Monitoring Microservices With Spring Cloud Sleuth, ELK, and Zipkin

Microservice's run in an environment isolated from the other microservices so it does not share resources such as databases or log files with them.

Here the challenge would be monitoring.

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.

Spring Boot and Spring Cloud frameworks provides very useful tools:

- ❑ Spring Cloud Sleuth
- ❑ Zipkin
- ❑ ELK -> Elasticsearch, Logstash, and Kibana

Zipkin is a distributed tracing system.

It helps gather timing data needed to troubleshoot latency problems in microservice architectures.

It manages both the collection and lookup of this data.

Applications are instrumented to report timing data to Zipkin.

The Zipkin UI also presents a Dependency diagram showing how many traced requests went through each application.

If we are troubleshooting latency problems or errors, we can filter or sort all traces based on the application, length of trace, annotation, or timestamp.

Once we select a trace, we can see the percentage of the total trace time each span takes which allows you to identify the problem application.

To start Zipkin, there are three options: using Java, Docker or running from source.

Regardless of how we start Zipkin, browse to http://your_host:9411 to find traces!

Architecture Overview

Tracers live in our 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.

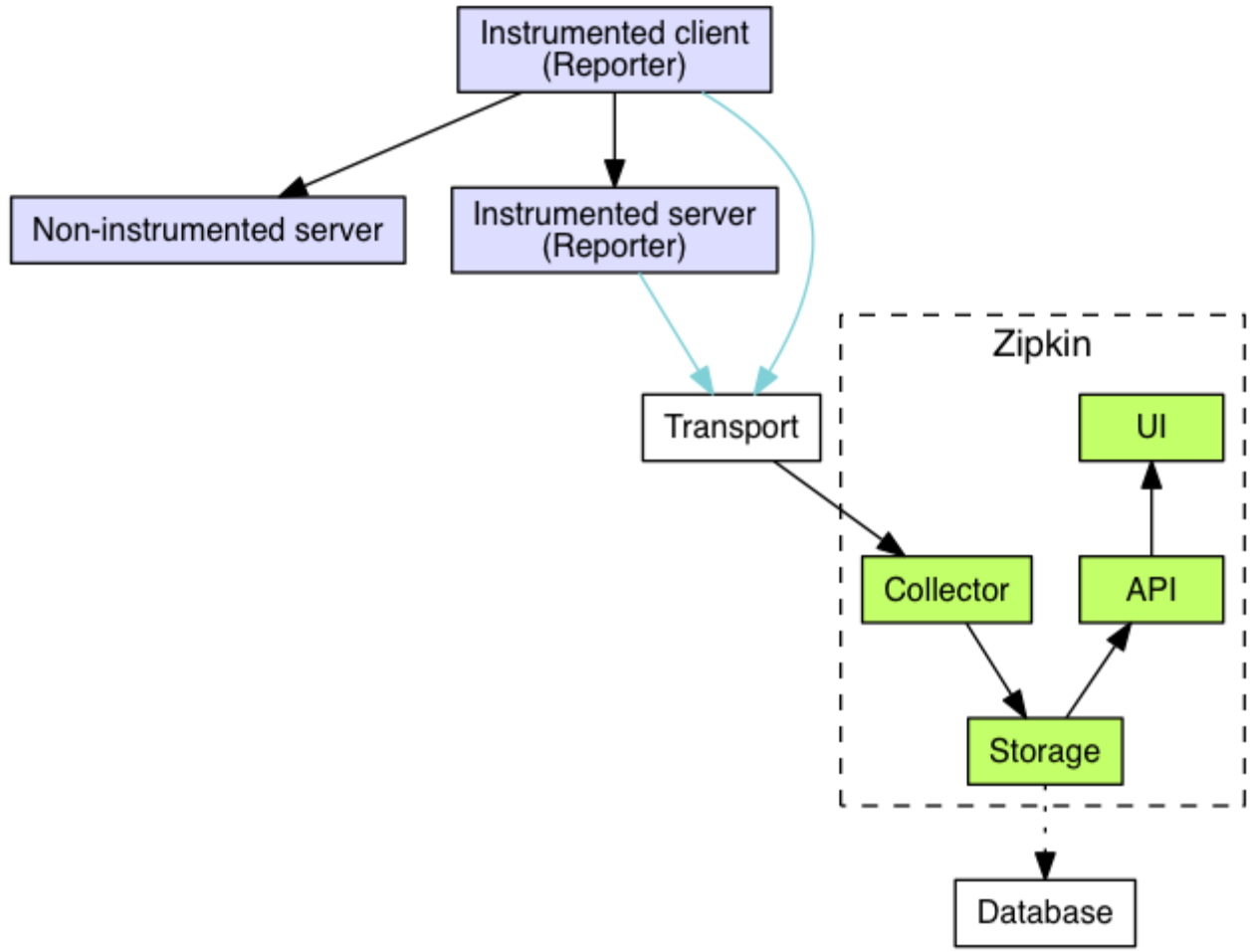
Completed spans are reported to Zipkin out-of-band, similar to how applications report metrics asynchronously.

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.

The component in an instrumented app that sends data to Zipkin is called a Reporter.

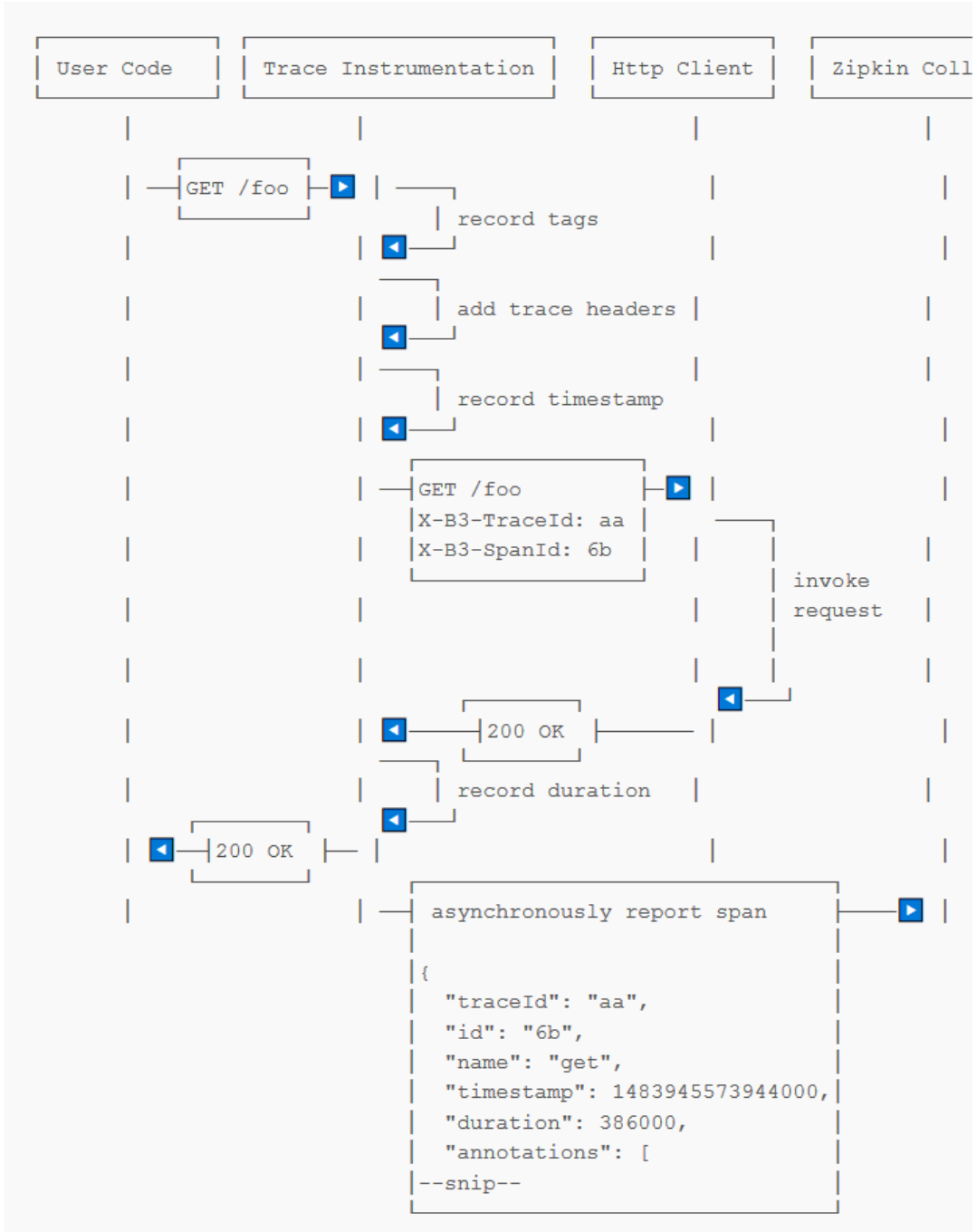
Reporters send trace data via one of several transports to Zipkin collectors, which persist trace data to storage.

Later, storage is queried by the API to provide data to the UI.


Instrumented client
(Reporter)
Non-instrumented server
Instrumented server
(Reporter)
Zipkin
Transport
UI
Collector
API
Storage
Database

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.



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.

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 is built to store data on Cassandra,ElasticSearch and MySQL.

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
The web UI provides a method for viewing traces based on service, time, and annotations.

Spring Cloud Sleuth

Spring Cloud Sleuth introduces unique IDs to the logging which are consistent between microservice calls which makes it possible to find how a single request travels from one microservice to the next.

Spring Cloud Sleuth adds two types of IDs to the logging, one called a trace ID and the other called a span ID.

The span ID represents a basic unit of work, for example sending an HTTP request. The trace ID contains a set of span IDs, forming a tree-like structure.

Spring Cloud Sleuth will send tracing information to any Zipkin server pointing to when we include the dependency spring-cloud-sleuth-zipkin in our project.

By default Sleuth assumes your Zipkin server is running at http://localhost:9411.

The location can be configured by setting spring.zipkin.baseUrl in our application properties.

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

In addition we need to tell our application how often we want to sample our logs to be exported to Zipkin.

We can do this by creating a bean for the AlwaysSampler. Add the following code to the application class.

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

ELK – Elasticsearch, Logstash, Kibana:

three different tools usually used together.

They are used for searching, analyzing, and visualizing log data in a real time.

We can use 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. The below entries should be present in every microservices projects pom.xml:

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

Kibana

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

It 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.

Docker setup:

1. set the memory for Docker instance
 Select Oracle VM Virtual Box -> System -> 4GB
Note : Stop the running instance and set it. To stop the instance select-> Close-> PowerOFF

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

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

4. docker run -d -it --name logstash -p 5000:5000 logstash -e 'input { tcp { port => 5000 codec => "json" } }
    output { elasticsearch { hosts => ["192.168.99.100"] index => "micro-%{serviceName}"} }'

5. Select Oracle VM -> settings > Network > NAT adapter > Port forwarding

Name: Demo
Protocol: TCP
Host IP : 0.0.0.0
Host Port : 5601
Guest IP:
Guest Port:

6. Access the application
http://localhost:8765/api/customer/customers

7. Kibana is available under address http://localhost:5601
Select Management -> Index patterns-> micro-*
Select Discover

8. To access Zipkin, point to the below url
http://localhost:9411/