTTP -> B:gRPC -> C:HTTP
2017–06–28 12:34:20,038 [**c245caf56bff91fa/c245caf56bff91fa**] [http-nio-8080-exec-2] INFO zipkintrace.servicea.ServiceA—Calling 'shortRunning' on service B
2017–06–28 12:34:20,038 [**c245caf56bff91fa/c245caf56bff91fa**] [http-nio-8080-exec-2] INFO z.servicea.services.ServiceB—Calling 'shortRunning' on Service B via gRPC
2017–06–28 12:34:20,536 [**c245caf56bff91fa/c245caf56bff91fa**] [http-nio-8080-exec-2] INFO z.servicea.services.ServiceB—Result of 'shortRunning' on Service B: 100
2017–06–28 12:34:20,536 [**c245caf56bff91fa/c245caf56bff91fa**] [http-nio-8080-exec-2] INFO zipkintrace.servicea.ServiceA—Calling 'longRunning' on service B
2017–06–28 12:34:20,536 [**c245caf56bff91fa/c245caf56bff91fa**] [http-nio-8080-exec-2] INFO z.servicea.services.ServiceB—Calling 'longRunning' on Service B via gRPC
2017–06–28 12:34:21,090 [**c245caf56bff91fa/c245caf56bff91fa**] [http-nio-8080-exec-2] INFO z.servicea.services.ServiceB—Result of 'longRunning' on Service B: 300

From the logging we see that our traceId for this call is: c245caf56bff91fa.

Service a now calls service b twice via gRPC which can be seen in the following logging:

2017–06–28 12:34:20,041 [**c245caf56bff91fa/f238828a188aec3d**] [pool-3-thread-5] INFO z.s.grpc.serviceb.GrpcServiceBImpl—Service B called with 'shortRunning' via gRPC
2017–06–28 12:34:20,041 [**c245caf56bff91fa/f238828a188aec3d**] [pool-3-thread-5] INFO zipkintrace.serviceb.ServiceB—Calling '/c' on service C via HTTP in 'shortRunning'
2017–06–28 12:34:20,414 [**c245caf56bff91fa/f238828a188aec3d**] [pool-3-thread-5] INFO z.serviceb.services.ServiceC—Got HTTP response from service C: test call C
2017–06–28 12:34:20,538 [**c245caf56bff91fa/8b076a2d1279d9aa**] [pool-3-thread-8] INFO z.s.grpc.serviceb.GrpcServiceBImpl—Service B called with 'longRunning' via gRPC
2017–06–28 12:34:20,538 [**c245caf56bff91fa/8b076a2d1279d9aa**] [pool-3-thread-8] INFO zipkintrace.serviceb.ServiceB—Calling '/c' on service C via HTTP in 'shortRunning'
2017–06–28 12:34:20,786 [**c245caf56bff91fa/8b076a2d1279d9aa**] [pool-3-thread-8] INFO z.serviceb.services.ServiceC—Got HTTP response from service C: test call C

Note that the traceId is the same in service a logging, but the spanId is indeed different for each call. Service b now calls service c via HTTP. The logging of service c shows:
2017–06–28 12:34:20,112 [**c245caf56bff91fa/06f7caee20fcca87**] [http-nio-8100-exec-1] INFO zipkintrace.servicec.web.CController—Received /c request via HTTP
2017–06–28 12:34:20,112 [**c245caf56bff91fa/06f7caee20fcca87**] [http-nio-8100-exec-1] INFO zipkintrace.servicec.ServiceC—Running 'process' on service C
2017–06–28 12:34:20,542 [**c245caf56bff91fa/74de642754d2d3b3**] [http-nio-8100-exec-2] INFO zipkintrace.servicec.web.CController—Received /c request via HTTP
2017–06–28 12:34:20,542 [**c245caf56bff91fa/74de642754d2d3b3**] [http-nio-8100-exec-2] INFO zipkintrace.servicec.ServiceC—Running 'process' on service C

Again, the traceId is nicely propagated to service c as well. This traceId was used in the screenshots at the beginning of the article.

Our JAVA microservices are using the configuration described above to log everything in JSON format (using the net.logstash.logback.encoder.LogstashEncoder), which automatically includes all MDC values as key - value pairs. With this, if we find interesting traces in Zipkin, we can now take the traceId and search for it in Kibana. When we have retrieved the log lines associated with the traceId, we can figure out exactly what is happening in the system.