Spring boot dag 2 - Cloud lab

We willen de volgende microservice architectuur neerzetten.



Het ziet er een beetje ingewikkeld uit, maar we willen het volgende:

* De config server managed alle externe configuratie van de services middels een git repo.
* De time service geeft de tijd terug als string, bijvoorbeeld “Het is nu 12:38:00”. Het statische deel “Het is nu” moet als configuratie uit de config server worden gehaald. Dus verzin bijvoorbeeld een andere tekst zoals “Volgens mij is het….” om te testen of de configuratie inderdaad werkt.
* De eureka server is de service registry en gaat het up of down zijn van alle microservices bijhouden.
* We maken een greeting service die een tekst teruggeeft “Greetings from port xxxx!” Waarbij de port de server port is waaronder de service is opgestart. We starten meerdere (iig 2) instanties. Deze service registreert zich ook bij de Eureka server.
* Voor de time service plaatsen we een zuul proxy die het centrale entrypoint voor onze microservices wordt.
* Eventueel kan je nog een trage service opzetten zodat je met een circuit breaker kan spelen. Dit is een bonus opdracht.



# Algemeen

We gaan dus 5 SpringBoot applicaties opzetten:

* Config server
* Eureka server
* Time service
* Greeting service
* Zuul proxy
* Evt trage service

Het is handig om hiervoor elke keer het volgende werkende skelet te clonen <https://github.com/MichelSchudel/springboot-skeleton> en dan de artifactnaam aan te passen en vervolgens de boel uit te breiden.

Je kan ook zelf elke keer een project vanaf scratch maken. Denk er dan aan in maven dependency management altijd de volgende configuratie te plaatsen:

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Brixton.SR7</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>1.5.1.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

# Config server opzetten

De configuratie van de services gaan we externaliseren middels een config server.

## Hints

* Nieuw maven project met als dependency management bovenstaande, en als enige dependency spring-cloud-config-server.
* Main spring boot class schrijven, gelijk aan de structuur van de service van dag 1, met annotaties @SpringBootApplication en @EnableConfigServer.
* Draai deze applicatie op poort 8888 middels server.port property in je application properties.
* Bij starten krijg je nu een melding dat er geen GIT uri opgegeven is. Maak daarom een nieuwe git repo aan op je file system, bv “spring-config-repository”. Verwijs middels de application property spring.cloud.config.server.git.uri naar deze repo (file://……).
* Plaats een property file genaamd time-service.properties in de root van de git repo. Dit wordt de externe configuratie van de time service. Plaats een property die door de time service wordt gebruikt (bv app.greeting=Volgens mij is het) in het bestand en en doe een commit.
* Start de applicatie (opnieuw). Je kunt op http://localhost:8888/time-service/default de default configuratie van de time service benaderen.

# Time service maken die van de config server gebruik maakt

* Nieuw maven project time-service maken
* maven pom met als dependencies spring-boot-starter-web en spring-cloud-starter-config.
* In je application properties dien je de property spring.application.name op te nemen met waarde time-service. Zo weet de service welke configuratie van de config server opgehaald moet worden.
* Schrijf een restcontroller met requestmapping “/time” die mbv van de config server de huidige tijd weergeeft. De output wordt dus iets als “Volgens mij is het 12:34:56”.
* Time service starten, deze zou nu de externe configuratie van de config server op moeten pikken. Op <http://localhost:8080/time> krijg je nu het antwoord.

# Eureka server maken die instanties van alle services bijhoudt

* Nieuw maven project eureka-server maken.
* Maven dependencies: spring-cloud-starter-config en spring-cloud-starter-eureka-server
* Voeg een application.yaml toe aan src/main/resources met onderstaande vulling. Dit zorgt ervoor dat de Eureka server zich niet bij zichzelf probeert te registreren!

server:

port: 8761

eureka:

client:

registerWithEureka: false

fetchRegistry: false

server:

waitTimeInMsWhenSyncEmpty: 0

enableSelfPreservation: false

healthcheck:

enabled: true

* Main springboot class schrijven met @SpringBootApplication en @EnableEurekaServer als annotaties, en de gebruikelijke main method.
* Run de applicatie. Op <http://localhost:8761/> staat de eureka interface.

# Time service registeren bij de Eureka server

Pas de time service zo aan dat deze zichzelf bij de Eureka Server registreert.

* spring-cloud-starter-eureka aan pom toevoegen
* @EnableDiscoveryClient annotatie toevoegen aan main class
* Applicatie herstarten.
* Check op http://localhost:8761/ of de timeservice inderdaad is geregistreerd

# Greeting service meerdere instanties

Op <https://github.com/MichelSchudel/springboot-localized-echo-service> staat een greeting service die een groet geeft met daarin het poortnummer. Het gemakkelijkste is om dit project even te clonen en te bouwen. Je mag natuurlijk ook zelf een echo service vanaf scratch schrijven.

* Zorg ervoor dat de ConfigServer en de EurekaServer draaien.
* Start meerdere instanties (bv 3) middels java -jar target\springboot-localized-greeting-service-1.0-SNAPSHOT.jar --server.port=9500. Bij de andere instanties kan je dan de server.port system property aanpassen naar 9501, 9502 etc. Wat ook kan: in bootstrap.properties server.port=0 zetten. Spring kiest dan random een poort.
* Kijk op <http://localhost:8761/> of de service instanties inderdaad zijn geregistreerd.
* Benader er eentje, bv <http://localhost:9500/greeting> om te kijken of er eentje werkt.

# Gebruik maken van het greeting service cluster in de time service

We gaan de time service zo uitbreiden dat het /time endpoint wordt uitgebreid met een greetings boodschap uit het greetings cluster. We maken gebruik van de NetFlix Ribbon load balancer (<http://cloud.spring.io/spring-cloud-static/Brixton.SR7/#spring-cloud-ribbon>) zodat requests naar de greeting instanties Round-Robin geloadbalanced worden.

* spring-cloud-starter-ribbon dependency opnemen in de pom van de time service.
* In de timeservice en class aanmaken geannoteerd met @Configuration die middels een method geannoteerd met @Bean @LoadBalanced een RestTemplate teruggeeft.
* Deze RestTemplate injecteren in je Time controller met annotaties @Autowired @LoadBalanced.
* De resttemplate is nu aan te roepen middels de methode getForObject. Het eerste argument is een “logische” url bv <http://my-service-name/request-mapping-endpoint>, (bv als de spring.application.name van je greeting service greeting-service is, wordt dit http://greeting-service/greeting) het tweede argument bv String.class.
* Voeg de tekst uit de greeting service en de time service samen tot één geheel.
* Herstart de time service applicatie. Op het /time endpoint zie je nu alternerend antwoord uit verschillende instanties van de greeting service.

# Zuul proxy installeren

Start een Zuul proxy waarbij alle urls die beginnen met /time naar de time service worden gerouteerd.

* Nieuwe SpringBoot applicatie maken.
* In de pom de dependencies spring-cloud-starter-zuul en spring-cloud-starter-eureka opnemen.
* @EnableZuulProxy als extra annotatie in de main class, en server.port in de applicatie properties op 9000 zetten. In de regel zet je daar nog bv een apache instantie voor die alles op poort 80/443 doorzet naar de Zuul proxy maar dat laten we vandaag even achterwege.
* Volgende als routing toevoegen voor de time service. Je moet nu op localhost:9000/time de time service kunnen bereiken.

zuul:

routes:

time-service:

path: /time/\*\*

# Bonus: circuit breaker

Als extraatje kan je een circuit breaker bouwen die tegen een trage service praat.

De trage service kan je hier uit GitHub clonen:

<https://github.com/MichelSchudel/springboot-trage-service>

Deze start op een random port, dus je dient de service via Eureka op te zoeken.

* Voeg spring-cloud-starter-hystrix en spring-cloud-starter-hystrix-dashboard als dependency toe aan je time service.
* Je kan nu een nieuwe controller met request mapping /traag bouwen die gaat praten tegen de trage service, maar je kan bijvoorbeeld ook je time service zodanig uitbreiden dat tegen de trage service wordt gepraat. Je kan dit op dezelfde manier doen als de rest call naar de greeting service.
* Om nu de circuit breaker te activeren moet je @EnableCircuitBreaker en @EnableHystrixDashboard als annotaties aan je main class toevoegen.
* Aan je controller method kan je nu een HystrixCommand toevoegen, bv als volgt:

@HystrixCommand(fallbackMethod = "fallback", commandProperties = {

@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000")

,@HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "5000")

})

* Schrijf nu ook een method genaamd fallback met dezelfde signature als je controller method die bijvoorbeeld “call mislukt!” teruggeeft.
* Als je nu naar localhost:8080/traag gaat zie je dat de fallback method gebruikt wordt.
* Ga nu naar localhost:8080/hystrix om het dashboard te openen. Vul bij de monitor in: localhost:8080/hystrix.stream. Nu zie je de monitoring op de circuit breaker.
* Als je nu het /traag endpoint een keer of 20 achter elkaar benadert, zie je de circuit breaker openspringen.