Pattern: Distributed tracing

Context

You have applied the Microservice architecture pattern. Requests often span multiple services. Each service handles a request by performing one or more operations, e.g. database queries, publishes messages, etc.

Problem

How to understand the behavior of an application and troubleshoot problems?

Forces

- External monitoring only tells you the overall response time and number of invocations - no insight into the individual operations
- Any solution should have minimal runtime overhead
- Log entries for a request are scattered across numerous logs

Solution

Instrument services with code that:

- Assigns each external request a unique external request id
- Passes the external request id to all services that are involved in handling the request (inclusive async communication)
- Includes the external request id in all log messages
- Records information (e.g. start time, end time) about the requests and operations performed when handling a external request in a centralized service

Terminology

Span

The “**span**” is the primary building block of a distributed trace, representing an individual unit of work done in a distributed system.

Each component of the distributed system contributes a span - a named, timed operation representing a piece of the workflow.

Spans can (and generally do) contain “References” to other spans, which allows multiple Spans to be assembled into one complete

Trace

Trace - a visualization of the life of a request as it moves through a distributed system.

Each span encapsulates the following state according to the OpenTracing specification:

- An operation name
- A start timestamp and finish timestamp
- A set of key:value span **Tags**
- A set of key:value span **Logs**
- A **SpanContext**

Types of Span

Example Span:

```
    t=0            operation name: db_query               t=x

     +-----------------------------------------------------+
     | · · · · · · · · · ·    Span     · · · · · · · · · · |
     +-----------------------------------------------------+

Tags:
- db.instance:"customers"
- db.statement:"SELECT * FROM mytable WHERE foo='bar'"
- peer.address:"mysql://127.0.0.1:3306/customers"

Logs:
- message:"Can't connect to mysql server on '127.0.0.1'(10061)"

SpanContext:
- trace_id:"abc123"
- span_id:"xyz789"
- Baggage Items:
  - special_id:"vsid1738"
```

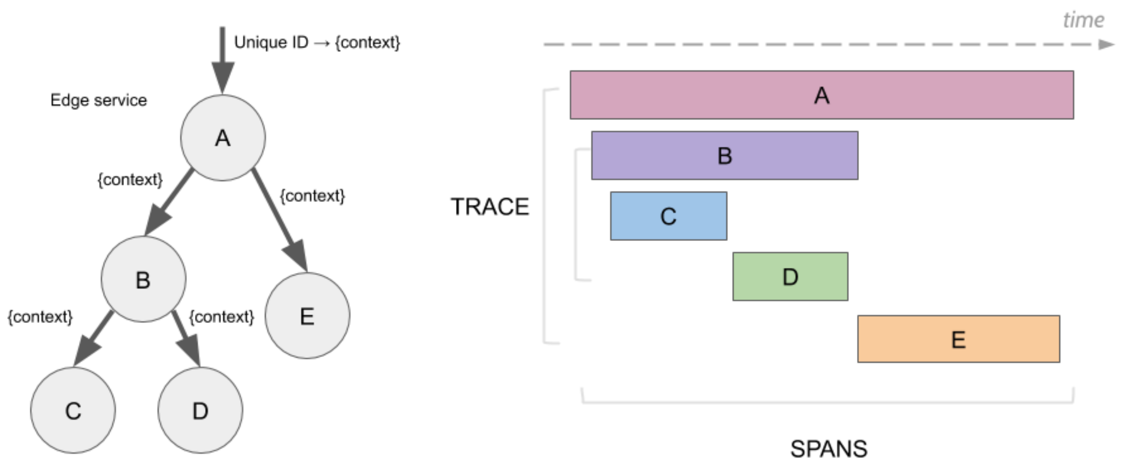
```
              [Span A]  ←←←(the root span)
                  |
           +------+------+
           |             |
       [Span B]      [Span C] ←←←(Span C is a `ChildOf` Span A)
           |             |
       [Span D]      +---+-------+
                     |           |
                 [Span E]    [Span F] >>> [Span G] >>> [Span H]
                                             ↑
                                             ↑
                                             ↑
                               (Span G `FollowsFrom` Span F)
```

`ChildOf` references: A Span may be the `ChildOf` a parent Span. In a `ChildOf` reference, the parent Span depends on the child Span in some capacity. All of the following would constitute `ChildOf` relationships:

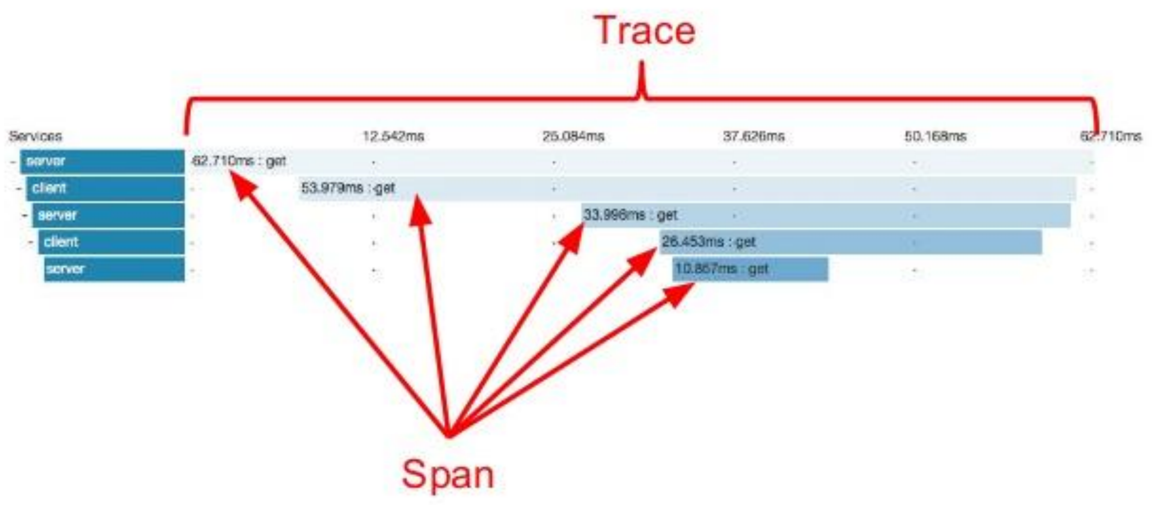
- A Span representing the server side of an RPC may be the `ChildOf` a Span representing the client side of that RPC
- A Span representing a SQL insert may be the `ChildOf` a Span representing an ORM save method
- Many Spans doing concurrent (perhaps distributed) work may all individually be the `ChildOf` a single parent Span that merges the results for all children that return within a deadline

`FollowsFrom` references: Some parent Spans do not depend in any way on the result of their child Spans. In these cases, we say merely that the child Span `FollowsFrom` the parent Span in a causal sense. There are many distinct `FollowsFrom` reference sub-categories, and in future versions of OpenTracing they may be distinguished more formally.

Examples

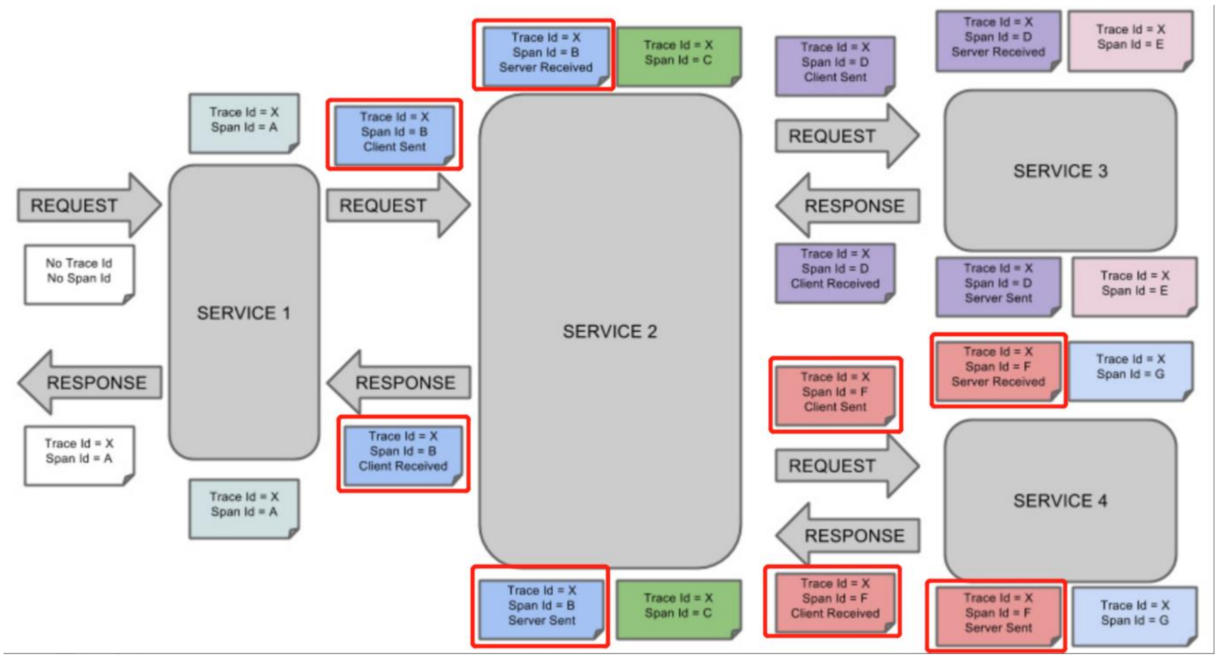


Trace and Span



Attributes:

spanId	the id of a specific operation that took place
appname	the id of the latency graph that contains the span
traceId	the id of the latency graph that contains the span
exportable	whether the log should be exported to Zipkin or not. When would you like the span not to be exportable? In the case in which you want to wrap some operation in a Span and have it written to the logs only.



Sleuth

Overview

Spring Cloud Sleuth provides Spring Boot auto-configuration for distributed tracing. Underneath, Spring Cloud Sleuth is a layer over a Tracer library named Brave.

Sleuth configures everything you need to get started. This includes where trace data (spans) are reported to, how many traces to keep (sampling), if remote fields (baggage) are sent, and which libraries are traced.

Features

Sleuth sets up instrumentation not only to track timing, but also to catch errors so that they can be analyzed or correlated with logs. This works the same way regardless of if the error came from a common instrumented library, such as `RestTemplate`, or your own code annotated with `@NewSpan` or similar.

Below, we'll use the word Zipkin to describe the tracing system, and include Zipkin screenshots. However, most services accepting Zipkin format have similar base features. Sleuth can also be configured to send data in other formats, something detailed later.

Log correlation

Sleuth configures the logging context with variables including the service name (`%{spring.zipkin.service.name}`) and the trace ID (`%{traceId}`). These help you connect logs with distributed traces and allow you choice in what tools you use to troubleshoot your services.

Once you find any log with an error, you can look for the trace ID in the message. Paste that into Zipkin to visualize the entire trace, regardless of how many services the first request ended up hitting.

```
backend.log:  2020-04-09 17:45:40.516 ERROR
[backend,5e8eeec48b08e26882aba313eb08f0a4,dcc1df555b5777b3,true] 97203 --- [nio-
9000-exec-1] o.s.c.s.i.web.ExceptionLoggingFilter     : Uncaught exception thrown
frontend.log:2020-04-09 17:45:40.574 ERROR
[frontend,5e8eeec48b08e26882aba313eb08f0a4,82aba313eb08f0a4,true] 97192 --- [nio-
8081-exec-2] o.s.c.s.i.web.ExceptionLoggingFilter     : Uncaught exception thrown
```

Above, you'll notice the trace ID is `5e8eeec48b08e26882aba313eb08f0a4`, for example. This log configuration was automatically setup by Sleuth.

Adding sleuth to your project

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

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>
      <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>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.

Add the dependency to `spring-cloud-starter-zipkin`.

Getting Run Example

Example contains two services: basic and composite, additionally basic service communicates asynchronously using Kafka topic.

Setup example

Prerequisites:

1. Checkout tracing-base-service and tracing-compose services from GutHub
2. Install [Kafka server]()
3. Install [Open Zipkin]()

Start:

1. Start Zookeeper server from Kafka installation: `bin/zookeeper-server-start.sh config/zookeeper.properties`
2. Start Kafka server: `bin/kafka-server-start.sh config/server.properties`
3. Start Open Zipkin: `start java -jar zipkin-server-2.12.9-exec.jar`
4. Start trace-basic-service as SpringBoot application
5. Start trace-composit-service as SpringBoot application
6. Access [http://localhost:9411]()

Synchronous Communication

Send get request to composite service: curl localhost:9081/composite/hello

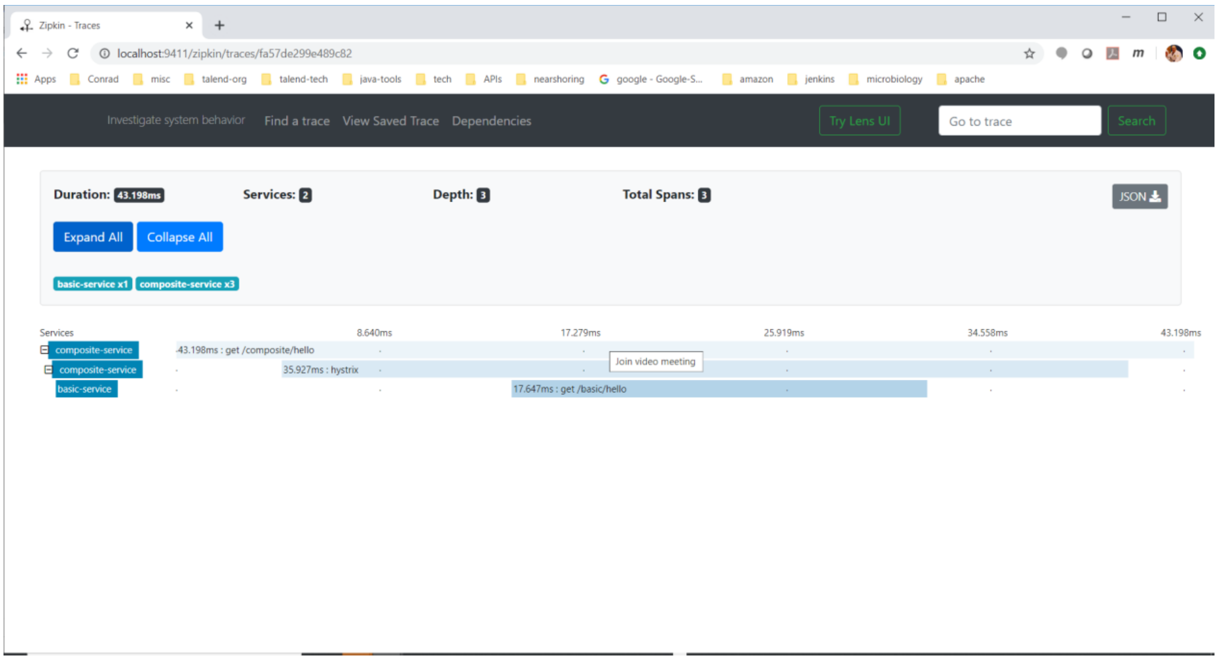
Examine log files in trace-composite-service:

```
15:02:37 INFO  [hystrix-default-5] [traceId=8bd520ab5c323898, spanId=94375acad9900bcf, spanExportable=true, X-Span-Export=true, X-B3-SpanId=94375acad9900bcf, X-B3-ParentSpanId=8bd520ab5c323898, X-B3-TraceId=8bd520ab5c323898, parentId=8bd520ab5c323898] c.t.m.patterns.trace.BasicServiceClient  Invoking basic service ...

15:02:37 INFO  [hystrix-default-5] [traceId=8bd520ab5c323898, spanId=94375acad9900bcf, spanExportable=true, X-Span-Export=true, X-B3-SpanId=94375acad9900bcf, X-B3-ParentSpanId=8bd520ab5c323898, X-B3-TraceId=8bd520ab5c323898, parentId=8bd520ab5c323898] c.t.m.patterns.trace.BasicServiceClient  Returned from basic service ...
```

```
15:02:37 TRACE [http-nio-9080-exec-2] [traceId=8bd520ab5c323898, spanId=dac952b53beca790, spanExportable=true, X-Span-Export=true, X-B3-SpanId=dac952b53beca790, X-B3-ParentSpanId=94375acad9900bcf, X-B3-TraceId=8bd520ab5c323898, parentId=94375acad9900bcf] org.zalando.logbook.Logbook {"origin":"local","type":"response","correlation":"d9b921b3af1a9bb0","duration":4,"protocol":"HTTP/1.1","status":200,"headers":{"Connection":["keep-alive"],"Content-Length":["24"],"Content-Type":["text/plain;charset=UTF-8"],"Date":["Thu, 11 Jun 2020 13:02:37 GMT"],"Keep-Alive":["timeout=60"]},"body":"Hello from basic service"}
```

Examine Zipkin server:



Asynchronous Communication

Send get request to composite service: curl localhost:9081/composite/send

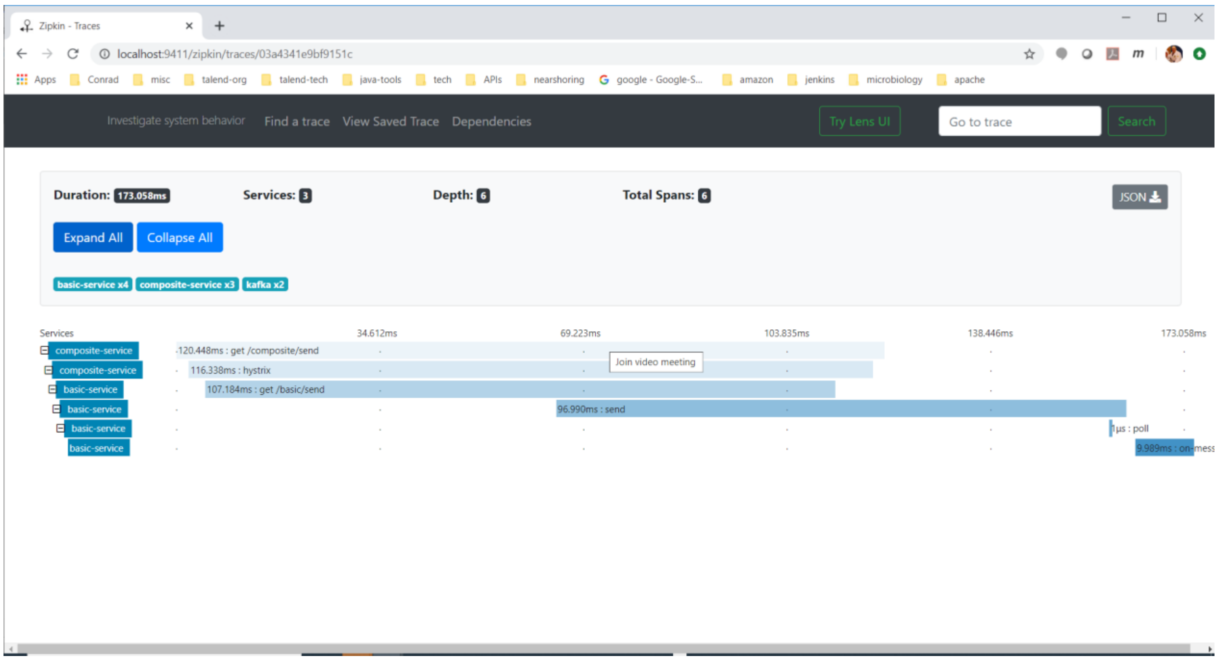
Examine log files:

```
15:06:21 INFO  [hystrix-default-6] [traceId=03a4341e9bf9151c, spanId=203949790dc9bfd0, spanExportable=true, X-Span-Export=true, X-B3-SpanId=203949790dc9bfd0, X-B3-ParentSpanId=03a4341e9bf9151c, X-B3-TraceId=03a4341e9bf9151c, parentId=03a4341e9bf9151c] c.t.m.patterns.trace.BasicServiceClient  Invoking basic service with send operation ...

15:06:21 INFO  [hystrix-default-6] [traceId=03a4341e9bf9151c, spanId=203949790dc9bfd0, spanExportable=true, X-Span-Export=true, X-B3-SpanId=203949790dc9bfd0, X-B3-ParentSpanId=03a4341e9bf9151c, X-B3-TraceId=03a4341e9bf9151c, parentId=03a4341e9bf9151c] c.t.m.patterns.trace.BasicServiceClient  Returned from basic service send operation ...
```

```
15:06:21 INFO  [org.springframework.kafka.KafkaListenerEndpointContainer#0-0-C-1] [traceId=03a4341e9bf9151c, spanId=21424d87b9692351, spanExportable=true, X-Span-Export=true, X-B3-SpanId=21424d87b9692351, X-B3-ParentSpanId=70b0c2047fe9f118, X-B3-TraceId=03a4341e9bf9151c, parentId=70b0c2047fe9f118] c.t.m.patterns.trace.kafka.Receiver received payload='basic test'; headers='{kafka_offset=0, kafka_consumer=brave.kafka.clients.TracingConsumer@94b9b47, kafka_timestampType=CREATE_TIME, kafka_receivedMessageKey=null, kafka_receivedPartitionId=0, kafka_receivedTopic=test, kafka_receivedTimestamp=1591880781907, kafka_groupId=helloworld}'
```

Examine Zipkin server:



Duration: 173.058ms Services: 3 Depth: 6 Total Spans: 6

basic-service x4 composite-service x3 kafka x2

```
Services                34.612ms      69.223ms      103.835ms      138.446ms      173.058ms
composite-service     120.448ms : get /composite/send
 composite-service     116.338ms : hystrix
  basic-service         107.184ms : get /basic/send
   basic-service                     96.990ms : send
    basic-service                                                          1µs : poll
     basic-service                                                          9.989ms : on-mess
```