# Spring Cloud Sleuth

Spring Cloud Sleuth is a distributed tracing tool for Spring Cloud. It borrows from [Dapper](http://research.google.com/pubs/pub36356.html), [Zipkin](https://github.com/openzipkin/zipkin), and [HTrace](http://htrace.incubator.apache.org/).

## Quick Start

Add sleuth to the classpath of a Spring Boot application (see “[Adding Sleuth to the Project](https://github.com/spring-cloud/spring-cloud-sleuth#sleuth-adding-project)” for Maven and Gradle examples), and you can see the correlation data being collected in logs, as long as you are logging requests.

For example, consider the following HTTP handler:

@RestController

public class DemoController {

private static Logger log = LoggerFactory.getLogger(DemoController.class);

@RequestMapping("/")

public String home() {

log.info("Handling home");

...

return "Hello World";

}

}

If you add that handler to a controller, you can see the calls to home() being traced in the logs and in Zipkin, if Zipkin is configured.

|  |  |
| --- | --- |
| Note | Instead of logging the request in the handler explicitly, you could set logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG. |
| Note | Set spring.application.name=myService (for instance) to see the service name as well as the trace and span IDs. |
| Important | If you use Zipkin, configure the probability of spans exported by setting spring.sleuth.sampler.probability (default: 0.1, which is 10 percent). Otherwise, you might think that Sleuth is not working be cause it omits some spans. |

## Terminology

**Span**: The basic unit of work. For example, sending an RPC is a new span, as is sending a response to an RPC. Spans are identified by a unique 64-bit ID for the span and another 64-bit ID for the trace the span is a part of. Spans also have other data, such as descriptions, timestamped events, key-value annotations (tags), the ID of the span that caused them, and process IDs (normally IP addresses).

Spans can be started and stopped, and they keep track of their timing information. Once you create a span, you must stop it at some point in the future.

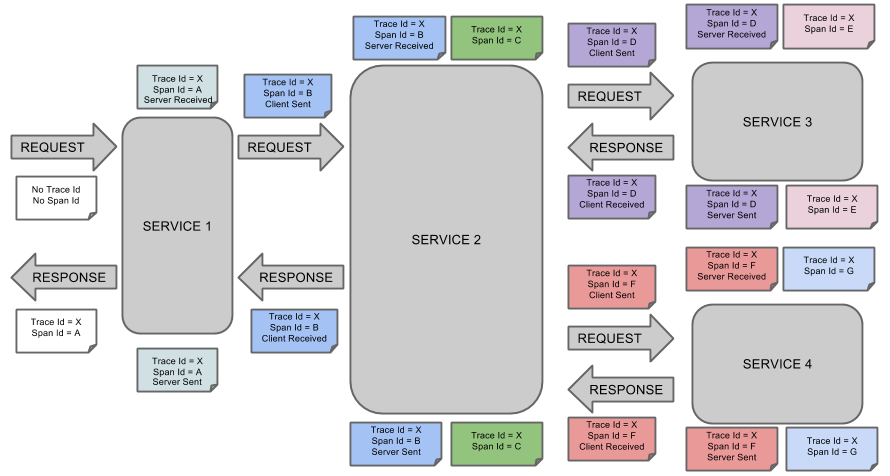
|  |  |
| --- | --- |
| Tip | The initial span that starts a trace is called a root span. The value of the ID of that span is equal to the trace ID. |

**Trace:** A set of spans forming a tree-like structure. For example, if you run a distributed big-data store, a trace might be formed by a PUT request.

**Annotation:** Used to record the existence of an event in time. With [Brave](https://github.com/openzipkin/brave) instrumentation, we no longer need to set special events for [Zipkin](https://zipkin.io/) to understand who the client and server are, where the request started, and where it ended. For learning purposes, however, we mark these events to highlight what kind of an action took place.

* **cs**: Client Sent. The client has made a request. This annotation indicates the start of the span.
* **sr**: Server Received: The server side got the request and started processing it. Subtracting the cs timestamp from this timestamp reveals the network latency.
* **ss**: Server Sent. Annotated upon completion of request processing (when the response got sent back to the client). Subtracting the sr timestamp from this timestamp reveals the time needed by the server side to process the request.
* **cr**: Client Received. Signifies the end of the span. The client has successfully received the response from the server side. Subtracting the cs timestamp from this timestamp reveals the whole time needed by the client to receive the response from the server.

The following image shows how **Span** and **Trace** look in a system, together with the Zipkin annotations:

[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/trace-id.png)

Each color of a note signifies a span (there are seven spans - from **A** to **G**). Consider the following note:

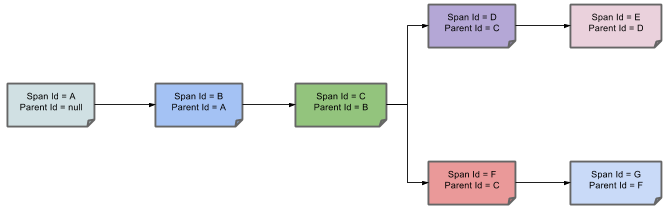
Trace Id = X

Span Id = D

Client Sent

This note indicates that the current span has **Trace Id** set to **X** and **Span Id** set to **D**. Also, the Client Sent event took place.

The following image shows how parent-child relationships of spans look:

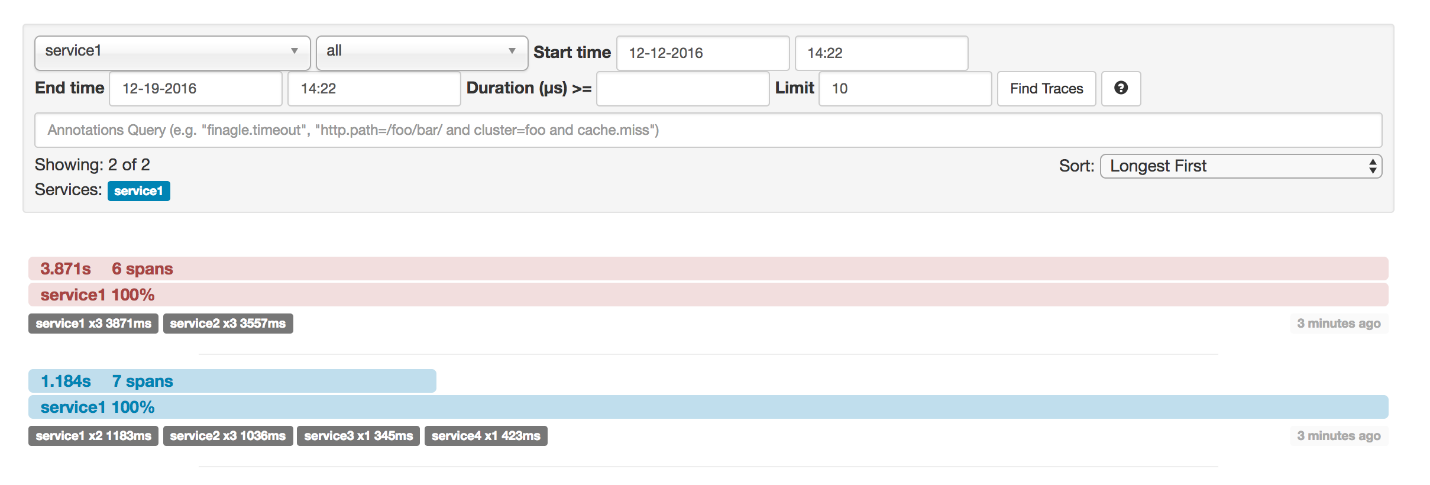
[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/parents.png)

## Purpose

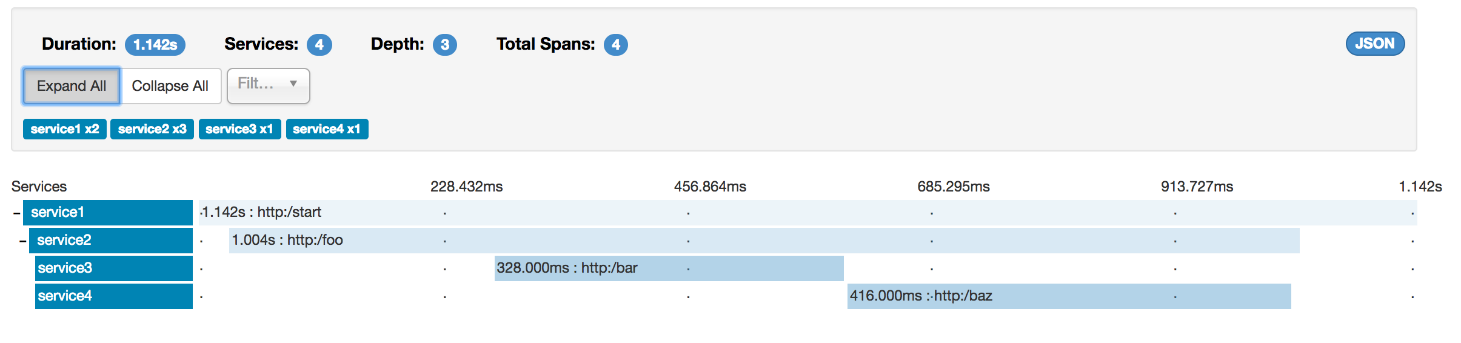
The following sections refer to the example shown in the preceding image.

### Distributed Tracing with Zipkin

This example has seven spans. If you go to traces in Zipkin, you can see this number in the second trace, as shown in the following image:

[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/zipkin-traces.png)

However, if you pick a particular trace, you can see four spans, as shown in the following image:

[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/zipkin-ui.png)

|  |  |
| --- | --- |
| Note | When you pick a particular trace, you see merged spans. That means that, if there were two spans sent to Zipkin with Server Received and Server Sent or Client Received and Client Sent annotations, they are presented as a single span. |

Why is there a difference between the seven and four spans in this case?

* Two spans come from the http:/start span. It has the Server Received (sr) and Server Sent (ss) annotations.
* Two spans come from the RPC call from service1 to service2 to the http:/foo endpoint. The Client Sent (cs) and Client Received (cr) events took place on the service1 side. Server Received (sr) and Server Sent (ss) events took place on the service2 side. These two spans form one logical span related to an RPC call.
* Two spans come from the RPC call from service2 to service3 to the http:/bar endpoint. The Client Sent (cs) and Client Received (cr) events took place on the service2 side. The Server Received (sr) and Server Sent (ss) events took place on the service3 side. These two spans form one logical span related to an RPC call.
* Two spans come from the RPC call from service2 to service4 to the http:/baz endpoint. The Client Sent (cs) and Client Received (cr) events took place on the service2 side. Server Received (sr) and Server Sent (ss) events took place on the service4 side. These two spans form one logical span related to an RPC call.

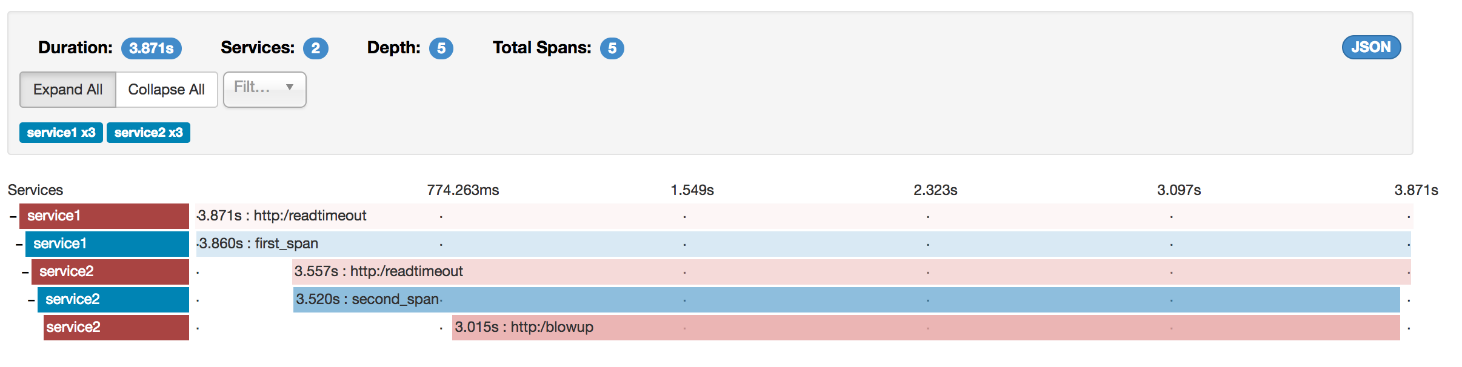
So, if we count the physical spans, we have one from http:/start, two from service1 calling service2, two from service2 calling service3, and two from service2 calling service4. In sum, we have a total of seven spans.

Logically, we see the information of four total Spans because we have one span related to the incoming request to service1and three spans related to RPC calls.

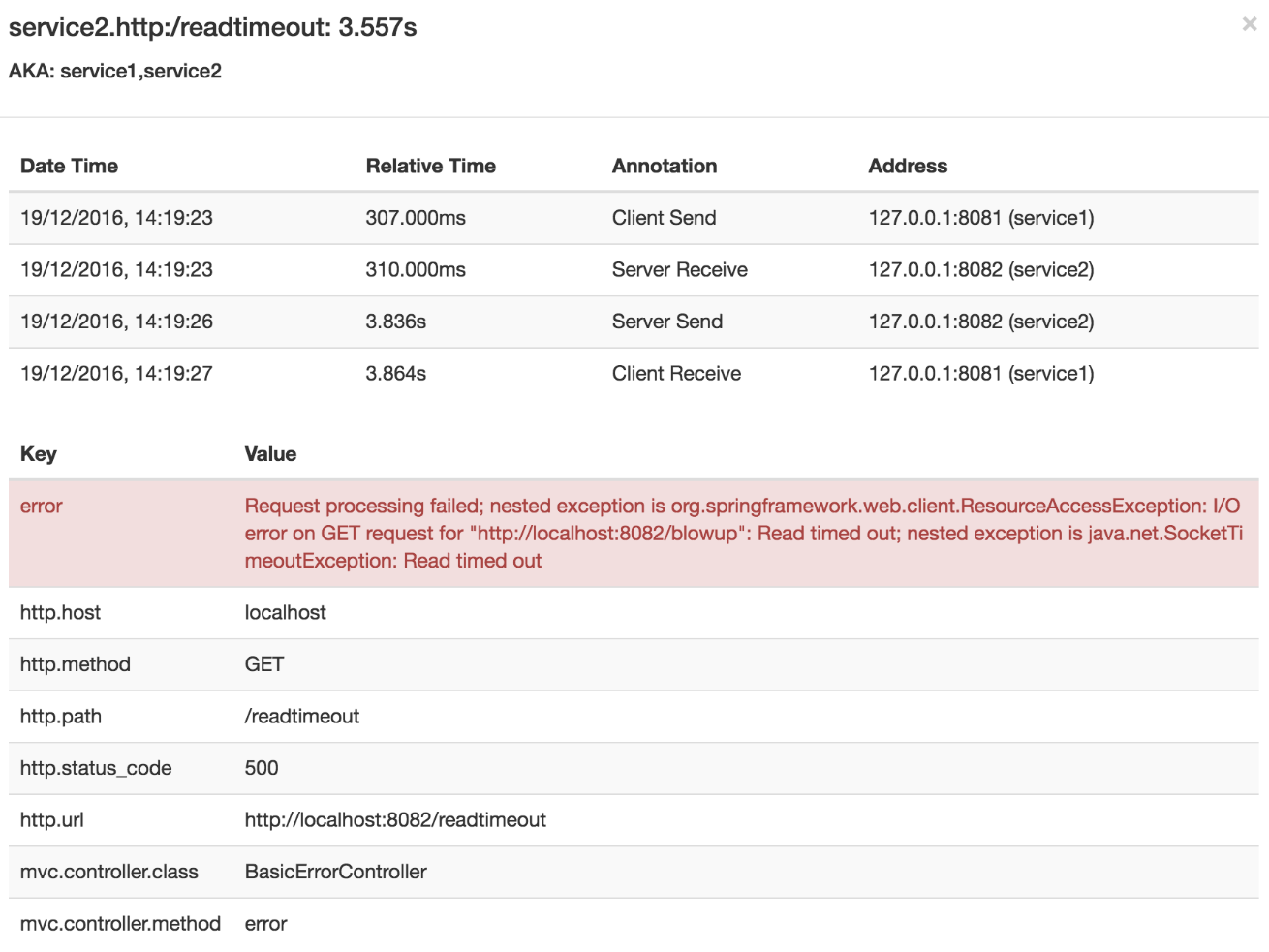
### Visualizing errors

Zipkin lets you visualize errors in your trace. When an exception was thrown and was not caught, we set proper tags on the span, which Zipkin can then properly colorize. You could see in the list of traces one trace that is red. That appears because an exception was thrown.

If you click that trace, you see a similar picture, as follows:

[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/zipkin-error-traces.png)

If you then click on one of the spans, you see the following

[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/zipkin-error-trace-screenshot.png)

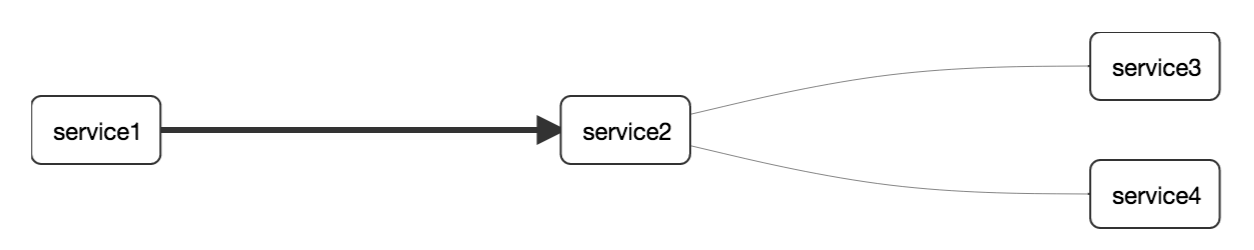
The span shows the reason for the error and the whole stack trace related to it.

### Distributed Tracing with Brave

Starting with version 2.0.0, Spring Cloud Sleuth uses [Brave](https://github.com/openzipkin/brave) as the tracing library. Consequently, Sleuth no longer takes care of storing the context but delegates that work to Brave.

Due to the fact that Sleuth had different naming and tagging conventions than Brave, we decided to follow Brave’s conventions from now on. However, if you want to use the legacy Sleuth approaches, you can set the spring.sleuth.http.legacy.enabledproperty to true.

### Live examples

[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/dependencies.png)

## Log correlation

When using grep to read the logs of those four applications by scanning for a trace ID equal to (for example) 2485ec27856c56f4, you get output resembling the following:

service1.log:2016-02-26 11:15:47.561 INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.Application : Hello from service1. Calling service2

service2.log:2016-02-26 11:15:47.710 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Hello from service2. Calling service3 and then service4

service3.log:2016-02-26 11:15:47.895 INFO [service3,2485ec27856c56f4,1210be13194bfe5,true] 68060 --- [nio-8083-exec-1] i.s.c.sleuth.docs.service3.Application : Hello from service3

service2.log:2016-02-26 11:15:47.924 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Got response from service3 [Hello from service3]

service4.log:2016-02-26 11:15:48.134 INFO [service4,2485ec27856c56f4,1b1845262ffba49d,true] 68061 --- [nio-8084-exec-1] i.s.c.sleuth.docs.service4.Application : Hello from service4

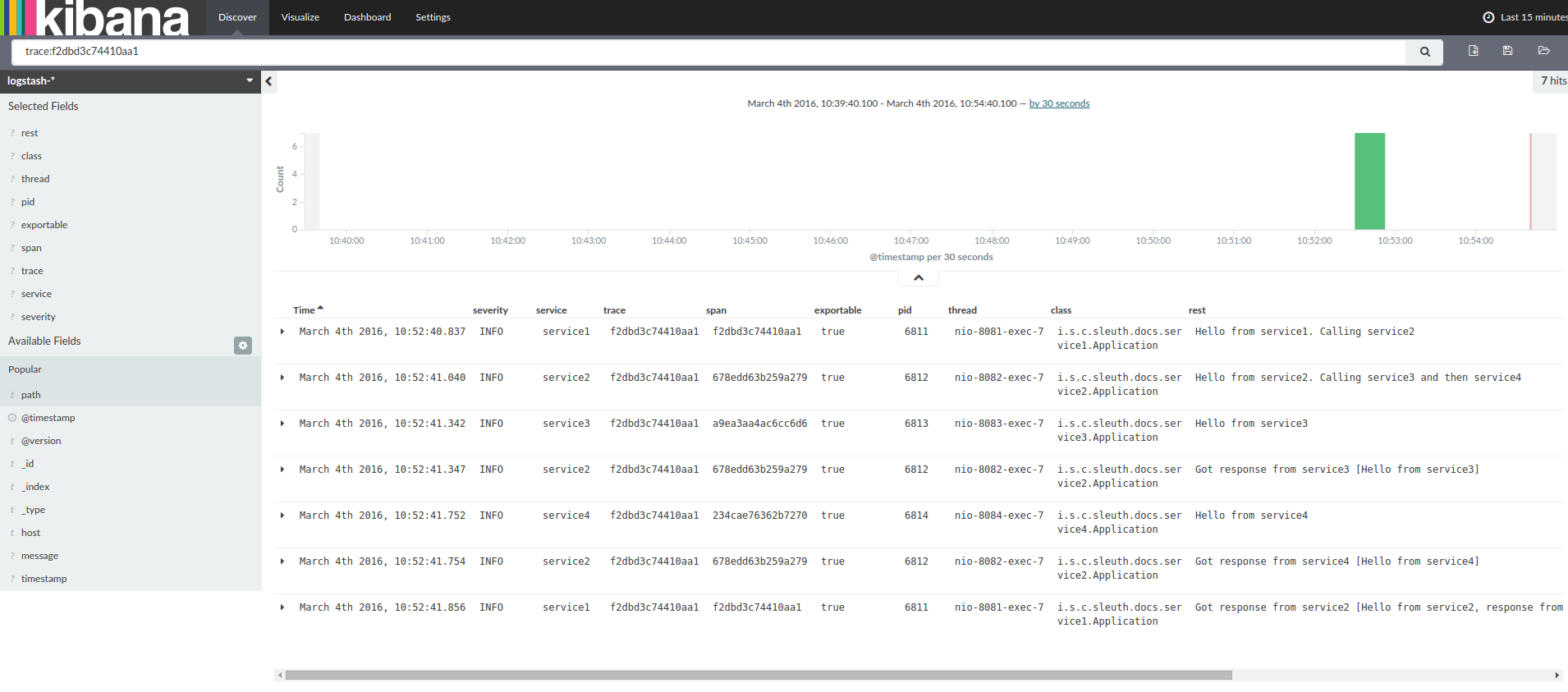
service2.log:2016-02-26 11:15:48.156 INFO

[service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Got response from service4 [Hello from service4]

service1.log:2016-02-26 11:15:48.182 INFO

[service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.Application : Got response from service2 [Hello from service2, response from service3 [Hello from service3] and from service4 [Hello from service4]]

If you use a log aggregating tool (such as [Kibana](https://www.elastic.co/products/kibana), [Splunk](http://www.splunk.com/), and others), you can order the events that took place. An example from Kibana would resemble the following image:

[](https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/kibana.png)

If you want to use [Logstash](https://www.elastic.co/guide/en/logstash/current/index.html), the following listing shows the Grok pattern for Logstash:

filter {

# pattern matching logback pattern

grok {

match => { "message" => "%{TIMESTAMP\_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }

}

}

|  |  |
| --- | --- |
| Note | If you want to use Grok together with the logs from Cloud Foundry, you have to use the following pattern: |

filter {

# pattern matching logback pattern

grok {

match => { "message" => "(?m)OUT\s+%{TIMESTAMP\_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }

}

}

### JSON Logback with Logstash

Often, you do not want to store your logs in a text file but in a JSON file that Logstash can immediately pick. To do so, you have to do the following (for readability, we pass the dependencies in the groupId:artifactId:version notation).

#### **Dependencies Setup**

1. Ensure that Logback is on the classpath (ch.qos.logback:logback-core).
2. Add Logstash Logback encode. For example, to use version 4.6, add net.logstash.logback:logstash-logback-encoder:4.6.

#### **Logback Setup**

Consider the following example of a Logback configuration file (named [logback-spring.xml](https://github.com/spring-cloud-samples/sleuth-documentation-apps/blob/master/service1/src/main/resources/logback-spring.xml)).

<?xml version="1.0" encoding="UTF-8"?>

<configuration>

<include resource="org/springframework/boot/logging/logback/defaults.xml"/>

​

<springProperty scope="context" name="springAppName" source="spring.application.name"/>

<!-- Example for logging into the build folder of your project -->

<property name="LOG\_FILE" value="${BUILD\_FOLDER:-build}/${springAppName}"/>​

<!-- You can override this to have a custom pattern -->

<property name="CONSOLE\_LOG\_PATTERN"

value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}"/>

<!-- Appender to log to console -->

<appender name="console" class="ch.qos.logback.core.ConsoleAppender">

<filter class="ch.qos.logback.classic.filter.ThresholdFilter">

<!-- Minimum logging level to be presented in the console logs-->

<level>DEBUG</level>

</filter>

<encoder>

<pattern>${CONSOLE\_LOG\_PATTERN}</pattern>

<charset>utf8</charset>

</encoder>

</appender>

<!-- Appender to log to file -->​

<appender name="flatfile" class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>${LOG\_FILE}</file>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<fileNamePattern>${LOG\_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>

<maxHistory>7</maxHistory>

</rollingPolicy>

<encoder>

<pattern>${CONSOLE\_LOG\_PATTERN}</pattern>

<charset>utf8</charset>

</encoder>

</appender>

​

<!-- Appender to log to file in a JSON format -->

<appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>${LOG\_FILE}.json</file>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<fileNamePattern>${LOG\_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>

<maxHistory>7</maxHistory>

</rollingPolicy>

<encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">

<providers>

<timestamp>

<timeZone>UTC</timeZone>

</timestamp>

<pattern>

<pattern>

{

"severity": "%level",

"service": "${springAppName:-}",

"trace": "%X{X-B3-TraceId:-}",

"span": "%X{X-B3-SpanId:-}",

"parent": "%X{X-B3-ParentSpanId:-}",

"exportable": "%X{X-Span-Export:-}",

"pid": "${PID:-}",

"thread": "%thread",

"class": "%logger{40}",

"rest": "%message"

}

</pattern>

</pattern>

</providers>

</encoder>

</appender>

​

<root level="INFO">

<appender-ref ref="console"/>

<!-- uncomment this to have also JSON logs -->

<!--<appender-ref ref="logstash"/>-->

<!--<appender-ref ref="flatfile"/>-->

</root>

</configuration>

That Logback configuration file:

* Logs information from the application in a JSON format to a build/${spring.application.name}.json file.
* Has commented out two additional appenders: console and standard log file.
* Has the same logging pattern as the one presented in the previous section.

|  |  |
| --- | --- |
| Note | If you use a custom logback-spring.xml, you must pass the spring.application.name in the bootstrap rather than the application property file. Otherwise, your custom logback file does not properly read the property. |

## Propagating Span Context

The span context is the state that must get propagated to any child spans across process boundaries. Part of the Span Context is the Baggage. The trace and span IDs are a required part of the span context. Baggage is an optional part.

Baggage is a set of key:value pairs stored in the span context. Baggage travels together with the trace and is attached to every span. Spring Cloud Sleuth understands that a header is baggage-related if the HTTP header is prefixed with baggage- and, for messaging, it starts with baggage\_.

|  |  |
| --- | --- |
| Important | There is currently no limitation of the count or size of baggage items. However, keep in mind that too many can decrease system throughput or increase RPC latency. In extreme cases, too much baggage can crash the application, due to exceeding transport-level message or header capacity. |

The following example shows setting baggage on a span:

Span initialSpan = this.tracer.nextSpan().name("span").start();

ExtraFieldPropagation.set(initialSpan.context(), "foo", "bar");

ExtraFieldPropagation.set(initialSpan.context(), "UPPER\_CASE", "someValue");

}

### Baggage versus Span Tags

Baggage travels with the trace (every child span contains the baggage of its parent). Zipkin has no knowledge of baggage and does not receive that information.

|  |  |
| --- | --- |
| Important | Starting from Sleuth 2.0.0 you have to pass the baggage key names explicitly in your project configuration. Read more about that setup [here](https://github.com/spring-cloud/spring-cloud-sleuth#prefixed-fields) |

Tags are attached to a specific span. In other words, they are presented only for that particular span. However, you can search by tag to find the trace, assuming a span having the searched tag value exists.

If you want to be able to lookup a span based on baggage, you should add a corresponding entry as a tag in the root span.

|  |  |
| --- | --- |
| Important | The span must be in scope. |

The following listing shows integration tests that use baggage:

The setup

spring.sleuth:

baggage-keys:

- baz

- bizarrecase

propagation-keys:

- foo

- upper\_case

The code

initialSpan.tag("foo",

ExtraFieldPropagation.get(initialSpan.context(), "foo"));

initialSpan.tag("UPPER\_CASE",

ExtraFieldPropagation.get(initialSpan.context(), "UPPER\_CASE"));

## Adding Sleuth to the Project

This section addresses how to add Sleuth to your project with either Maven or Gradle.

|  |  |
| --- | --- |
| Important | To ensure that your application name is properly displayed in Zipkin, set the spring.application.nameproperty in bootstrap.yml. |

### Only Sleuth (log correlation)

If you want to use only Spring Cloud Sleuth without the Zipkin integration, add the spring-cloud-starter-sleuth module to your project.

The following example shows how to add Sleuth with Maven:

Maven

<dependencyManagement> (1)

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${release.train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependency> (2)

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

1. We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
2. Add the dependency to spring-cloud-starter-sleuth.

The following example shows how to add Sleuth with Gradle:

Gradle

dependencyManagement { (1)

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"

}

}

dependencies { (2)

compile "org.springframework.cloud:spring-cloud-starter-sleuth"

}

1. We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
2. Add the dependency to spring-cloud-starter-sleuth.

### Sleuth with Zipkin via HTTP

If you want both Sleuth and Zipkin, add the spring-cloud-starter-zipkin dependency.

The following example shows how to do so for Maven:

Maven

<dependencyManagement> (1)

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${release.train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependency> (2)

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

1. We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
2. Add the dependency to spring-cloud-starter-zipkin.

The following example shows how to do so for Gradle:

Gradle

dependencyManagement { (1)

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"

}

}

dependencies { (2)

compile "org.springframework.cloud:spring-cloud-starter-zipkin"

}

1. We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
2. Add the dependency to spring-cloud-starter-zipkin.

### Sleuth with Zipkin over RabbitMQ or Kafka

If you want to use RabbitMQ or Kafka instead of HTTP, add the spring-rabbit or spring-kafka dependency. The default destination name is zipkin.

If using Kafka, you must set the property spring.zipkin.sender.type property accordingly:

spring.zipkin.sender.type: kafka

|  |  |
| --- | --- |
| Caution | spring-cloud-sleuth-stream is deprecated and incompatible with these destinations. |

If you want Sleuth over RabbitMQ, add the spring-cloud-starter-zipkin and spring-rabbit dependencies.

The following example shows how to do so for Gradle:

Maven

<dependencyManagement> (1)

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${release.train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependency> (2)

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

<dependency> (3)

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

</dependency>

1. We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
2. Add the dependency to spring-cloud-starter-zipkin. That way, all nested dependencies get downloaded.
3. To automatically configure RabbitMQ, add the spring-rabbit dependency.

Gradle

dependencyManagement { (1)

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"

}

}

dependencies {

compile "org.springframework.cloud:spring-cloud-starter-zipkin" (2)

compile "org.springframework.amqp:spring-rabbit" (3)

}

1. We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
2. Add the dependency to spring-cloud-starter-zipkin. That way, all nested dependencies get downloaded.
3. To automatically configure RabbitMQ, add the spring-rabbit dependency.

## Additional Resources

You can watch a video of [Reshmi Krishna](https://twitter.com/reshmi9k) and [Marcin Grzejszczak](https://twitter.com/mgrzejszczak) talking about Spring Cloud Sleuth and Zipkin [by clicking here](https://content.pivotal.io/springone-platform-2017/distributed-tracing-latency-analysis-for-your-microservices-grzejszczak-krishna).

You can check different setups of Sleuth and Brave [in the openzipkin/sleuth-webmvc-example repository](https://github.com/openzipkin/sleuth-webmvc-example).

## Features

* Adds trace and span IDs to the Slf4J MDC, so you can extract all the logs from a given trace or span in a log aggregator, as shown in the following example logs:
* 2016-02-02 15:30:57.902 INFO [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
* 2016-02-02 15:30:58.372 ERROR [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...

2016-02-02 15:31:01.936 INFO [bar,46ab0d418373cbc9,46ab0d418373cbc9,false] 23030 --- [nio-8081-exec-4] ...

Notice the [appname,traceId,spanId,exportable] entries from the MDC:

* + **spanId**: The ID of a specific operation that took place.
  + **appname**: The name of the application that logged the span.
  + **traceId**: The ID of the latency graph that contains the span.
  + **exportable**: Whether the log should be exported to Zipkin. When would you like the span not to be exportable? When you want to wrap some operation in a Span and have it written to the logs only.
* Provides an abstraction over common distributed tracing data models: traces, spans (forming a DAG), annotations, and key-value annotations. Spring Cloud Sleuth is loosely based on HTrace but is compatible with Zipkin (Dapper).
* Sleuth records timing information to aid in latency analysis. By using sleuth, you can pinpoint causes of latency in your applications.
* Sleuth is written to not log too much and to not cause your production application to crash. To that end, Sleuth:
  + Propagates structural data about your call graph in-band and the rest out-of-band.
  + Includes opinionated instrumentation of layers such as HTTP.
  + Includes a sampling policy to manage volume.
  + Can report to a Zipkin system for query and visualization.
* Instruments common ingress and egress points from Spring applications (servlet filter, async endpoints, rest template, scheduled actions, message channels, Zuul filters, and Feign client).
* Sleuth includes default logic to join a trace across HTTP or messaging boundaries. For example, HTTP propagation works over Zipkin-compatible request headers.
* Sleuth can propagate context (also known as baggage) between processes. Consequently, if you set a baggage element on a Span, it is sent downstream to other processes over either HTTP or messaging.
* Provides a way to create or continue spans and add tags and logs through annotations.
* If spring-cloud-sleuth-zipkin is on the classpath, the app generates and collects Zipkin-compatible traces. By default, it sends them over HTTP to a Zipkin server on localhost (port 9411). You can configure the location of the service by setting spring.zipkin.baseUrl.
  + If you depend on spring-rabbit, your app sends traces to a RabbitMQ broker instead of HTTP.
  + If you depend on spring-kafka, and set spring.zipkin.sender.type: kafka, your app sends traces to a Kafka broker instead of HTTP.

|  |  |
| --- | --- |
| Caution | spring-cloud-sleuth-stream is deprecated and should no longer be used. |

* Spring Cloud Sleuth is [OpenTracing](http://opentracing.io/) compatible.

|  |  |
| --- | --- |
| Important | If you use Zipkin, configure the probability of spans exported by setting spring.sleuth.sampler.probability(default: 0.1, which is 10 percent). Otherwise, you might think that Sleuth is not working be cause it omits some spans. |
| Note | The SLF4J MDC is always set and logback users immediately see the trace and span IDs in logs per the example shown earlier. Other logging systems have to configure their own formatter to get the same result. The default is as follows: logging.pattern.level set to %5p [${spring.zipkin.service.name:${spring.application.name:-}},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}] (this is a Spring Boot feature for logback users). If you do not use SLF4J, this pattern is NOT automatically applied. |

# Zipkin

## Architecture

### Architecture Overview

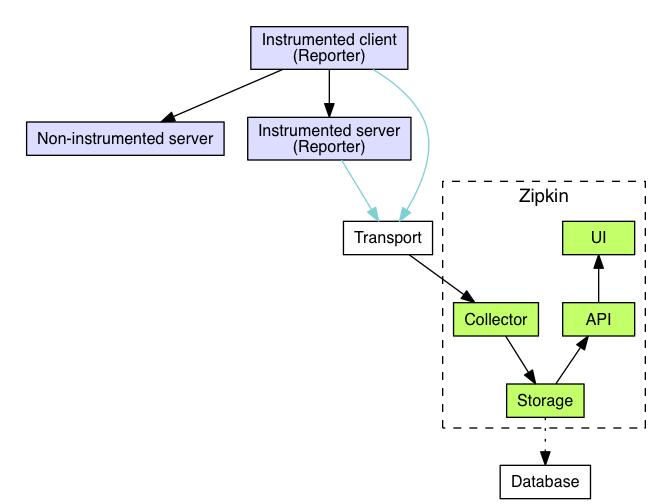
Tracers live in your applications and record timing and metadata about operations that took place. They often instrument libraries, so that their use is transparent to users. For example, an instrumented web server records when it received a request and when it sent a response. The trace data collected is called a Span.

Instrumentation is written to be safe in production and have little overhead. For this reason, they only propagate IDs in-band, to tell the receiver there’s a trace in progress. Completed spans are reported to Zipkin out-of-band, similar to how applications report metrics asynchronously.

For example, when an operation is being traced and it needs to make an outgoing http request, a few headers are added to propagate IDs. Headers are not used to send details such as the operation name.

The component in an instrumented app that sends data to Zipkin is called a Reporter. Reporters send trace data via one of several transports to Zipkin collectors, which persist trace data to storage. Later, storage is queried by the API to provide data to the UI.

Here’s a diagram describing this flow:



To see if an instrumentation library already exists for your platform, see the list of [existing instrumentations](https://zipkin.io/pages/existing_instrumentations).

### Example flow

As mentioned in the overview, identifiers are sent in-band and details are sent out-of-band to Zipkin. In both cases, trace instrumentation is responsible for creating valid traces and rendering them properly. For example, a tracer ensures parity between the data it sends in-band (downstream) and out-of-band (async to Zipkin).

Here’s an example sequence of http tracing where user code calls the resource /foo. This results in a single span, sent asynchronously to Zipkin after user code receives the http response.

┌─────────────┐ ┌───────────────────────┐ ┌─────────────┐ ┌──────────────────┐

│ User Code │ │ Trace Instrumentation │ │ Http Client │ │ Zipkin Collector │

└─────────────┘ └───────────────────────┘ └─────────────┘ └──────────────────┘

│ │ │ │

┌─────────┐

│ ──┤GET /foo ├─▶ │ ────┐ │ │

└─────────┘ │ record tags

│ │ ◀───┘ │ │

────┐

│ │ │ add trace headers │ │

◀───┘

│ │ ────┐ │ │

│ record timestamp

│ │ ◀───┘ │ │

┌─────────────────┐

│ │ ──┤GET /foo ├─▶ │ │

│X-B3-TraceId: aa │ ────┐

│ │ │X-B3-SpanId: 6b │ │ │ │

└─────────────────┘ │ invoke

│ │ │ │ request │

│

│ │ │ │ │

┌────────┐ ◀───┘

│ │ ◀─────┤200 OK ├─────── │ │

────┐ └────────┘

│ │ │ record duration │ │

┌────────┐ ◀───┘

│ ◀──┤200 OK ├── │ │ │

└────────┘ ┌────────────────────────────────┐

│ │ ──┤ asynchronously report span ├────▶ │

│ │

│{ │

│ "traceId": "aa", │

│ "id": "6b", │

│ "name": "get", │

│ "timestamp": 1483945573944000,│

│ "duration": 386000, │

│ "annotations": [ │

│--snip-- │

└────────────────────────────────┘

Trace instrumentation report spans asynchronously to prevent delays or failures relating to the tracing system from delaying or breaking user code.

### Transport

Spans sent by the instrumented library must be transported from the services being traced to Zipkin collectors. There are three primary transports: HTTP, Kafka and Scribe.

### Components

There are 4 components that make up Zipkin:

* collector
* storage
* search
* web UI

#### Zipkin Collector

Once the trace data arrives at the Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.

#### Storage

Zipkin was initially built to store data on Cassandra since Cassandra is scalable, has a flexible schema, and is heavily used within Twitter. However, we made this component pluggable. In addition to Cassandra, we natively support ElasticSearch and MySQL. Other back-ends might be offered as third party extensions.

#### Zipkin Query Service

Once the data is stored and indexed, we need a way to extract it. The query daemon provides a simple JSON API for finding and retrieving traces. The primary consumer of this API is the Web UI.

#### Web UI

We created a GUI that presents a nice interface for viewing traces. The web UI provides a method for viewing traces based on service, time, and annotations. Note: there is no built-in authentication in the UI!

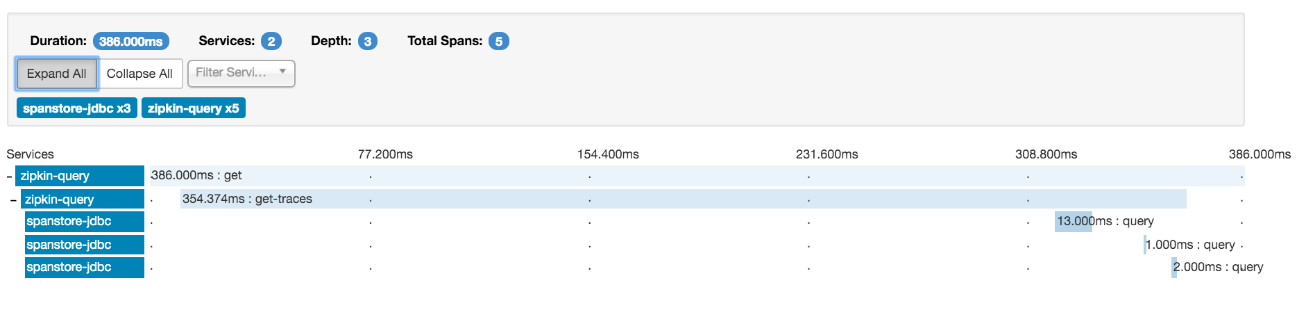
## Data Model

*Note* this page is out-of-date. Please view the [Zipkin Api](https://zipkin.io/zipkin-api/#/default/post_spans) doc which elaborates the fields in the model until this page is updated.

In order to illustrate the tracing data that Zipkin displays, let’s relate it to the equivalent information in the Zipkin data model. By comparing these, we see that

* inbound and outbound requests are in different spans
* spans that include cs can log an sa annotation of where they are going
  + This helps when the destination protocol isn’t Zipkin instrumented, such as MySQL.

First, we see one trace as shown in the Zipkin trace viewer:



And the same trace in the data model of Zipkin:

[

{

"traceId": "bd7a977555f6b982",

"name": "get",

"id": "bd7a977555f6b982",

"timestamp": 1458702548467000,

"duration": 386000,

"annotations": [

{

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

},

"timestamp": 1458702548467000,

"value": "sr"

},

{

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

},

"timestamp": 1458702548853000,

"value": "ss"

}

],

"binaryAnnotations": []

},

{

"traceId": "bd7a977555f6b982",

"name": "get-traces",

"id": "ebf33e1a81dc6f71",

"parentId": "bd7a977555f6b982",

"timestamp": 1458702548478000,

"duration": 354374,

"annotations": [],

"binaryAnnotations": [

{

"key": "lc",

"value": "JDBCSpanStore",

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

}

},

{

"key": "request",

"value": "QueryRequest{serviceName=zipkin-query, spanName=null, annotations=[], binaryAnnotations={}, minDuration=null, maxDuration=null, endTs=1458702548478, lookback=86400000, limit=1}",

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

}

}

]

},

{

"traceId": "bd7a977555f6b982",

"name": "query",

"id": "be2d01e33cc78d97",

"parentId": "ebf33e1a81dc6f71",

"timestamp": 1458702548786000,

"duration": 13000,

"annotations": [

{

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

},

"timestamp": 1458702548786000,

"value": "cs"

},

{

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

},

"timestamp": 1458702548799000,

"value": "cr"

}

],

"binaryAnnotations": [

{

"key": "jdbc.query",

"value": "select distinct `zipkin\_spans`.`trace\_id` from `zipkin\_spans` join `zipkin\_annotations` on (`zipkin\_spans`.`trace\_id` = `zipkin\_annotations`.`trace\_id` and `zipkin\_spans`.`id` = `zipkin\_annotations`.`span\_id`) where (`zipkin\_annotations`.`endpoint\_service\_name` = ? and `zipkin\_spans`.`start\_ts` between ? and ?) order by `zipkin\_spans`.`start\_ts` desc limit ?",

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

}

},

{

"key": "sa",

"value": true,

"endpoint": {

"serviceName": "spanstore-jdbc",

"ipv4": "127.0.0.1",

"port": 3306

}

}

]

},

{

"traceId": "bd7a977555f6b982",

"name": "query",

"id": "13038c5fee5a2f2e",

"parentId": "ebf33e1a81dc6f71",

"timestamp": 1458702548817000,

"duration": 1000,

"annotations": [

{

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

},

"timestamp": 1458702548817000,

"value": "cs"

},

{

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

},

"timestamp": 1458702548818000,

"value": "cr"

}

],

"binaryAnnotations": [

{

"key": "jdbc.query",

"value": "select `zipkin\_spans`.`trace\_id`, `zipkin\_spans`.`id`, `zipkin\_spans`.`name`, `zipkin\_spans`.`parent\_id`, `zipkin\_spans`.`debug`, `zipkin\_spans`.`start\_ts`, `zipkin\_spans`.`duration` from `zipkin\_spans` where `zipkin\_spans`.`trace\_id` in (?)",

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

}

},

{

"key": "sa",

"value": true,

"endpoint": {

"serviceName": "spanstore-jdbc",

"ipv4": "127.0.0.1",

"port": 3306

}

}

]

},

{

"traceId": "bd7a977555f6b982",

"name": "query",

"id": "37ee55f3d3a94336",

"parentId": "ebf33e1a81dc6f71",

"timestamp": 1458702548827000,

"duration": 2000,

"annotations": [

{

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

},

"timestamp": 1458702548827000,

"value": "cs"

},

{

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

},

"timestamp": 1458702548829000,

"value": "cr"

}

],

"binaryAnnotations": [

{

"key": "jdbc.query",

"value": "select `zipkin\_annotations`.`trace\_id`, `zipkin\_annotations`.`span\_id`, `zipkin\_annotations`.`a\_key`, `zipkin\_annotations`.`a\_value`, `zipkin\_annotations`.`a\_type`, `zipkin\_annotations`.`a\_timestamp`, `zipkin\_annotations`.`endpoint\_ipv4`, `zipkin\_annotations`.`endpoint\_port`, `zipkin\_annotations`.`endpoint\_service\_name` from `zipkin\_annotations` where `zipkin\_annotations`.`trace\_id` in (?) order by `zipkin\_annotations`.`a\_timestamp` asc, `zipkin\_annotations`.`a\_key` asc",

"endpoint": {

"serviceName": "zipkin-query",

"ipv4": "192.168.1.2",

"port": 9411

}

},

{

"key": "sa",

"value": true,

"endpoint": {

"serviceName": "spanstore-jdbc",

"ipv4": "127.0.0.1",

"port": 3306

}

}

]

}

]

## Instrumenting a library

This is an advanced topic. Before reading further, you may want to check whether an instrumentation library for your platform [already exists](https://zipkin.io/pages/existing_instrumentations). If not and if you want to take on creating an instrumentation library, first things first; jump on [Zipkin Gitter chat channel](https://gitter.im/openzipkin/zipkin) and let us know. We’ll be extremely happy to help you along the way.

### Overview

To instrument a library, you’ll need to understand and create the following elements:

1. Core data structures - the information that is collected and sent to Zipkin
2. Trace identifiers - what tags for the information are needed so it can be reassembled in a logical order by Zipkin
   * Generating identifiers - how to generate these IDs and which IDs should be inherited
   * Communicating trace information - additional information that is sent to Zipkin along with the traces and their IDs.
3. Timestamps and duration - how to record timing information about an operation.

Alright, ready? Here we go.

### Core data structures

Core data structures are documented in detail in [Thrift](https://github.com/openzipkin/zipkin-api/blob/master/thrift/zipkinCore.thrift) comments. Here’s a high-level description to get you started:

**Annotation**

An Annotation is used to record an occurrence in time. There’s a set of core annotations used to define the beginning and end of an RPC request:

* **cs** - Client Send. The client has made the request. This sets the beginning of the span.
* **sr** - Server Receive: The server has received the request and will start processing it. The difference between this and cs will be combination of network latency and clock jitter.
* **ss** - Server Send: The server has completed processing and has sent the request back to the client. The difference between this and sr will be the amount of time it took the server to process the request.
* **cr** - Client Receive: The client has received the response from the server. This sets the end of the span. The RPC is considered complete when this annotation is recorded.

When using message brokers instead of RPCs, the following annotations help clarify the direction of the flow:

* **ms** - Message Send: The producer sends a message to a broker.
* **mr** - Message Receive: A consumer received a message from a broker.

Unlike RPC, messaging spans never share a span ID. For example, each consumer of a message is a different child span of the producing span.

Other annotations can be recorded during the request’s lifetime in order to provide further insight. For instance adding an annotation when a server begins and ends an expensive computation may provide insight into how much time is being spent pre and post processing the request versus how much time is spent running the calculation.

**BinaryAnnotation**

Binary annotations do not have a time component. They are meant to provide extra information about the RPC. For instance when calling an HTTP service, providing the URI of the call will help with later analysis of requests coming into the service. Binary annotations can also be used for exact match search in the Zipkin Api or UI.

**Endpoint** Annotations and binary annotations have an endpoint associated with them. With two exceptions, this endpoint is associated with the traced process. For example, the service name drop-down in the Zipkin UI corresponds with Annotation.endpoint.serviceName or BinaryAnnotation.endpoint.serviceName. For the sake of usability, the cardinality of Endpoint.serviceName should be bound. For example, it shouldn’t include variables or random numbers.

**Span**

A set of Annotations and BinaryAnnotations that correspond to a particular RPC. Spans contain identifying information such as traceId, spanId, parentId, and RPC name.

Spans are usually small. For example, the serialized form is often measured in KiB or less. When spans grow beyond orders of KiB, other problems occur, such as hitting limits like Kafka message size (1MiB). Even if you can raise message limits, large spans will increase the cost and decrease the usability of the tracing system. For this reason, be conscious to store data that helps explain system behavior, and don’t store data that doesn’t.

**Trace**

A set of spans that share a single root span. Traces are built by collecting all Spans that share a traceId. The spans are then arranged in a tree based on spanId and parentId thus providing an overview of the path a request takes through the system.

## Trace identifiers

In order to reassemble a set of spans into a full trace three pieces of information are required. Trace identifiers can be 128-bit, but span identifiers within a trace are always 64-bit.

**Trace Id**

The overall 64 or 128-bit ID of the trace. Every span in a trace shares this ID.

**Span Id**

The ID for a particular span. This may or may not be the same as the trace id.

**Parent Id**

This is an optional ID that will only be present on child spans. That is the span without a parent id is considered the root of the trace.

### Generating identifiers

Let’s walk through how Spans are identified.

When an incoming request has no trace information attached, we generate a random trace ID and span ID. The span ID can be reused as the lower 64-bits of the trace ID, but it can also be completely different.

If the request already has trace information attached to it, the service should use that information as server receive and server send events are part of the same span as the client send and client receive events

If the service calls out to a downstream service a new span is created as a child of the former span. It is identified by the same trace id, a new span id, and the parent id is set to the span id of the previous span. The new span id should be 64 random bits.

**Note** This process must be repeated if the service makes multiple downstream calls. That is each subsequent span will have the same trace id and parent id, but a new and different span id.

### Communicating trace information

Trace information needs to be passed between upstream and downstream services in order to reassemble a complete trace. Five pieces of information are required:

* Trace Id
* Span Id
* Parent Id
* Sampled - Lets the downstream service know if it should record trace information for the request.
* Flags - Provides the ability to create and communicate feature flags. This is how we can tell downstream services that this is a “debug” request.

Check [here](https://github.com/openzipkin/b3-propagation) for the format.

[Finagle](https://twitter.github.io/finagle/) provides mechanisms for passing this information with HTTP and Thrift requests. Other protocols will need to be augmented with the information for tracing to be effective.

**Instrumentation sampling decisions are made at the edge of the system**

Downstream services must honour the sampling decision of the upstream system. If there’s no “Sampled” information in the incoming request, the library should make a decision on whether to sample this request, and include the decision in further downstream requests. This simplifies the math when it comes to understanding what’s sampled and what isn’t. It also ensures that a request is either fully traced, or not traced at all, making the sampling policy easier to understand and configure.

Note that the debug flag will force a trace to be sampled, regardless of any sampling rules. The debug flag also applies to storage tier sampling, which is configured on the server side of Zipkin.

**HTTP Tracing**

HTTP headers are used to pass along trace information.

The B3 portion of the header is so named for the original name of Zipkin: BigBrotherBird.

Ids are encoded as [hex strings](https://github.com/twitter/finagle/blob/master/finagle-core/src/main/scala/com/twitter/finagle/tracing/Id.scala):

* X-B3-TraceId: 128 or 64 lower-hex encoded bits (required)
* X-B3-SpanId: 64 lower-hex encoded bits (required)
* X-B3-ParentSpanId: 64 lower-hex encoded bits (absent on root span)
* X-B3-Sampled: Boolean (either “1” or “0”, can be absent)
* X-B3-Flags: “1” means debug (can be absent)

For more information on B3, please see its [specification](https://github.com/openzipkin/b3-propagation).

**Thrift Tracing**

Finagle clients and servers negotate whether they can handle extra information in the header of the thrift message when a connection is established. Once negotiated trace data is packed into the front of each thrift message.

## Timestamps and duration

Span recording is when timing information or metadata is structured and reported to zipkin. One of the most important parts of this process is appropriately recording timestamps and duration.

**Timestamps are microseconds**

All Zipkin timestamps are in epoch microseconds (not milliseconds). This value should use the most precise measurement available. For example, clock\_gettime or simply multiply epoch milliseconds by 1000. Timestamps fields are stored as 64bit signed integers eventhough negative is invalid.

Microsecond precision primarily supports “local spans”, which are in-process operations. For example, with higher precision, you can tell nuances of what happened before something else.

All timestamps have faults, including clock skew between hosts and the chance of a time service resetting the clock backwards. For this reason, spans should record their duration when possible.

**Span duration is also microseconds**

While it is possible to get nanosecond-precision timing information, Zipkin uses microsecond granularity. Here are some reasons why:

First, using the same unit as timestamps makes math easier. For example, if you are troubleshooting a span, it is easier to identify with terms in the same unit.

Next, the overhead of recording a span is often variable and can be microseconds or more: suggesting a higher resolution than overhead can be distracting.

Future versions of Zipkin may revisit this topic, but for now, everything is microseconds.

**When to set Span.timestamp and duration**

Span.timestamp and duration should only be set by the host that started the span.

The simplest logic is generally this:

unless (logging "sr" in an existing span) {

set Span.timestamp and duration

}

Zipkin merges spans together that share the same trace and span ID. The most common case of this is to merge a span reported by both the client (cs, cr) and the server (sr, ss). For example, the client starts a span, logging “cs” and propagates it via B3 headers, the server continues that span by logging “sr”.

In this case, the client started the span, so it should record Span.timestamp and duration, and those values should match the difference between “cs” and “cr”. The server did not start this span, so it should not set Span.timestamp or duration.

Another common case is when a server starts a root span from an uninstrumented client, such as a web browser. It knows it should start a trace because none was present in B3 headers or similar. Since it started the trace, it should record Span.timestamp and duration on the root span.

Note: When a span is incomplete, you could set Span.timestamp, but not duration as there’s not enough information to do that accurately.

**What happens when Span.timestamp and duration are not set?**

Span.timestamp and Span.duration are fields added in 2015, 3 years after Zipkin started. Not all libraries log these. When these fields are not set, Zipkin adds them at query time (not collection time); this is not ideal.

The [duration query](https://zipkin.io/zipkin-api/#/paths/%252Ftraces/get/parameters/minDuration) will not work as there’s no data to query. Also, local (in-process) spans aren’t required to have annotations, so they cannot be queried unless their timestamp is set.

When duration isn’t set by instrumentation, Zipkin tries to derive duration at query time, it has to use the problematic method of timestamp math. Ex. if an NTP update happened inside the span, the duration Zipkin caculates will be wrong.

Finally, there’s a desire for many to move to single-host spans. The migration path towards this is to split dual-host RPC spans into two. When instrumentation logs timestamp only for spans it owns, splitting collectors have a heuristic to distinguish a server-initiated root span from a client-initiated, dual-host one.

The bottom-line is that choosing not to record Span.timestamp and duration will result in less accurate data and less functionality. Since it is very easy to record these authoritatively before reporting, all Zipkin instrumentation should do it or ask someone to help them do it.

## One-way RPC Tracing

One-way is the same as normal RPC tracing, except there is no response anticipated.

In normal RPC tracing 4 annotations are used: “cs” “sr” (request) then “ss” “cr” (response). In one-way tracing, the first two are used “cs” “sr” as there is no response returned to the caller.

So, the client adds “cs” to a span and reports it to zipkin. Then, the server adds “sr” to the same span and reports it. Neither side add Span.timestamp or duration because neither side know both when the span started and finished.

Here’s a diagram of one-way RPC tracing:

Client Tracer Server Tracer

+------------------+ +------------------+

| +--------------+ | +-----------------+ | +--------------+ |

| | TraceContext |======>| Request Headers |========>| TraceContext | |

| +--------------+ | +-----------------+ | +--------------+ |

+--------||--------+ +--------||--------+

start || ||

\/ finish ||

span(context).annotate("cs") \/

span(context).annotate("sr")

Here’s an example of this process using the [Brave Tracer](https://github.com/openzipkin/brave/blob/master/brave/src/test/java/brave/features/async/OneWaySpanTest.java):

Client side:

// Add trace identifiers to the outbound span

tracing.propagation().injector(Request::addHeader)

.inject(span.context(), request);

client.send(request);

// start the client side and flush instead of processing a response

span.kind(Span.Kind.CLIENT)

.start().flush();

// The above will report to zipkin trace identifiers, a "cs" annotation with the

// endpoint of the client

Server side:

// Parse the span from request headers

TraceContextOrSamplingFlags result =

tracing.propagation().extractor(Request::getHeader).extract(request);

// Reuse the same span ids by joining that context

span = tracer.joinSpan(result.context())

// start the server side and flush instead of processing a response

span.kind(Span.Kind.SERVER)

.start().flush();

// The above will report to zipkin trace identifiers, a "sr" annotation with the

// endpoint of the server

The above flow assumes a tracer can “flush” a span, which simply sends the span to Zipkin without attempting to calculate duration locally.

## Message Tracing

Message Tracing is different than RPC tracing because the producer and consumer don’t share span IDs.

In normal RPC tracing, client and server annotations go on the same span. This doesn’t work for messaging because there may be multiple consumers for a given message. The trace context propagated to the consumer is the parent.

Similar to one-way RPC tracing, messaging tracing doesn’t have a response path: only two annotations are used “ms” and “mr”. Unlike one-way RPC tracing, it is fine to set Span.timestamp and duration as the producer and each consumer use separate spans.

So, the producer adds “ms” to a span and reports it to zipkin. Then, each consumer creates a child span adding “mr” to it.

Here’s a diagram of Message tracing:

Producer Tracer Consumer Tracer

+------------------+ +------------------+

| +--------------+ | +-----------------+ | +--------------+ |

| | TraceContext |======>| Message Headers |========>| TraceContext | |

| +--------------+ | +-----------------+ | +--------------+ |

+--------||--------+ +--------||--------+

start || ||

\/ finish ||

span(context).annotate("ms") \/

.address("ma", broker) span(context).annotate("mr")

.address("ma", broker)

Here’s an example of this process using the [Brave Tracer](https://github.com/openzipkin/brave):

Producer side:

// Add trace identifiers to the outbound span

tracing.propagation().injector(Message::addHeader)

.inject(span.context(), message);

producer.send(message);

// start and finish the producer side

span.kind(Span.Kind.PRODUCER)

.remoteEndpoint(broker.endpoint())

.start().finish();

// The above will report to zipkin trace identifiers, a "ms" annotation with the

// endpoint of the producer, and a "ma" (Message Address) with the endpoint of

// the broker

Consumer side:

// Parse the span from message headers

TraceContextOrSamplingFlags result =

tracing.propagation().extractor(Message::getHeader).extract(message);

// Reuse the same span ids by joining that context

span = tracer.newChild(result.context())

// start and finish the consumer side indicating the message arrived.

span.kind(Span.Kind.CONSUMER)

.remoteEndpoint(broker.endpoint())

.start().finish();

// The above will report to zipkin trace identifiers, a "mr" annotation with the

// endpoint of the consumer, and a "ma" (Message Address) with the endpoint of

// the broker.

Many consumers act in bulk, receiving many messages at the same time. It may be helpful to inject each consumer span’s trace context into its corresponding message headers. This allows a processor to create a child later, at the right place in the trace tree.

Here’s an example of doing this with Kafka’s poll api:

public ConsumerRecords<K, V> poll(long timeout) {

ConsumerRecords<K, V> records = delegate.poll(timeout);

for (ConsumerRecord<K, V> record : records) {

handleConsumed(record);

}

return records;

}

void handleConsumed(ConsumerRecord record) {

// notifies zipkin the record arrived

Span span = startAndFinishConsumerSpan(record);

// allows a processor to see the parent ID (the consumer trace context)

injector.inject(span.context(), record.headers());

}