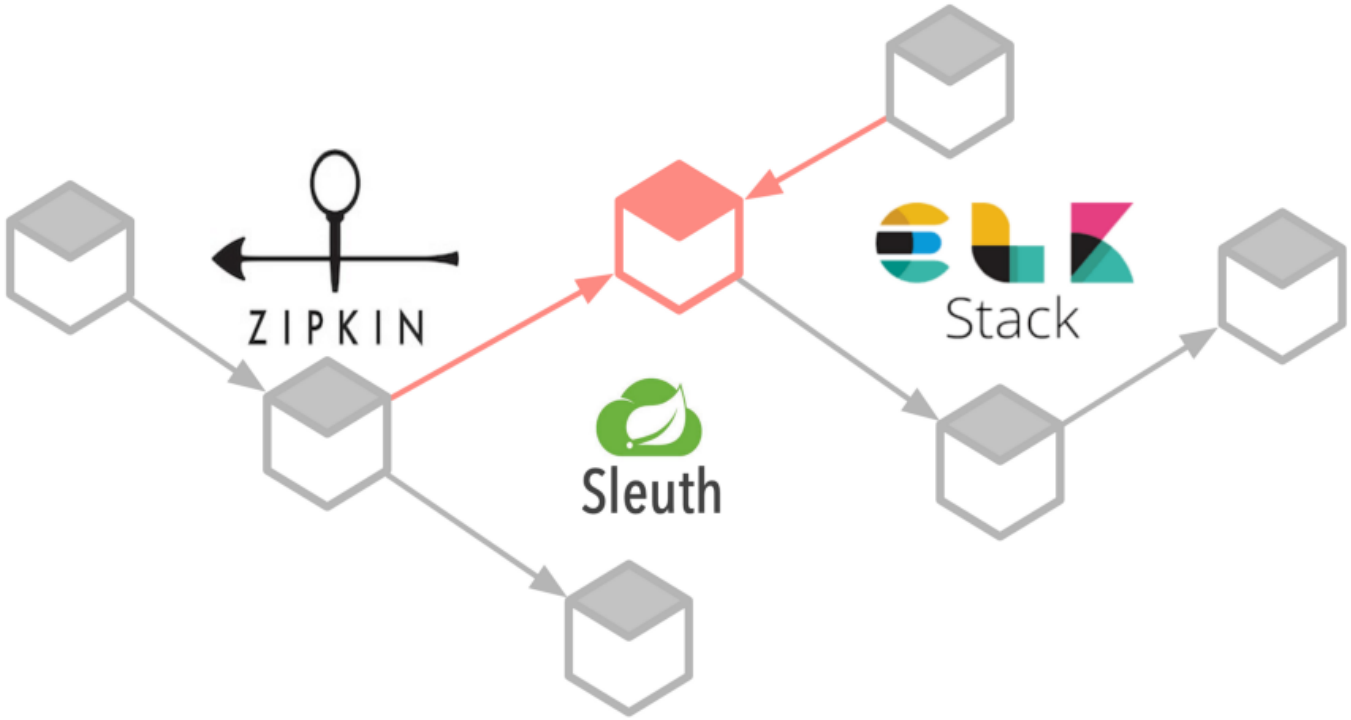
Sohan Ganapathy Follow

Jun 6, 2019 · 6 min read ★ · Listen

Distributed Tracing in Micoservices using Zipkin, Sleuth and ELK Stack.



What is Distributed Tracing ?

One of the major challenges in microservices is the ability to debug issues and monitor

such as databases or log files. In addition to that, we might also want to track down why a certain microservice call is taking so much time in a given business flow.

The Distributed Tracing pattern addresses the above challenges developers face while building microservices. There are some helpful open-source tools that can be used for distributed tracing, when creating microservices with Spring Boot and Spring Cloud frameworks. This blog walks through the installation steps and implementations of these tools.

The Tools

Spring Cloud Sleuth: A Spring Cloud library that lets you track the progress of subsequent microservices by adding trace and span id's on the appropriate HTTP request headers. The library is based on the MDC (Mapped Diagnostic Context) concept, where you can easily extract values put to context and display them in the logs.

Zipkin: A Java-based distributed tracing application that helps gather timing data for every request propagated between independent services. It has a simple management console where we can find a visualization of the time statistics generated by subsequent services.

ELK Stack: Three open source tools — **Elasticsearch**, **Logstash** and **Kibana** form the ELK stack. They are used for searching, analyzing, and visualizing log data in real-time. Elasticsearch is a search and analytics engine. Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like Elasticsearch. Kibana lets us visualize this data with charts and graphs.

How do they work together ?

Based on the below diagram (Image A), when the Orchestrator Service makes a HTTP call on the service `*/order/{orderId}*`, the call is intercepted by Sleuth and it adds the necessary tags to the request headers. After the Orchestrator Service receives the HTTP response, the data is sent asynchronously to Zipkin to prevent delays or failures relating to

request. The trace ID contains a set of span IDs, forming a tree-like structure. The trace ID will remain the same as one microservice calls the next.

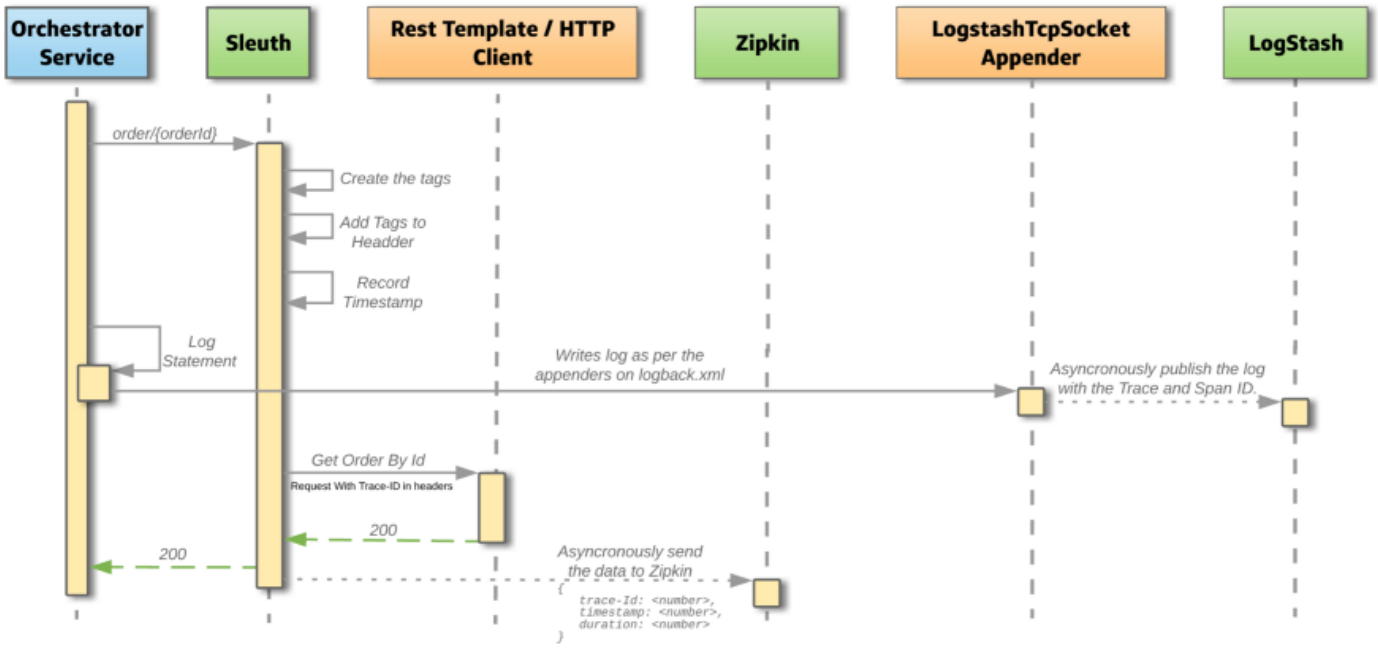


Image A — How Zipkin, Sleuth and ELK fit in.

The logs are published directly to Logstash in this example for convenience, but we can also use **Beats.** Beats is a simple data shipper that either sits on servers or on containers, that listen to log file locations and ship them to either Logstash for transformation or Elasticsearch.

Installation of the needed tools

The guide assumes the user has docker pre-installed. If not you can follow the steps for installation here.

1) Installing Zipkin

Run the first docker command to pull the Zipkin image from *hub.docker.com* and then the next docker command to start it on port 9411.

```
$ docker pull openzipkin/zipkin
```

Validate the setup by accessing the Zipkin web interface on the url: *http://localhost:9411/zipkin/*. The below screen (Image 1) should open up if there are no issues.

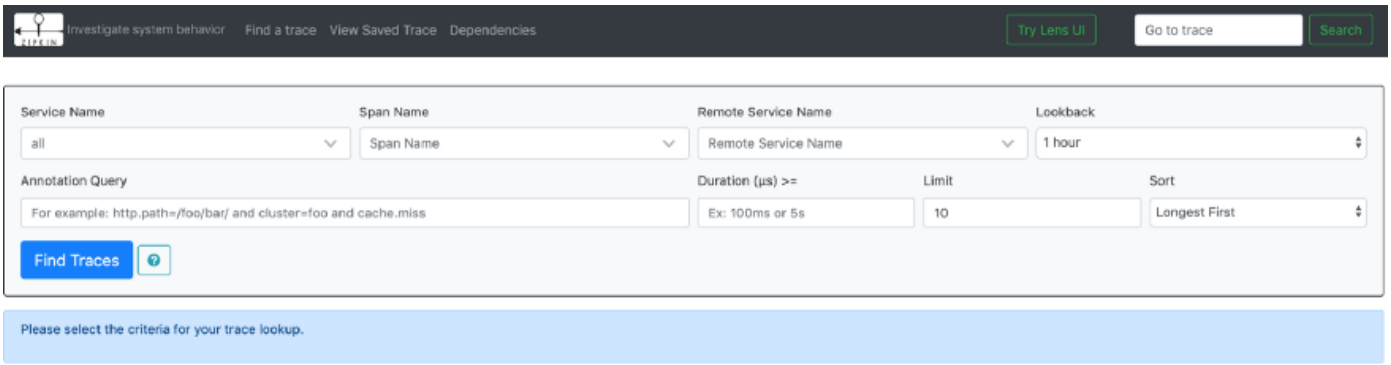


Image 1— Zipkin Dashboard

2) Installing ELK Stack

This install will be using the image `*sebp/elk*`, on this image we will be making changes to disable SSL and setup indexes for Elastic search on the Log-stash configuration files.

Create the 2 files with the configuration below:

```
input {
  tcp {
    port => 5044
    ssl => false
  }
}
```

02-beats-input.conf hosted with ❤ by GitHub view raw

The input configuration for disabling SSL

```
filter {
  json {
    source => "message"
  }
}
```

```
    }
}
```

30-output.conf hosted with ❤ by GitHub view raw

The output configuration for setting up ElasticSearch

Then create a `*DockerFile*` as below, using the configurations created above

```
FROM sebp/elk

# overwrite existing file
RUN rm /etc/logstash/conf.d/30-output.conf
COPY 30-output.conf /etc/logstash/conf.d/30-output.conf

RUN rm /etc/logstash/conf.d/02-beats-input.conf
COPY 02-beats-input.conf /etc/logstash/conf.d/02-beats-input.conf
```

DockerFile hosted with ❤ by GitHub view raw

Execute the below docker commands to build the image with tag `*local-elk*` and start all three components.

```
$ docker build . --tag local-elk

$ docker run -p 5601:5601 -p 9200:9200 -p 5044:5044 -it --name elk
local-elk
```

The `*docker run*`, command starts up Kibana on port 5601, ElasticSearch on port 9200 and LogStash on port 5044.

Validate the kibana setup by accessing the web console on url '*http://localhost:5601*'. The below screen (Image 2) should show up on the browser.

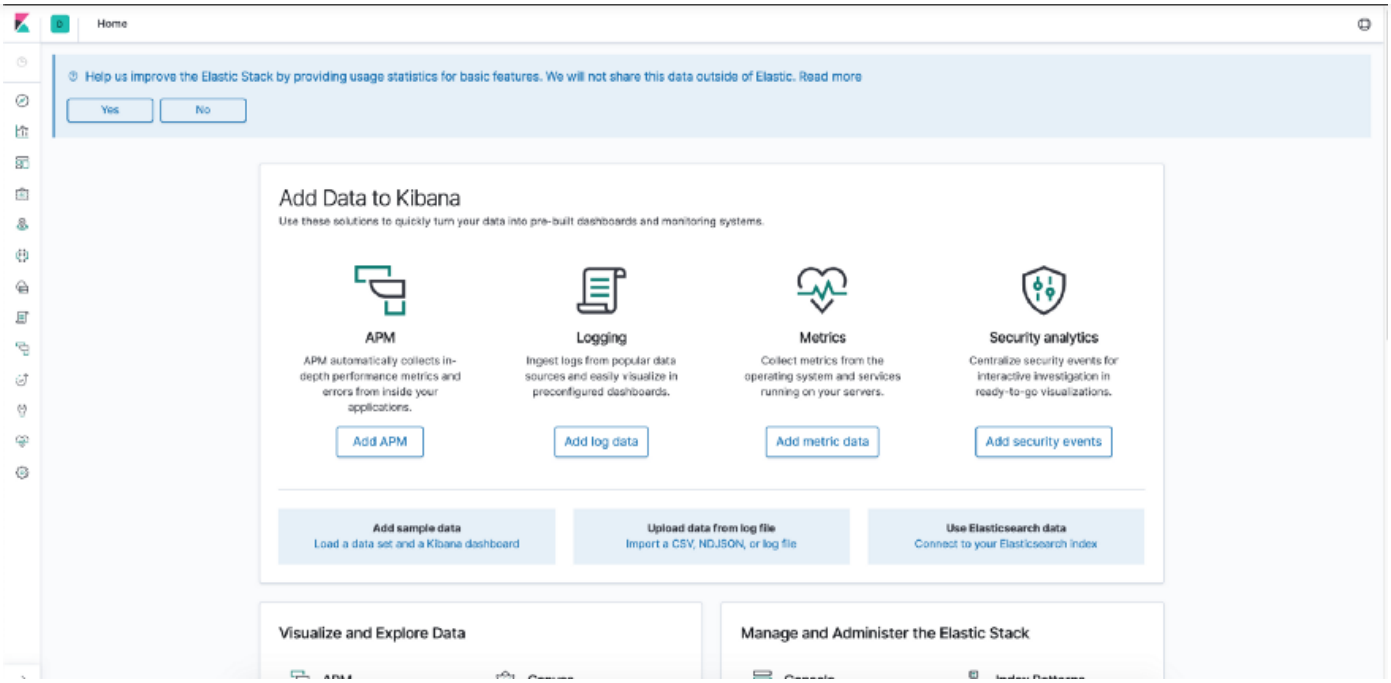


Image 2 — Kibana Dashboard

Validate Elasticsearch with the below curl command

```
curl http://localhost:9200/_cat/indices
```

This completes our installations !

Example Microservices

As depicted in Image 3, we have three microservices. The Order service (running on port 8081) has operations to fetch an order based on a given Order ID. The Customer service (running on port 8082) has operations to fetch a customer based on a given Customer ID. The Orchestrator (running on port 8080) exposes an operation to get both the order and customer details for a given Order ID. The Orchestrator first calls the order service to get the order details then makes another call to the customer service to get the customer details and returns both the details.

You can find all the code **here**

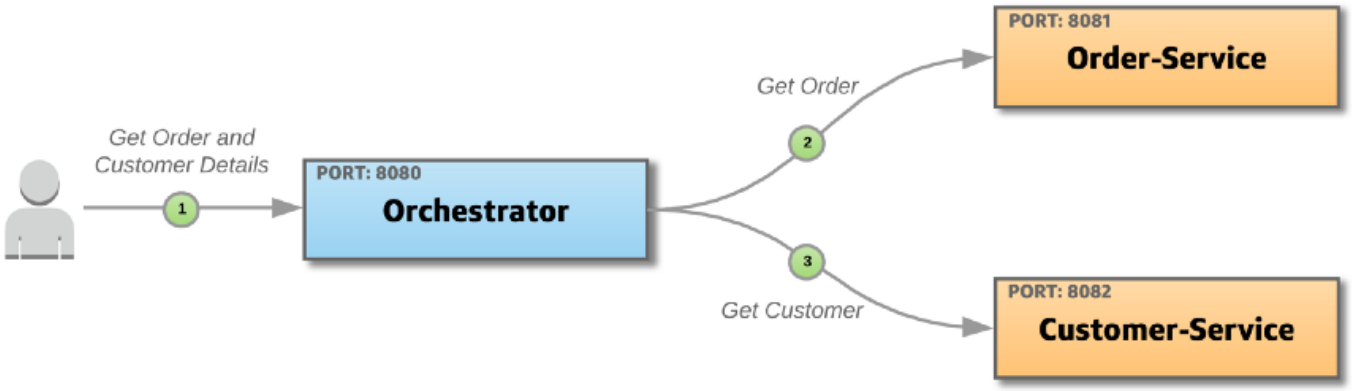


Image 3 — Simple microservice orchestration

To enable Sleuth, Zipkin and ELK stack, we need to make the below changes on all 3 microservices.

First change is the *pom.xml*, where we add the cloud-starter dependencies for both Sleuth and Zipkin, also the logback dependencies needed for logstash.

```
    <properties>
        <spring-cloud.version>Greenwich.RELEASE</spring-cloud.version>
    </properties>

...
    <!-- Dependencies for Zipkin and Sleuth -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zipkin</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-sleuth</artifactId>
    </dependency>

   <!-- Dependencies for LogStash -->
   <dependency>
        <groupId>net.logstash.logback</groupId>
        <artifactId>logstash-logback-encoder</artifactId>
        <version>5.3</version>
```

```
        </dependency>
...

        <dependencyManagement>
            <dependencies>
                <dependency>
                    <groupId>org.springframework.cloud</groupId>
                    <artifactId>spring-cloud-dependencies</artifactId>
                    <version>${spring-cloud.version}</version>
                    <type>pom</type>
                    <scope>import</scope>
                </dependency>
            </dependencies>
        </dependencyManagement>
```

pom.xml hosted with ❤ by GitHub view raw

Dependencies for Zipkin, Sleuth and Logback.

The second change is to add the URL, in the *application.properties* for spring to publish data to Zipkin.

```
# Zipkin info
spring.zipkin.base-url=http://localhost:9411/
spring.sleuth.sampler.probability=1
```

application.properties hosted with ❤ by GitHub view raw

Zipkin URL in the spring application.properties

The final change is the *logback.xml*, to publish the logs to LogStash. The appender publishes all the logs to Logstash running on port *5044*, using an Async TCP Appender. Again as mentioned above **Beats** can be used to ship logs to Logstash too.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">
    <include resource="org/springframework/boot/logging/logback/base.xml"/>
    <appender name="logstash" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
```

```
                <version/>
                <logLevel/>
                <loggerName/>
                <message/>
                <pattern>
                    <pattern>
                        {
                            "appName": "order-service"
                        }
                    </pattern>
                </pattern>
                <threadName/>
                <stackTrace/>
            </providers>
        </encoder>
    </appender>
    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="logstash"/>
    </root>
    <logger name="org.springframework" level="INFO"/>
    <logger name="com.sohan" level="INFO"/>
</configuration>
```

logback.xml hosted with ❤ by GitHub view raw

Logback xml to publish logs to Logstash

All the services that need to use the Distributed Tracing feature, will need the above three changes / additions.

Seeing the magic happen

Once all three services are up and running, we can test the setup by executing the below curl which returns the Order and Customer details.

```
curl -X GET http://localhost:8080/customer-orders/100
```

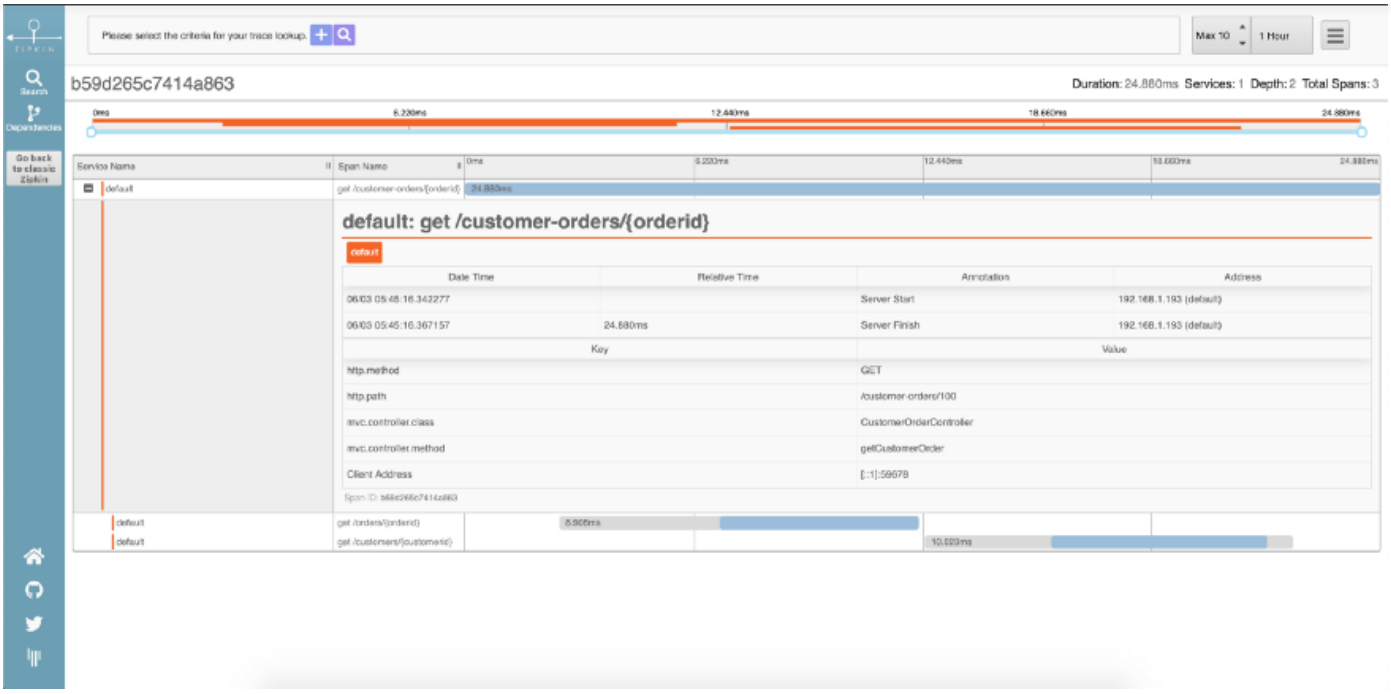


Image 4 — Zipkin trace for the Order-Customer-Orchestrator

By stopping the customer-service, we can see the failure (Image 5) being flagged on Zipkin dashboard.

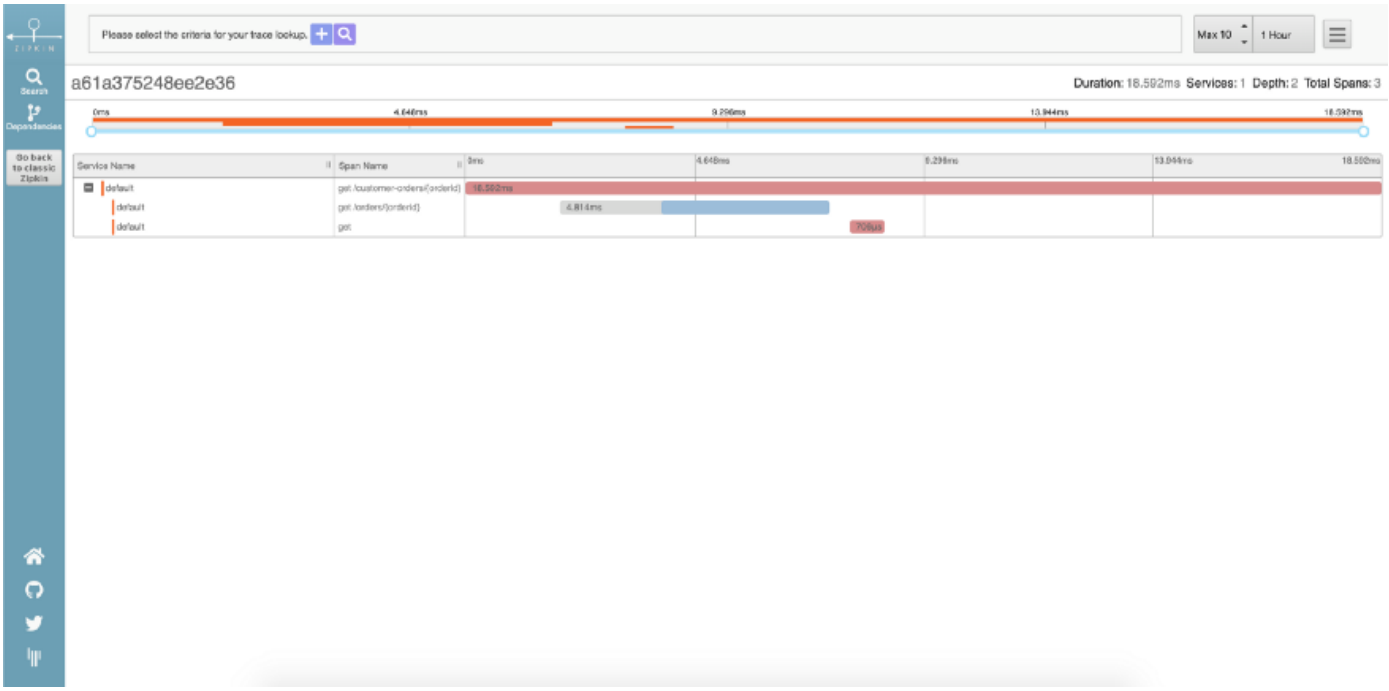


Image 5 — A failure trace

Before we see the logs we need to configure the Kibana dashboard to use the index we created on log stash.

1. In the output configuration we, created the index with the name ‘logstash-local’. On the Kibana dashboard, click on `Create Index Pattern`, enter that as the index pattern and click on “Next Step”.

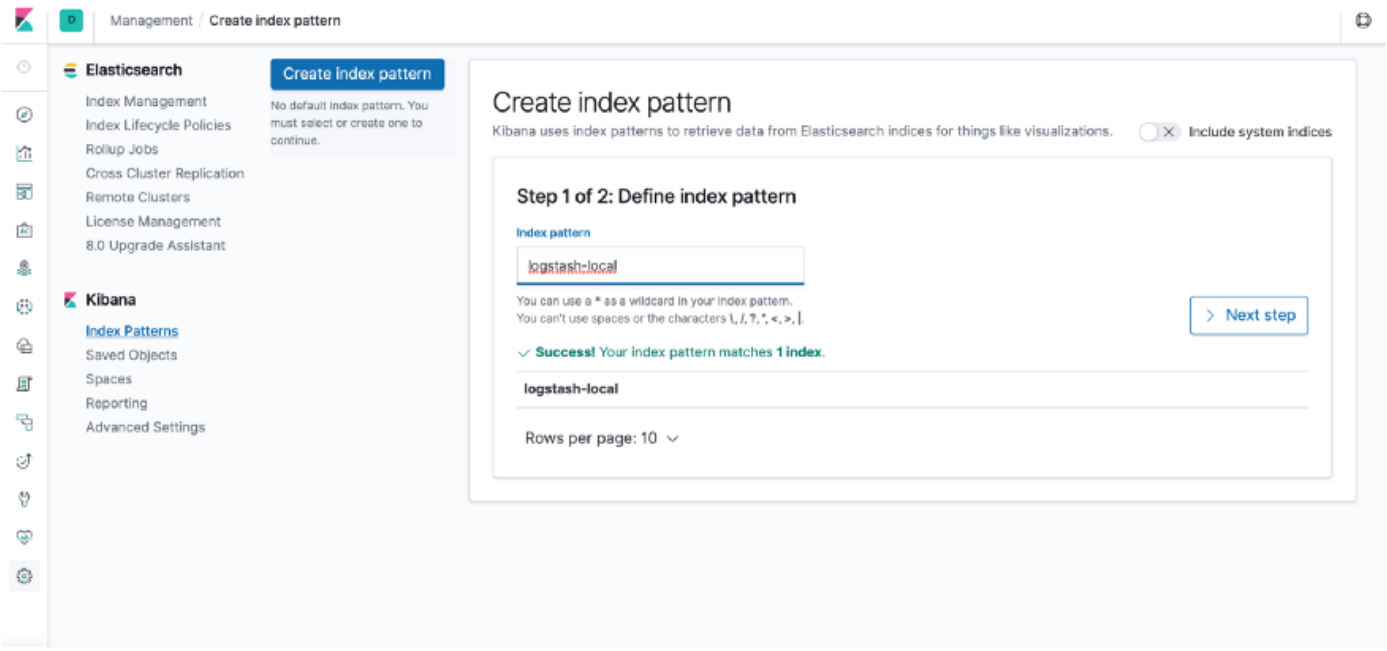


Image 6 — Zipkin trace for the Order-Customer-Orchestrator

The next step would be to select `timestamp`, on the configure setting screen. Once the pattern has been created, you should see a screen as Image 7.

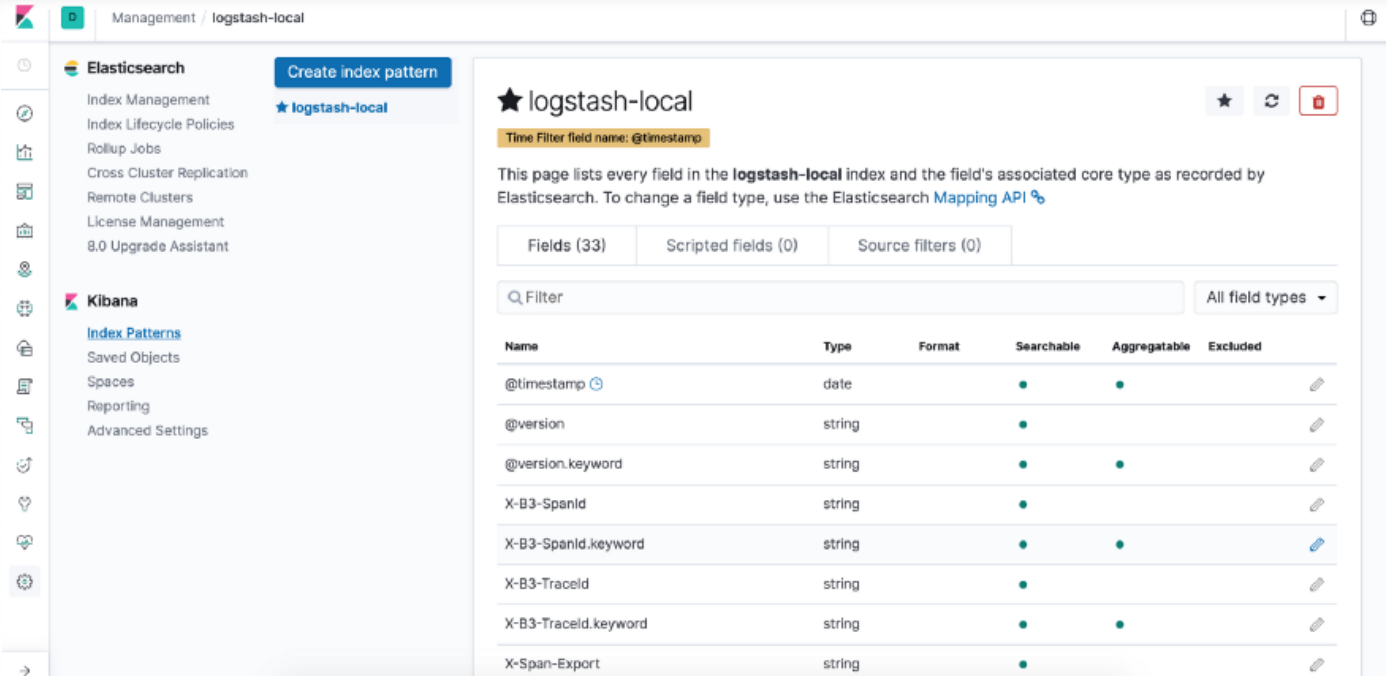
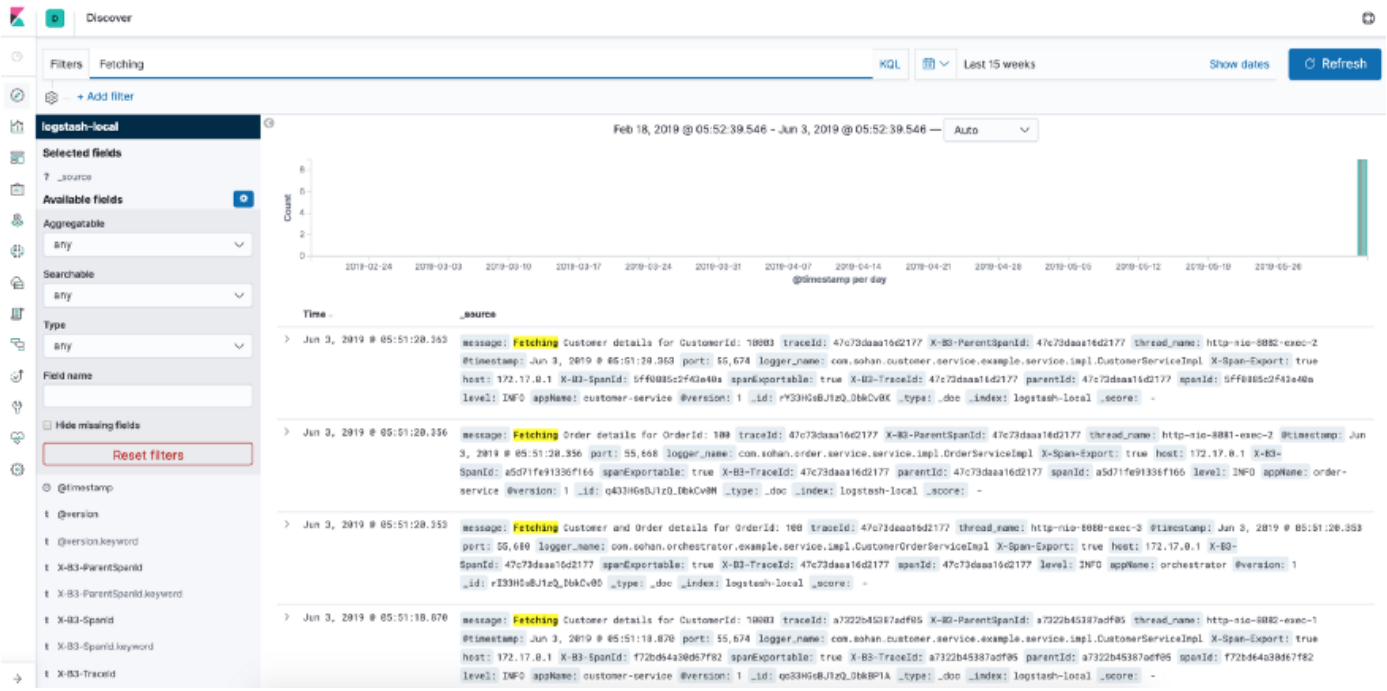


Image 7 — Zipkin trace for the Order-Customer-Orchestrator

Now by clicking the discover link (the button as a compass on the left menu), we should be able to see the logs generated from the services. We can also filter them by the TraceId's, from the Zipkin dashboard.



With tools like Sleuth, Zipkin and ELK Stack, Distributed Tracing doesn't seem to be a very difficult problem to solve. As the application grows, these tools can provide us with much-needed information on where requests are spending their time and help tracing the flow. There are also some other tools that provide the same solution like Opentracing and Jaeger.