Debugging Java in de Cloud

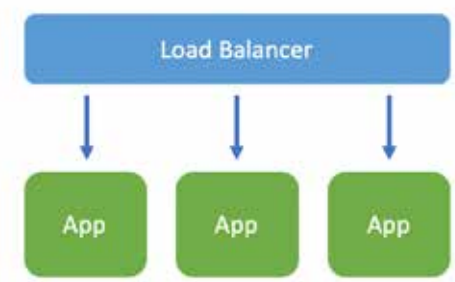
Een complicatie waar iedereen wel eens mee te maken heeft

De tijd dat we applicaties puur monolithisch bouwden hebben we alweer enige tijd achter ons gelaten. Tegenwoordig worden applicaties steeds vaker opgebouwd uit een collectie van kleinere services. Best een mooie ontwikkeling, maar het brengt ook een aantal complicaties met zich mee. In dit artikel belicht ik één van de complicaties waar iedereen direct mee te maken krijgt: debuggen.

De beste vrienden van een ontwikkelaar voor het ontrafelen van problemen met een applicatie zijn logging en breakpoints. Al sinds de begindagen van het programmeren, schrijven ontwikkelaars informatie over het gedrag van de applicatie naar logfiles. Deze informatie kan gebruikt worden voor allerlei doeleinden, maar heeft meestal als hoofdfunctie het kunnen detecteren van problemen en het geven van context om te helpen bij het vinden van de oorzaak van optredende problemen. Als logging niet genoeg informatie geeft om het opgetreden probleem te verhelpen, dan kan het plaatsen van één of meerdere breakpoints in de applicatie helpen. Hiermee kan een applicatie worden stilgezet om de interne status op dat specifieke moment te bekijken. Breakpoints zijn vooral handig tijden het ontwikkelen. Je zal ze niet gauw gebruiken in een productie omgeving om de eenvoudige reden dat je applicatie letterlijk gepauzeerd wordt met als gevolg dat er geen bewerkingen meer plaatsvinden. Wat niet wil zeggen dat het niet kan, maar daarover later meer.

Monolitisch

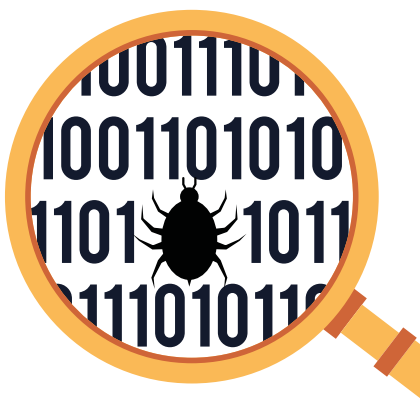
Voor we applicaties zijn gaan opsplitsen, draaiden applicaties als monoliet op een veelvoud aan servers. Als je wilde weten wat er met een request gebeurde, dan moest je uitvinden op welke server de request terecht gekomen was en kon je daar de logfile bekijken. Niet dat dit nou persé heel gemakkelijk was. Soms had je te maken met honderden servers, maar als je de juiste server gevonden had, had je ook meteen alle informatie over die request gevonden.

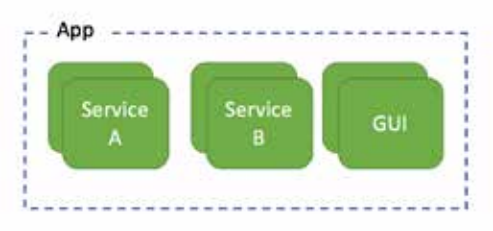


Services

Met het opsplitsen van applicaties in kleinere services die verspreidt draaien over een veelvoud aan servers wordt het een stuk ingewikkelder om de afhandeling van een request te volgen. Waar de afhandeling van een request voorheen door exact één server gedaan werd, wordt nu in veel gevallen de afhandeling door een aantal verschillende services gedaan die zich niet noodzakelijkerwijs op dezelfde server bevinden. Daarmee staat de informatie over die request ook op verschillende servers.

Bas Passon is werkzaam als senior Java developer bij First8. Op dit moment houdt hij zich vooral bezig met Cloud Native Java applicatie ontwikkeling.





Container Orkestratie

Een complicerende factor is dat veel applicaties tegenwoordig in een gedistribueerde containeromgeving draaien waarbij de verschillende services niet permanent op één vooraf gekozen server draaien. Containeromgevingen, gemanaged door containerorkestratie systemen, tegenwoordig bijna altijd Kubernetes, willen services (containers) nog wel eens verplaatsen van de ene server naar de andere. Een eindgebruiker heeft daar geen last van, maar een ontwikkelaar die logging wil inzien wel.

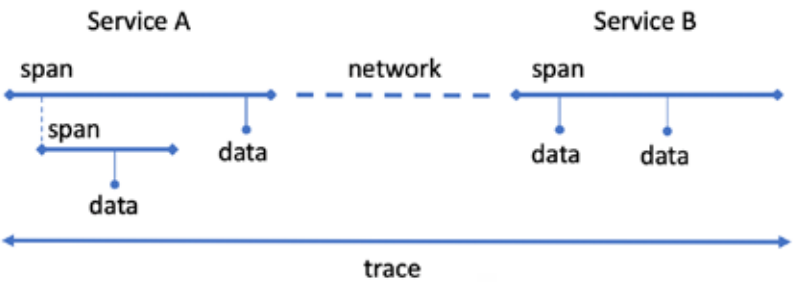
Logging Aggregatie

Met het opsplitsen van applicaties in kleinere services, waarbij deze draaien in gedistribueerde containeromgevingen, is het voor ontwikkelaars een stuk lastiger geworden om snel de benodigde log informatie te vinden. Waar voorheen inloggen op een server en daar de logfile bekijken volstond, is het nu niet altijd evident waar de log informatie te vinden is.

Geen reden tot wanhoop. Nieuwe problemen vragen om nieuwe oplossingen en hier is dat aggregatie van log informatie op een centrale plek. Als ontwikkelaar hoef je dan niet te weten waar services precies draaien, maar kan je alle log informatie op een centrale plek bekijken. Al een hele verbetering ten opzichte van het zoeken naar log informatie op de verschillende servers waar services zich zouden kunnen bevinden. We hebben nu dan wel alle log informatie op een centrale plek beschikbaar, maar het is nog niet heel eenvoudig om de log informatie te relateren aan requests van eindgebruikers. Alles staat door elkaar en het is niet direct duidelijke welke log informatie bij welke request hoort.

Distributed Tracing

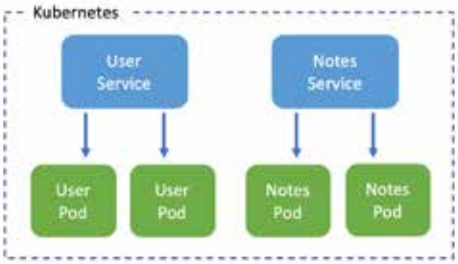
Om log informatie van requests aan elkaar te koppelen over verschillende service aanroepen heen, moeten deze op de één of andere manier aan elkaar gerelateerd kunnen worden. Dit is waar distributed tracing om de hoek komt kijken. Dit is het kunnen volgen en analyseren van een request als deze door het landschap van services beweegt. De basis hiervoor is ooit door Google beschreven in een paper met de naam: 'Dapper, a Large-Scale Distributed Systems Tracing Infrastructure' (1). In de kern komt het erop neer dat bij de start van een request een trace opgestart wordt waar services spans aan kunnen toevoegen met informatie. Als we het model visualiseren, dan krijgen we het volgende:



Het analyseren laten we hier even links liggen. Het gaat ons nu vooral om het kunnen volgen van requests. Dit kunnen we bewerkstelligen door gebruik te maken van de trace en de spans die eraan gelinkt zijn.

Spring Boot

We weten nu hoe we een applicatie opgesplitst in services draaiend in een gedistribueerd container platform kunnen debuggen. Nou ja, theoretisch dan. Laten we eens kijken wat we moeten doen om een applicatie bestaande uit een tweetal Spring Boot gebaseerde services draaiend in Kubernetes, weergegeven in onderstaande figuur, te voorzien van distributed tracing.



De applicatie bestaat uit een simpele use case. We hebben gebruikers die notes hebben opgeslagen. De user service weet alles van de

DE EVOLUTIE VAN MONOLIET NAAR (MICRO) SERVICES EN CLOUD HEEFT EEN HEEL NIEUW SCALA AAN MOGELIJKHEDEN GECREËERD

users en de notes service weet alles van de notes van gebruikers. Een aanroep naar de user service haalt de informatie van de user op inclusief zijn notes, die opgehaald worden bij de notes service. Niet de meest interessante use-case, maar het volstaat voor ons doel: het volgen van requests door het service landschap. Hoe pakken we dat aan? Alles begint met Spring Cloud Sleuth.

Spring Cloud Sleuth

Spring Cloud Sleuth (3) is een distributed tracing implementatie voor het Spring platform. Het maakt gebruik van de ideeën van Google's Dapper paper en bouwt op ZipKin en HTrace, beide distributed tracing frameworks. Sleuth faciliteert tracing door op alle -binnen een Spring Boot applicatie gevonden koppelvlakken- interceptors te plaatsen die zorg te dragen voor het aanmaken van traces en spans en deze te voorzien van analyse informatie. Om Sleuth te gebruiken, moet de dependency van **listing 1** toegevoegd worden aan de project pom.xml.
Met deze dependency op het classpath wordt runtime het logging formaat van je applicatie aangepast, zodat trace en span hierin komen. Dat ziet er dan als **listing 2** uit.

Het verschil met het standaard logging formaat is de toevoeging van **[user-service,1af74bee2eef448c,1af74bee2eef448c,false**] achter het level. Deze informatie wordt door Sleuth ingevoegd en is de basis voor het kunnen volgen van requests. De betekenis van deze waarden is **[service-identifier,trace-id,span-id,exported].** De eerste drie zijn interessant voor het volgen van requests, exported is nuttig als je ook analyses doet met een analyse server, b.v. ZipKin. De betreffende trace is dan geëxporteerd voor analyse doeleinden.

Nu we in de logs waarden hebben die we kunnen gebruiken om log output te koppelen aan een unieke request, moeten we deze nog wel ergens aggregeren. De log output staat immers nu nog op verschillende plekken in het Kubernetes cluster en dat maakt het volgen van de request onhandig. Voor het aggregeren van de log output zijn verschillende opties. Hier kiezen we voor de welbekende Elastic Stack (4) bestaande uit Elasticsearch, Logstash en Kibana.

Elastic Stack

Elasticsearch, Logstash, Filebeat en Kibana zijn vier open source producten die samen een krachtige log analyse oplossing vormen. Elasticsearch is een NoSQL database en uitermate geschikt voor het opslaan van log informatie. Filebeat verzamelt log informatie. Logstash ontvangt en transformeert log informatie en Kibana levert visualisatie en analyse mogelijkheden.

Waar het in de begindagen van cloud computing nog best een uitdaging was om de Elastic Stack aan de praat te krijgen, is dat tegenwoordig een stuk eenvoudiger geworden met de komst van Helm. Dit is een package manager voor Kubernetes. Met behulp van de Helm chart (5) voor Elastic Stack zijn we vrij snel up en running. Na de initiële configuratie van Kibana komen de log regels van de service binnen. Echter, de informatie is nog niet opgesplitst in losse velden waar we op kunnen zoeken en filteren. Om dit voor elkaar te krijgen, moet de configuratie van Logstash aangepast worden. We maken gebruik van de Grok filter plugin.

Grok

De Grok filter plugin kan op basis van reguliere expressies velden extraheren uit de ruwe log informatie. Het opstellen van de juiste reguliere expressie heeft altijd wat voeten in aarde. Gelukkig is voor alles een app. Zo is hier Grok Debug (6) om ons te helpen. Na wat pogingen volstaat de expressie van **listing 3**,

Nu Logstash de verschillende velden uit de log informatie haalt, kunnen we in Kibana gebruik maken van de losse velden. Als we een trace isoleren, ziet dat er als **afbeelding 1** uit.

NIEUWE PROBLEMEN VRAGEN OM NIEUWE OPLOSSINGEN

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

Listing 1: dependency

```
2018-12-10 22:48:45.354  INFO [user-service,1af74bee2eef448c,1af74bee
2eef448c,false]
    13522 --- [ctor-http-nio-2] nl.first8.debug.UserController
: Getting user for name joe.
```

Listing 2: trace en span

```
%{TIMESTAMP_ISO8601:timestamp} *%{LOGLEVEL:level} \[%{DATA:applicati
on},%{DATA:trace},%{DATA:span},%{DATA:exported}] %{DATA:pid} --- *\
[%{DATA:thread}] %{JAVACLASS:class} *: %{GREEDYDATA:message}
```

Listing 3: trace en span

Time	level	service	trace	span	message
December 12th 2018, 21:32:40.476	INFO	notes-service	58b1f9deef36c7cb	c27fdeabad03b5eb	Finding notes for user bosp
December 12th 2018, 21:32:40.462	INFO	user-service	58b1f9deef36c7cb	58b1f9deef36c7cb	Getting user for name bosp.

Afbeelding 1

We kunnen nu de afhandeling van een request volgen zonder dat we hoeven te weten waar de verschillende services zich bevinden. Alle informatie is op basis van trace aan elkaar te koppelen. Het bij elkaar halen van relevante log informatie in het geval van problemen is nu wel heel eenvoudig geworden.

Bonus: Breakspoints in Kubernetes

Zoals al eerder aangaven, is log informatie soms niet genoeg om erachter te komen wat er aan de hand is. Soms wil je een breakpoint kunnen gebruiken om in de service zelf te kijken. Iets wat lokaal heel eenvoudig is, lijkt in een Kubernetes omgeving haast onmogelijk, maar niets is minder waar. De Kubernetes command line utility, kubectl, komt met een standaard port-forward functionaliteit. Deze kunnen we gebruiken om een lokale poort door te lussen naar een port op een container in het cluster. Dat kan natuurlijk ook de debug poort van de JVM zijn. Voegen we de volgende argumenten toe aan de JVM in de container om deze in debug modus te starten:

```
-Xdebug
-Xrunjdwp:server=y,transport=dt_
socket,address=5000,suspend=n
```

De JVM luistert op poort 5000. Hier kan een remote debugger dan op verbinden. Het is goed even extra te vermelden dat je de waarde van suspend echt op n laat om te voorkomen dat de JVM gaat staan wachten tot een remote debugger verbinding maakt. Je wilt uiteindelijk alleen indien nodig verbinden met één van de vele containers. Stel, je wilt een breakpoint zetten in één van de user-service containers uit het eerdere voorbeeld met id user-service-765d459796-258hz, dan kunnen we de de port-forward als volgt opzetten:

```
kubectl port-forward user-ser-
vice-765d459796-258hz 5000:5000
```

Nu kan met een remote debugger verbinding gemaakt worden op localhost:5000 voor het zetten van breakpoints. Zo eenvoudig is het! Let wel, de JVM is dan gepauzeerd en zal geen requests meer afhandelen die er naartoe doorgezet worden, zolang je een andere request aan het debuggen bent.

Conclusie

De evolutie van monoliet naar (micro)services en cloud heeft een heel nieuw scala aan mogelijkheden gecreëerd voor het bouwen van oplossingen en daarmee ook een heel scala aan nieuwe beheer uitdagingen. Logging is er één van en was in de tijd van de monoliet al een uitdaging die er met de komst van (micro)services en cloud niet eenvoudiger op geworden is. In dit artikel heb ik laten zien dat het logging probleem goed op te lossen is met een aantal populaire open source oplossingen. Naast logging hebben we het kort gehad over debuggen met breakpoints. Met wat kunst en vliegwerk is ook dat voor elkaar te krijgen. Het heeft echter wel als nadeel dat eindgebruikers er direct hinder van ondervinden. Zelf zal ik het dan ook alleen in uiterste noodgevallen inzetten op een productie omgeving. ■

ALLES BEGINT MET SPRING CLOUD SLEUTH

REFERENTIES:

1. **https://ai.google/research/pubs/pub36356**
2. **https://spring.io/projects/spring-boot**
3. **https://spring.io/projects/spring-cloud-sleuth**
4. **https://www.elastic.co/**
5. **https://github.com/helm/charts/tree/master/stable/elastic-stack**
6. **https://grokdebug.herokuapp.com/**