



**HoGent**

Faculteit Bedrijf en Organisatie

Tracing in microservices met Spring Cloud Sleuth en Zipkin

Frederic Everaert

Scriptie voorgedragen tot het bekomen van de graad van  
professionele bachelor in de toegepaste informatica

Promotor:  
Harm De Weirdt  
Co-promotor:  
Geert Vandenstein

Instelling: —

Academiejaar: 2016-2017

Eerste examenperiode



Faculteit Bedrijf en Organisatie

Tracing in microservices met Spring Cloud Sleuth en Zipkin

Frederic Everaert

Scriptie voorgedragen tot het bekomen van de graad van  
professionele bachelor in de toegepaste informatica

Promotor:  
Harm De Weirdt  
Co-promotor:  
Geert Vandenstein

Instelling: —

Academiejaar: 2016-2017

Eerste examenperiode



## Samenvatting

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus.

Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

## Voorwoord





# Inhoudsopgave

<b>1</b>	<b>Inleiding .....</b>	<b>9</b>
1.1	Stand van zaken	10
1.2	Probleemstelling en Onderzoeksvragen	10
1.3	Opzet van deze bachelorproef	10
<b>2</b>	<b>Gebruikte technologieën .....</b>	<b>13</b>
2.1	Spring en Spring Boot	13
2.2	Containerisatie met Docker	14
2.3	Tracing	15
2.3.1	Spring Cloud Sleuth .....	15
2.3.2	Zipkin .....	15

<b>3</b>	<b>Onderzoek</b>	<b>17</b>
<b>3.1</b>	<b>Methodologie</b>	<b>17</b>
3.1.1	Opzetten microservices	17
3.1.2	Sampling strategieën en performantieverschillen	19
<b>4</b>	<b>Conclusie</b>	<b>21</b>
	<b>Bibliografie</b>	<b>23</b>

# 1. Inleiding

Monolithische architecturen ruimen steeds meer baan voor microservice architecturen of kortweg microservices. Een monolithische applicatie is gebouwd als een enkelvoudige, zelfstandige eenheid. Bij een client-server model, kan bijvoorbeeld de applicatie langs de server zijde bestaan uit een enkele applicatie die de HTTP requests behandelt, logica uitvoert en data ophaalt of bijwerkt in de database. Het grote probleem van monolithische applicaties is dat ze moeilijk te onderhouden zijn. Een kleine verandering in een bepaalde functionaliteit van de applicatie kan ervoor zorgen dat andere delen van de applicatie ook bijgewerkt moeten worden. De volledige applicatie moet opnieuw gebuild en gedeployed worden. Als een bepaalde functionaliteit gescaled moet worden, moet de volledige applicatie gescaled worden. Microservices bieden hiervoor een oplossing, aangezien de verschillende functionaliteiten elk een op zich zelf staande service kunnen vormen.

Bij microservices schieten echter de traditionele tools voor debugging tekort. Een enkele service kan niet het volledige beeld geven over bijvoorbeeld de performantie van de applicatie in zijn geheel. Om dit te verhelpen, kunnen requests getraced worden. Een trace is de volledige reisweg van een request die spans bevat voor alle doorkruiste microservices. Een span bestaat uit tags of metadata zoals de start- en stoptijdstippen. Deze data kan dan verzameld worden om een volledig beeld van het gedrag van de applicatie te geven.

De bedoeling van dit onderzoek is om de meerwaarde van tracing in microservices aan te tonen en ook om eventuele tekortkomingen vast te stellen. Er wordt specifiek gekeken naar tracing van requests met behulp van Spring Cloud Sleuth en Zipkin. In de praktijk gaat men niet alle requests gaan traceren om onnodige overhead te vermijden. Beslist welke requests getraced worden en welke niet wordt sampling genoemd. Wat kunnen bijvoorbeeld enkele

interessante use cases zijn waarbij er op een intelligente manier gesampled kan worden? Ten slotte visualiseert Zipkin enkel de verschillende traces, maar niet de log berichten, deze worden dan ook nog niet verzameld. Om logs te verzamelen en visualiseren wordt gekeken naar Elasticsearch, Logstash en Kibana. Er wordt aangetoond hoe deze tools in combinatie met Zipkin helpen om de complexiteit van microservices in bedwang te houden.

## 1.1 Stand van zaken

Onderzoeken rond distributed tracing systemen zijn schaars, maar de technologieën die gebruikt worden in dit onderzoek, namelijk Spring Cloud Sleuth en Zipkin, zijn gebaseerd op Google Dapper. (Sigelman e.a., 2010) Deze technische paper beschrijft in detail de werking van Google's distributed tracing systeem. Net zoals Dapper gebruikt Sleuth een annotatie gebaseerde methode om tracing toe te voegen aan requests. De terminologie is volledig overgenomen. Een trace bevat meerdere spans voor elke hop naar een andere service. Spans die dezelfde trace id bevatten, maken deel uit van dezelfde trace. Een groot ontwerpdoel van Dapper was om de overhead voor tracing zo laag mogelijk te houden. Sampling werd hierdoor geïntroduceerd. In een eerste productie gebruikten ze eenzelfde sampling percentage voor alle processen bij Google, gemiddeld één trace per 1024 kandidaten. Dit schema bleek heel effectief te zijn voor diensten met veel verkeer, maar bij diensten met minder verkeer gingen er zo interessante events verloren. Als oplossing werd toegelaten dat dit sampling percentage wel aan te passen was, ook al wou Google met Dapper dit soort manuele interventie vermijden. Spring Cloud Sleuth neemt dezelfde instelling over en laat ook toe om zelf sampling in te stellen.

## 1.2 Probleemstelling en Onderzoeksvragen

- Hoe kan tracing efficiënt toegepast worden, dus met zo weinig mogelijk overhead, bij microservices aan de hand van sampling?
- Hoe helpen Zipkin en Kibana bij het debugging proces?
- Welke logging shipper is performanter: Logstash of Fluentd?
- Fluent logger Fluency zou 4x sneller zijn dan de standaard java fluent-logger. Is dit altijd zo?

## 1.3 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk ?? wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.



## 2. Gebruikte technologieën

### 2.1 Spring en Spring Boot

Dit onderzoek maakt gebruik van Spring Boot om microservices op te stellen. Het Spring framework staat bekend voor zijn Inversion of Control (IoC) principe, ook gekend als dependency injection. Dit betekent onder andere dat een object geannoteerd kan worden met `@Component` en Spring zal het object aanmaken, alle nodige velden vullen en het object toevoegen aan de context van de applicatie. Op deze manier is het mogelijk om verschillende objecten toe te voegen aan de context, zodat ze onderling gemakkelijk samenwerken. Het IoC principe vergemakkelijk hierdoor de manier om andere bibliotheken te integreren in de applicatie.

Spring Boot maakt het gemakkelijk om op zichzelf staande, productiewaardige Spring applicaties te maken waar het niet nodig is om veel aan de configuratie te sleutelen. Het mantra van Spring Boot is dan ook *convention-over-configuration*. Dit is bijzonder handig voor microservices omdat bijvoorbeeld een eenvoudige REST applicatie aangemaakt kan worden in minder dan 20 lijnen code.

Listing 2.1: eenvoudige Spring Boot REST app

```
package hello ;

import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;
```

```
@Controller
@EnableAutoConfiguration
public class SampleController {

    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "Hello_World!";
    }

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

De `@Controller` annotatie zorgt ervoor dat de klasse gebruikt kan worden door Spring MVC om web requests te behandelen. Vanaf Spring 4.0 is het ook mogelijk om `@RestController` te gebruiken, die de annotaties `@Controller` en `@ResponseBody` combineert. De laatste annotatie is handig om `return` waarden van requests op te vangen in een zelf-gedefinieerde Java klasse. `@EnableAutoConfiguration` zegt dat Spring Boot de applicatie automatisch moet configureren op basis van de toegevoegde bibliotheken. Ten slotte is er nog de `@RequestMapping` annotatie die Spring vertelt dat het pad `/` gemapped moet worden op de `home` methode.

## 2.2 Containerisatie met Docker

Elke microservice, bijvoorbeeld een Spring Boot applicatie, kan door Docker uitgevoerd worden zijn eigen container. De verschillende containers zijn geïsoleerd van elkaar en delen enkel de minimale kernel van het besturingssysteem. Containers lijken op het eerste zicht op virtuele machines, in het opzicht dat beide geïsoleerde omgevingen zijn die beheerd worden door een controlerend proces: een container manager en hypervisor respectievelijk. Het grootste verschil tussen de twee is echter dat, voor elke virtuele machine, een volledige stack van componenten uitgevoerd dienen te worden: het besturingssysteem tot en met de applicatielaag en de virtuele hardware met netwerkkaarten, CPU's en geheugen.

Containers daarentegen functioneren eerder als volledig geïsoleerde *sandboxes*. De containers delen met elkaar de kernel van het besturingssysteem met de aanwezige systeembronnen. Dit betekent dat containers veel minder zwaar zijn op het onderliggende systeem, zodat meer containers uitgevoerd kunnen worden dan virtuele machines. Een belangrijke limitatie van containers is echter dat ze enkel uitgevoerd kunnen worden in Linux-gebaseerde besturingssystemen. Dit omdat Docker gebruik maakt van kernel isolatie, wat een specifieke Linux technologie is.



Docker kan niet rechtstreeks uitgevoerd worden op Mac of Windows systemen, maar er is een eenvoudige workaround om dit te verhelpen: Docker Toolbox. Er zal eerst een Linux virtuele machine opgestart worden in VirtualBox, waarna de docker containers uitgevoerd kunnen worden in deze virtuele machine.

## 2.3 Tracing

Een van de uitdagingen in een microservice omgeving is om het overzicht te bewaren hoe alle verschillende services met elkaar samenwerken. Omdat elke service onafhankelijk is van de volgende, is het moeilijker om het gedrag van het volledige systeem te controleren. Zoals eerder vermeld in 1 kan tracing hierbij helpen en omdat het gaat over tracing in een microservice systeem, spreekt men ook over distributed tracing.

### 2.3.1 Spring Cloud Sleuth

Spring Cloud biedt een aantal bibliotheken aan zoals Config (een plaats om configuraties van verschillende microservices te beheren), Netflix (Netflix OSS integraties voor Spring Boot apps zoals Eureka) en dus ook Sleuth. Sleuth is een distributed tracing oplossing die concepten van Dapper, Zipkin en HTrace leent.

Een trace ziet er bij Sleuth uit zoals bij Dapper. Het bevat een trace id dat gevormd wordt wanneer de eerste request gemaakt wordt. Voor elke service die de request doorkruist, wordt een span id toegekend voor die service en toegevoegd aan de trace. Sleuth voegt die id's toe aan de headers in de request response. (invoegen voorbeeld trace, met trace id en span id headers) De eerste span van een trace kent dezelfde id als de trace.

Sleuth voegt de trace en span id's ook toe aan de logging voor de microservice. Dit ziet er zo uit: `[my-service-id, 73b62c0f90d11e06, 73b62c0f90d11e06, false].my-service-id` is de naam van de service, de eerste id duidt de trace aan, de volgende is de span en de laatste waarde geeft aan of de span geëxporteerd dient te worden naar Zipkin.

### 2.3.2 Zipkin

Om de traces die Sleuth aanbiedt te verzamelen en te analyseren, komt Zipkin goed van pas. Sleuth heeft de mogelijkheid om tracing informatie te sturen naar een Zipkin server door de dependency `spring-cloud-sleuth-zipkin` toe te voegen aan de applicatie. Sleuth gaat er standaard vanuit dat de Zipkin server loopt op `http://localhost:9411`. De

locatie kan echter aangepast worden door `spring.zipkin.baseUrl` toe te voegen aan de applicatie eigenschappen (`application.properties`).

## 3. Onderzoek

### 3.1 Methodologie

#### 3.1.1 Opzetten microservices

De opstelling kent de volgende microservices:

- eureka: service discovery
- zipkin: Zipkin server voor tracing
- ms-web: Spring Boot app die root microservice voorstelt
- ms-service-n: Spring Boot app voor bijkomende microservices

Dit onderzoek bekijkt drie verschillende opstellingen, geonderscheid op complexiteit, om de tracing performantie na te gaan. De setup met lage complexiteit kent in totaal 4 microservices en 2 niveau's (include img), de setup met gemiddelde complexiteit kent 11 microservices en 3 niveau's (include img) en de setup met hoge complexiteit kent 28 microservices en 4 niveau's (include img).

De Spring Cloud Sleuth bibliotheek wordt toegevoegd aan de ms-web en ms-service-n microservices, zodat deze services tracing info doorsturen naar de zipkin server.

Elke microservice krijgt zijn eigen Docker container, die allemaal tegelijk worden opgestart met Docker Compose, dankzij het toevoegen van een docker-compose.yml script (bijlage) die alle service definities bevat.

De microservices communiceren asynchroon met elkaar, zodat de ene service niet hoeft te wachten op de andere. Dit verbetert de responstijd. Vergelijk (include img sync traces) met (include img sync traces). Om de asynchrone communicatie tussen services mogelijk te maken wordt gebruik gemaakt van `ListenableFuture` van `guava` (Google Core libraries voor Java) in combinatie met `TraceCallable` van `Sleuth`. `TraceCallable` is een wrapper van `Sleuth` voor een Java `Callable` die tracing informatie toevoegt. Een gewone `Callable` voert een taak uit op een andere thread en geeft een resultaat terug.

Listing 3.1: Asynchrone REST communicatie

```
@RestController
@RequestMapping("/")
public class SimpleRestController {
    private static final ListeningExecutorService executor = MoreExecutors

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private Tracer tracer;

    @Autowired
    private SpanNamer spanNamer;

    @RequestMapping(method = RequestMethod.GET)
    public String get() throws ExecutionException, InterruptedException {
        Callable<ResponseEntity<String>> callable1 = new Callable<ResponseEntity<String>>() {
            @Override
            public ResponseEntity<String> call() throws Exception {
                return requestService("http://ms-service-1:8080/rest/");
            }
        };
        Callable<ResponseEntity<String>> traceCallable1 = new TraceCallable<ResponseEntity<String>>
            (tracer, spanNamer, callable1, "get_service_1");

        ListenableFuture<ResponseEntity<String>> future1 = executor.submit(traceCallable1);

        ...

        // wait for results
        ResponseEntity<String> service1Response = future1.get();

        ...

        return serviceResponse.getBody();
    }

    private ResponseEntity<String> requestService(String serviceUrl) {
```

```
        return restTemplate.exchange(serviceUrl , HttpMethod.GET, null ,  
    }  
  
}
```

Een `Tracer` en een `SpanNamer` worden geïnjecteerd door `Sleuth` en een `RestTemplate` door `Spring`. Omdat `Sleuth` automatisch tracing informatie toevoegt aan een `RestTemplate`, moet dit niet verder geconfigureerd worden.

### 3.1.2 Sampling strategieën en performantieverschillen

Er wordt onderzocht wat voor overhead het toevoegen van tracing informatie met zich meebrengt op de drie verschillende testopstellingen (zie 3.1.1). Omdat `Sleuth` gebaseerd is op `Dapper`, wordt bekeken hoe verschillende sampling strategieën de performantie beïnvloeden. Uit (Sigelman e.a., 2010) blijkt immers dat verzameling van traces de grootste invloed kan hebben op vertragingen in het netwerk bij het maken van requests.

Met een gesampelde trace wordt bedoeld dat de trace gemarkeerd is om verzameld te worden. In dit onderzoek betekent dit dat `Sleuth` een gesampelde trace zal doorsturen naar de Zipkin server.

#### Percentuele sampling



## 4. Conclusie

Curabitur nunc magna, posuere eget, venenatis eu, vehicula ac, velit. Aenean ornare, massa a accumsan pulvinar, quam lorem laoreet purus, eu sodales magna risus molestie lorem. Nunc erat velit, hendrerit quis, malesuada ut, aliquam vitae, wisi. Sed posuere. Suspendisse ipsum arcu, scelerisque nec, aliquam eu, molestie tincidunt, justo. Phasellus iaculis. Sed posuere lorem non ipsum. Pellentesque dapibus. Suspendisse quam libero, laoreet a, tincidunt eget, consequat at, est. Nullam ut lectus non enim consequat facilisis. Mauris leo. Quisque pede ligula, auctor vel, pellentesque vel, posuere id, turpis. Cras ipsum sem, cursus et, facilisis ut, tempus euismod, quam. Suspendisse tristique dolor eu orci. Mauris mattis. Aenean semper. Vivamus tortor magna, facilisis id, varius mattis, hendrerit in, justo. Integer purus.

Vivamus adipiscing. Curabitur imperdiet tempus turpis. Vivamus sapien dolor, congue venenatis, euismod eget, porta rhoncus, magna. Proin condimentum pretium enim. Fusce fringilla, libero et venenatis facilisis, eros enim cursus arcu, vitae facilisis odio augue vitae orci. Aliquam varius nibh ut odio. Sed condimentum condimentum nunc. Pellentesque eget massa. Pellentesque quis mauris. Donec ut ligula ac pede pulvinar lobortis. Pellentesque euismod. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent elit. Ut laoreet ornare est. Phasellus gravida vulputate nulla. Donec sit amet arcu ut sem tempor malesuada. Praesent hendrerit augue in urna. Proin enim ante, ornare vel, consequat ut, blandit in, justo. Donec felis elit, dignissim sed, sagittis ut, ullamcorper a, nulla. Aenean pharetra vulputate odio.

Quisque enim. Proin velit neque, tristique eu, eleifend eget, vestibulum nec, lacus. Vivamus odio. Duis odio urna, vehicula in, elementum aliquam, aliquet laoreet, tellus. Sed velit. Sed vel mi ac elit aliquet interdum. Etiam sapien neque, convallis et, aliquet vel, auctor non, arcu. Aliquam suscipit aliquam lectus. Proin tincidunt magna sed wisi. Integer blandit

lacus ut lorem. Sed luctus justo sed enim.

Morbi malesuada hendrerit dui. Nunc mauris leo, dapibus sit amet, vestibulum et, commodo id, est. Pellentesque purus. Pellentesque tristique, nunc ac pulvinar adipiscing, justo eros consequat lectus, sit amet posuere lectus neque vel augue. Cras consectetur libero ac eros. Ut eget massa. Fusce sit amet enim eleifend sem dictum auctor. In eget risus luctus wisi convallis pulvinar. Vivamus sapien risus, tempor in, viverra in, aliquet pellentesque, eros. Aliquam euismod libero a sem.

Nunc velit augue, scelerisque dignissim, lobortis et, aliquam in, risus. In eu eros. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur vulputate elit viverra augue. Mauris fringilla, tortor sit amet malesuada mollis, sapien mi dapibus odio, ac imperdiet ligula enim eget nisl. Quisque vitae pede a pede aliquet suscipit. Phasellus tellus pede, viverra vestibulum, gravida id, laoreet in, justo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer commodo luctus lectus. Mauris justo. Duis varius eros. Sed quam. Cras lacus eros, rutrum eget, varius quis, convallis iaculis, velit. Mauris imperdiet, metus at tristique venenatis, purus neque pellentesque mauris, a ultrices elit lacus nec tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent malesuada. Nam lacus lectus, auctor sit amet, malesuada vel, elementum eget, metus. Duis neque pede, facilisis eget, egestas elementum, nonummy id, neque.



## Bibliografie

Sigelman, B. H., Andr, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., ... Shanbhag, C. (2010). Dapper , a Large-Scale Distributed Systems Tracing Infrastructure. *Google Research*, (April), 14. doi:dapper-2010-1



## Lijst van figuren



## Lijst van tabellen