Pattern: Distributed tracing

Context

You have applied the Microservice architecture pattern (../microservices.html). 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
- Includes the external request id in all log messages (application-logging.html)
- Records information (e.g. start time, end time) about the requests and operations performed when handling a external request in a centralized service

This instrumentation might be part of the functionality provided by a Microservice Chassis framework (../microservice-chassis.html).

Examples

The Microservices Example application (https://github.com/cer/microservices-examples) is an example of an application that uses client-side service discovery. It is written in Scala and uses Spring Boot and Spring Cloud as the Microservice chassis (microservice-chassis.html). They provide various capabilities including Spring Cloud Sleuth (https://cloud.spring.io/spring-cloud-sleuth/), which provides support for distributed tracing. It instruments Spring components to gather trace information and can delivers it to a Zipkin Server, which gathers and displays traces.

The following Spring Cloud Sleuth dependencies are configured in `build.gradle`:

```
dependencies {
  compile "org.springframework.cloud:spring-cloud-sleuth-stream"
  compile "org.springframework.cloud:spring-cloud-starter-sleuth"
  compile "org.springframework.cloud:spring-cloud-stream-binder-rabbit"
```

RabbitMQ is used to deliver traces to Zipkin.

The services are deployed with various Spring Cloud Sleuth-related environment variables set in the `docker-compose.yml`:

```
environment:
  SPRING_RABBITMQ_HOST: rabbitmq
  SPRING_SLEUTH_ENABLED: "true"
  SPRING_SLEUTH_SAMPLER_PERCENTAGE: 1
  SPRING_SLEUTH_WEB_SKIPPATTERN: "/api-docs.*|/autoconfig|/configprops|/dump|/health|/info|/metrics.*|/mappin
gs|/trace|/swagger.*|.*\\.png|.*\\.css|.*\\.js|/favicon.ico|/hystrix.stream"
```

This properties enable Spring Cloud Sleuth and configure it to sample all requests. It also tells Spring Cloud Sleuth to deliver traces to Zipkin via RabbitMQ running on the host called `rabbitmq`.

The Zipkin server is a simple, Spring Boot application:

```
@SpringBootApplication
@EnableZipkinStreamServer
public class ZipkinServer {

  public static void main(String[] args) {
    SpringApplication.run(ZipkinServer.class, args);
  }

}
```

It is deployed using Docker:

```
zipkin:
  image: java:openjdk-8u91-jdk
  working_dir: /app
  volumes:
    - ./zipkin-server/build/libs:/app
  command: java -jar /app/zipkin-server.jar --server.port=9411
  links:
    - rabbitmq
  ports:
    - "9411:9411"
  environment:
    RABBIT_HOST: rabbitmq
```

Resulting Context

This pattern has the following benefits:

- It provides useful insight into the behavior of the system including the sources of latency
- It enables developers to see how an individual request is handled by searching across aggregated logs (application-logging.html) for its external request id

This pattern has the following issues:

- Aggregating and storing traces can require significant infrastructure

Related patterns

- Log aggregation (application-logging.html) - the external request id is included in each log message

See also

- Open Zipkin (http://zipkin.io/) - service for recording and displaying tracing information
- Open Tracing (http://opentracing.io) - standardized API for distributed tracing