



Get unlimited access

Open in app



Published in OmarElgabry's Blog



Omar Elgabry

Follow

Jun 11, 2018 · 3 min read · Listen



Save



Microservices with Spring Boot — Circuit Breaker & Log Tracing (Part 4)

Handle Failures and Trace Requests



Handle Failures and Trace Requests



The Github repository for the application:

<https://github.com/OmarElGabry/microservices-spring-boot>





Get unlimited access

Open in app

A -> B (failure) -> C

Another example, lets say a service A calls a remote service R, and for some reason the remote service is down. How can we handle such a situation?

What we would like to do is stop failures from cascading down, and provide a way to self-heal, which improves the system's overall resiliency.

Hystrix is the implementation of Circuit Breaker pattern, which gives a control over latency and failure between distributed services.

The main idea is to stop cascading failures by failing fast and recover as soon as possible — *Important aspects of fault-tolerant systems that self-heal*.

So, we'll add Hystrix to gallery service, and we'll simulate a failure at image service. In the pom.xml file

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

In the spring boot main class

```
// ...
@EnableCircuitBreaker      // Enable circuit breakers

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



[Get unlimited access](#)[Open in app](#)

Hystrix watches for failures in that method, and if failures reached a threshold (limit), Hystrix opens the circuit so that subsequent calls will automatically fail. Therefore, and while the circuit is open, Hystrix redirects calls to the fallback method.

So, In the controller class, update `getGallery()` , add annotation and fallback method.





Get unlimited access

Open in app

```
throw new Exception("Images can't be fetched");
```

Now, go to the browser, and hit `localhost:8762/gallery/1`. You should get an empty gallery object (no images).

```
{
  "id": 1,
  "images": null
}
```

Sleuth

If you have, let's say 3 services, A, B and C. We made three different requests.

One request went from $A \rightarrow B$, another from $A \rightarrow B \rightarrow C$, and last one went from $B \rightarrow C$.

```
A -> B
A -> B -> C
B -> C
```

As the number of microservices grow, tracing requests that propagate from one microservice to another and figure out how a requests travels through the application can be quite daunting.

Sleuth makes it possible to trace the requests by adding unique ids to logs.

A *trace id* (1st) is used for tracking across the microservices; represents the whole journey of a request across all the microservices, while *span id* (2nd) is used for tracking within the individual microservice.

To use Sleuth, add dependency to `pom.xml` ...





Get unlimited access

Open in app

```
<artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

And In controller of gallery (or image) service, use logger to log some info.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

// ...
private static final Logger LOGGER =
    LoggerFactory.getLogger(HomeController.class);

public Gallery getGallery(@PathVariable final int id) {
    LOGGER.info("Creating gallery object ... ");
    // ...
    LOGGER.info("Returning images ... ");
    return images;
}
// ...
```

Now, when you make a request to gallery service, you should see the trace and span ids in the console logs.

```
INFO [gallery-service,adcd5217a36fe469,8639d164315daca9,false] ...
```

Thank you for reading! If you enjoyed it, please clap 🙌 for it.





Get unlimited access

Open in app

