Monitoring Microservices With Spring Cloud Sleuth, ELK, and Zipkin

by Piotr Mińkowski MVB CORE · Apr. 07, 17 · Performance Zone · Tutorial

One of the most frequently mentioned challenges related to the creation of microservices-based architecture is monitoring. Each microservice should be run in an environment isolated from the other microservices so it does not share resources such as databases or log files with them.

However, the essential requirement for microservices architecture is that it is relatively easy to access the call history, including the ability to look through the request propagation between multiple microservices. Grepping the logs is not the right solution for that problem. There are some helpful tools that can be used when creating microservices with Spring Boot and Spring Cloud frameworks.

Tools

- **Spring Cloud Sleuth**: A library available as a part of Spring Cloud project. Lets you track the progress of subsequent microservices by adding the appropriate headers to the HTTP requests. 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 distributed tracing system that helps gather timing data for every request propagated between independent services. It has simple management console where we can find a visualization of the time statistics generated by subsequent services.

- **ELK**. Elasticsearch, Logstash, and Kibana — three different tools usually used together. They are used for searching, analyzing, and visualizing log data in real-time.

Many of you, even if you have not had used Java or microservices before, have probably heard about Logstash and Kibana. For example, if you look at *hub.docker.com*, among the most popular images, you will find the ones for the above tools. In our example, we will just use

popular images, you will find the ones for the above tools. In our example, we will just use those images. Let's begin with running the container with Elasticsearch.

```
docker run -d -it --name es -p 9200:9200 -p 9300:9300 elasticsearch
```

Then, we run the Kibana container and link it to Elasticsearch.

```
docker run -d -it --name kibana --link es:elasticsearch -p 5601:5601 kibana
```

At the end, we start Logstash with input and output declared. As an input, we declare TCP, which is compatible with `LogstashTcpSocketAppender` and used as a logging appender in our sample application. As an output, Elasticsearch has been declared. Each microservice will be indexed on its name with a *micro* prefix. There are many other input and output plugins available for Logstash which could be used, which are listed here. One of another input configuration method using RabbitMQ and Spring `AMQPAppender` is described in one of my previous post How to ship logs with Logstash, Elasticsearch, and RabbitMQ.

```
docker run -d -it --name logstash -p 5000:5000 logstash -e 'input { tcp { port => 5000 codec
```

Microservices

Now, let's take a look on sample microservices. This post is a continuation of one previous posts on my blog, about creating microservice using Spring Cloud, Eureka, and Zuul. Architecture and exposed services are the same as in the previous sample. The source code is available on GitHub (branch `logstash`). Like mentioned before, we will use the Logback library for sending log data to Logstash. In addition to the three Logback dependencies, we also add libraries for Zipkin integration and Spring Cloud Sleuth starter. Here's a fragment of the `pom.xml` for microservices:

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
<dependency>
        <groupId>net.logstash.logback</groupId>
        <artifactId>logstash-logback-encoder</artifactId>
        <version>4.9</version>
</dependency>
<dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
</dependency>
```

```
</dependency>
<dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-core</artifactId>
        <version>1.2.3</version>
</dependency>
```

There is also a Logback configuration file in the `src/main/resources` directory. Here's a `logback.xml` fragment. We can configure which logging fields are sending to Logstash by declaring tags like `mdc`, `logLevel`, `message`, etc. We are also appending service name field for Elasticsearch index creation.

```
<appender name="STASH" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
        <destination>192.168.99.100:5000</destination>
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
                <providers>
                        <mdc />
                        <context />
                        <logLevel />
                        <loggerName />
                        <pattern>
                                <pattern>
          {
          "serviceName": "account-service"
          }
        </pattern>
                        </pattern>
                        <threadName />
                        <message />
                        <logstashMarkers />
                        <stackTrace />
                </providers>
        </encoder>
</appender>
```

The configuration of Spring Cloud Sleuth is very simple. We only have to add the `spring-cloud-starter-sleuth` dependency to `pom.xml` and declare sampler `@Bean`. In the sample, I declared `AlwaysSampler`, which exports every span — but there is also another option, `PercentageBasedSampler`, which samples a fixed fraction of spans.

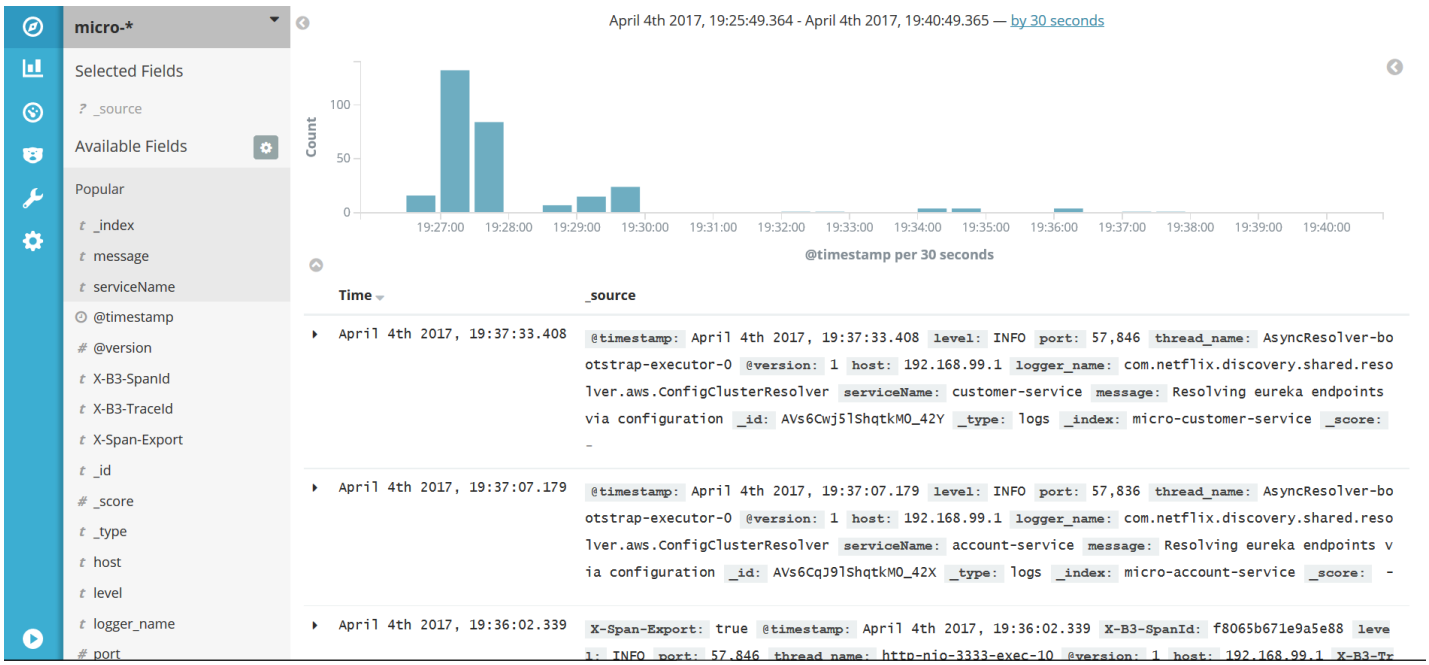
```
@Bean
public AlwaysSampler defaultSampler() {
  return new AlwaysSampler();
}
```

Kibana

After starting ELK docker containers we need to run our microservices. There are five Spring Boot applications that need to be run:

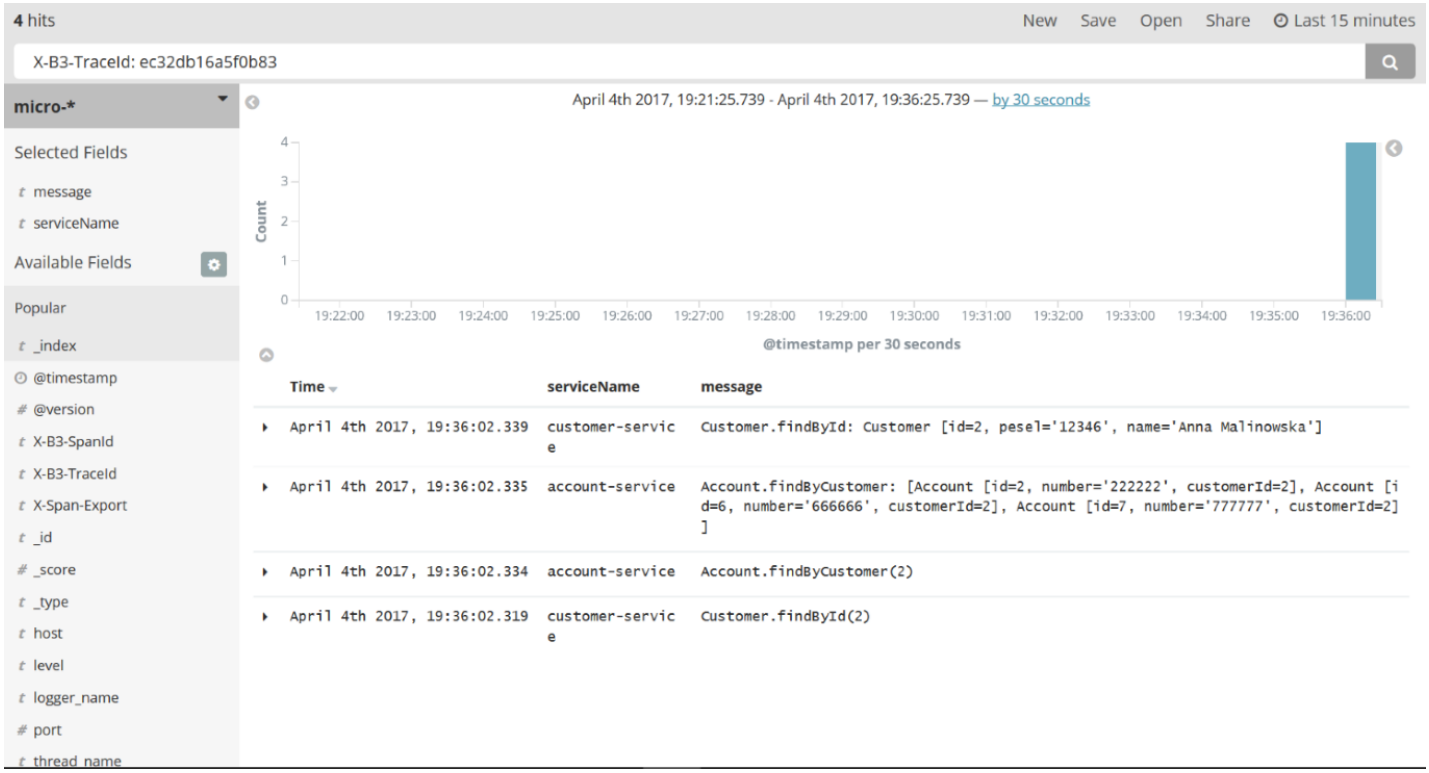
1. `discovery-service`.
2. `account-service`.
3. `customer-service`.
4. `gateway-service`.
5. `zipkin-service`.

After launching all of them, we can try call some services — for example, *http://localhost:8765/api/customer/customers/{id}*, which causes the calling of both customer and account services. All logs will be stored in Elasticsearch with the `micro-%{serviceName}` index. They can be searched in Kibana with the `micro-*` index pattern. Index patterns are created in Kibana under section **Management** > **Index patterns**. Kibana is available under address *http://192.168.99.100:5601*. After first running it, we will be prompted for an index pattern, so let's type `micro-*`. Under the **Discover** section, we can take a look at all logs matching the typed pattern with a timeline visualization.



Kibana is a rather intuitive and user-friendly tool. I will not describe in the details how to use Kibana because you can easily find it out by yourself reading a documentation or just clicking UI. The most important thing is to be able to search logs by filtering criteria. In the picture below, there is an example of searching logs by the `X-B3-TraceId` field, which is added to the request header by Spring Cloud Sleuth. Sleuth also adds `X-B3-TraceId` for marking requests for a single microservice. We can select which fields are displayed in the result list; in this sample, I selected `message` and `serviceName`, as you can see in the left pane of the

picture below.



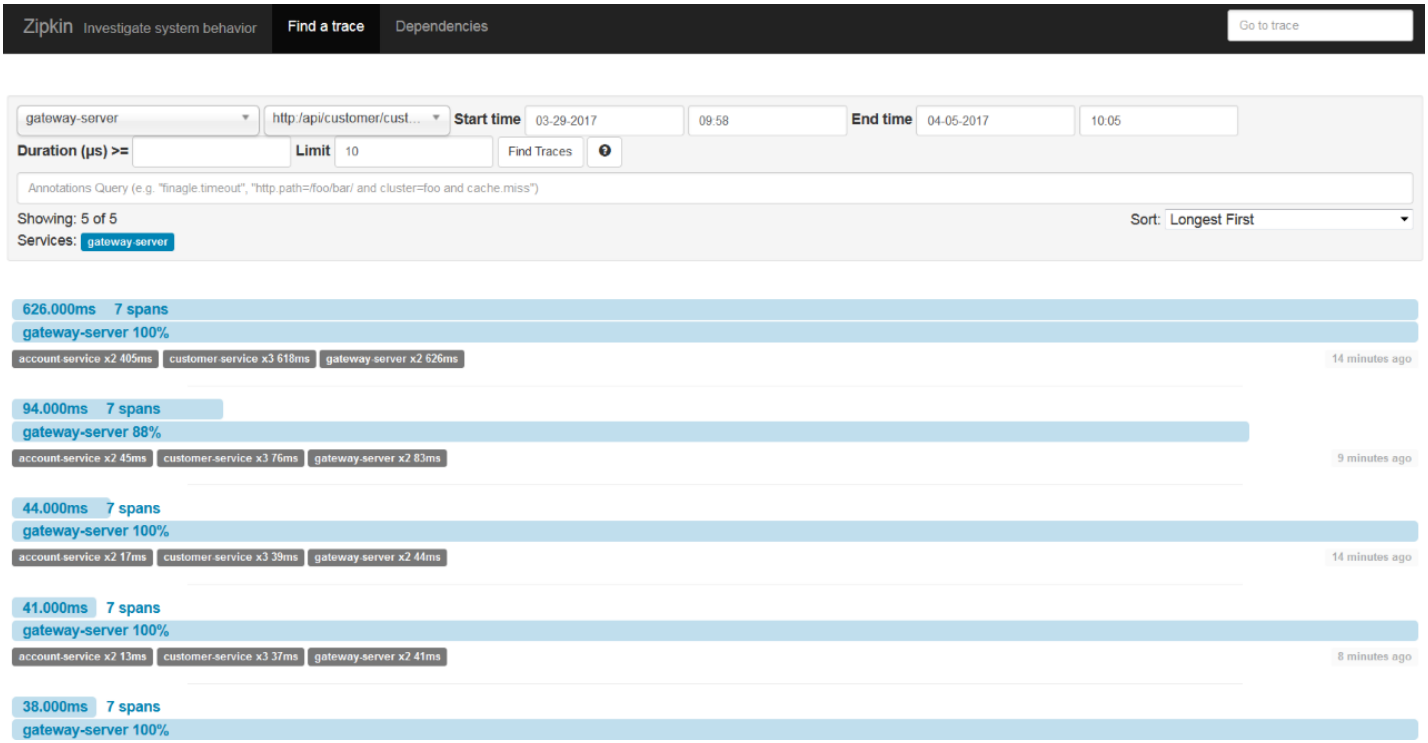
Here's a picture with single request details. It is visible after expanding each log row.

Zipkin

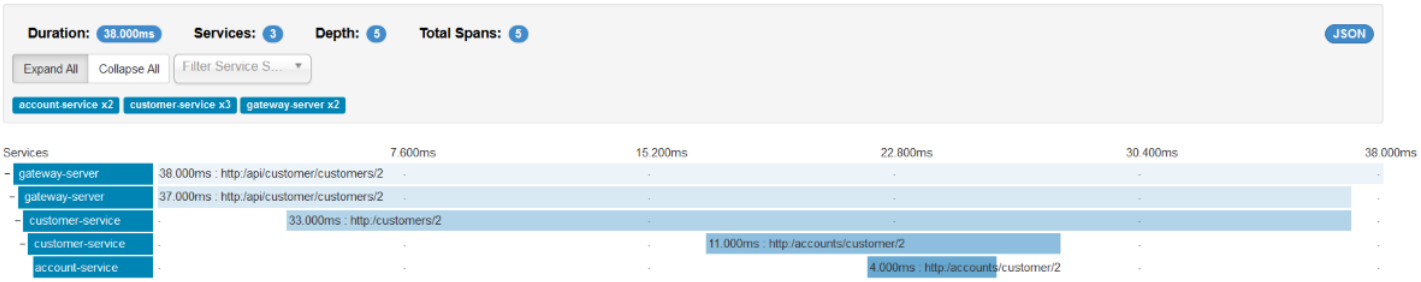
Spring Cloud Sleuth also sends statistics to Zipkin. That is another kind of data than the data that is stored in Logstash. These are timing statistics for each request. Zipkin UI is really simple. You can filter the requests by some criteria like time, service name, and endpoint name.

Below is a picture with the same requests that were visualized with Kibana

(*http://localhost:8765/api/customer/customers/{id}*).



We can always see the details of each request by clicking on it. Then, you see the picture similar to what is visible below. In the beginning, the request has been processed on the API gateway. Then, the gateway discovers customer service on Eureka server and calls that service. Customer service also has to discover the account service and then call it. In this view, you can easily find out which operation is the most time-consuming.



Conclusion

Distributed, independent microservices and centralized log monitoring make for the right solution. With tools like ELK and Zipkin, microservices monitoring seems to not be a very difficult problem to solve. There are also some other tools — for example, Hystrix and Turbine — that provide real-time metrics for the requests processed by microservices.
