SVEUČILIŠTE U ZAGREBU

**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

**Mikrousluge - Spring Cloud**

Laboratorij telekomunikacija i informatike 1: Raspodijeljeni sustavi

Kristijan Ilišinović, Luka Jurić, Marko Pobi, Jakov Živković

Zagreb, siječanj 2018.

Sadržaj

[1. Uvod 1](#_Toc504407822)

[2. Spring Framework i Spring Boot 3](#_Toc504407823)

[2.1. Spring Framework 3](#_Toc504407824)

[2.2. Spring Boot 6](#_Toc504407825)

[2.3. Spring Cloud 7](#_Toc504407826)

[2.4. Spring Cloud Config 8](#_Toc504407827)

[2.5. Spring Cloud Bus 9](#_Toc504407828)

[2.6. Implementacija 9](#_Toc504407829)

[2.6.1. Spring Cloud Config 9](#_Toc504407830)

[2.6.1.1. Poslužiteljski dio 10](#_Toc504407831)

[2.6.1.2. Klijentski dio 11](#_Toc504407832)

[2.6.2. Spring Cloud Bus 11](#_Toc504407833)

[2.6.2.1. Poslužiteljski dio 12](#_Toc504407834)

[2.6.2.2. Klijentski dio 13](#_Toc504407835)

[3. Logging u mikrouslugama 14](#_Toc504407836)

[3.1. Sleuth 15](#_Toc504407837)

[3.2. Zipkin 18](#_Toc504407838)

[3.3. ELK stack 21](#_Toc504407839)

[3.4. Upute za pokretanje 29](#_Toc504407840)

[4. Registracija i pronalazak usluga, rasterećenje sustava i oporavak od ispada usluga 30](#_Toc504407841)

[4.1. Eureka 30](#_Toc504407842)

[4.1.1. Eureka poslužitelj 30](#_Toc504407843)

[4.1.2. Eureka klijent 31](#_Toc504407844)

[4.2. Hystrix 33](#_Toc504407845)

[4.3. Ribbon 34](#_Toc504407846)

[4.4. Feign 35](#_Toc504407847)

[4.4.1. Primjer Feigna s Ribbonom i Hystrixom 35](#_Toc504407848)

[4.5. Ovisnosti 37](#_Toc504407849)

[5. Pristup mikrouslugama 38](#_Toc504407850)

[5.1. Zuul 39](#_Toc504407851)

[5.2. Spring Cloud Gateway 41](#_Toc504407852)

[5.3. Implementacija Spring Cloud Gateway servisa 45](#_Toc504407853)

[5.4. Generiranje jar datoteke 47](#_Toc504407854)

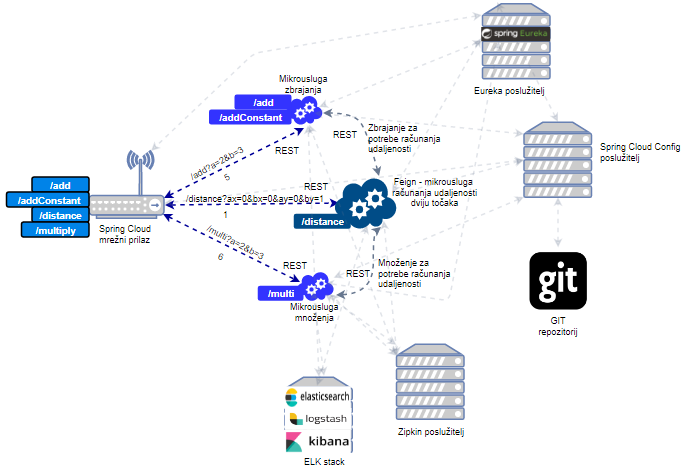
[6. Pokretanje 48](#_Toc504407855)

[7. Zaključak 50](#_Toc504407856)

[8. Literatura 51](#_Toc504407857)

# Uvod

Mikrousluge (*engl. microservices)* predstavljaju arhitekturu za razvoj aplikacija koja se sastoji od niza manjih usluga od kojih se svaka usluga izvršava u neovisnom procesu i ne moraju se nalaziti na istom računalu. Svaka mikrousluga je neovisna, zamjenjiva i može se samostalno nadograditi. One međusobno ne utječu jedna na drugu zbog sučelja između njih koje se ne mijenja u slučaju promjene u implementaciji na nekoj od usluga. Komunikacija se odvija otvorenim protokolima kao što su HTTP/REST, AMPQ i drugi. U sustavu za isporuku mikrousluga koje su implementirane u sklopu ovoga projekta korištene su sljedeće tehnologije. Za razvoj mikrousluga korišten je *Spring Boot*, potprojekt Springa, te *Spring Framework* koji su opisani u drugom poglavlju. Oni su nužni za uspješan kratak pregled te uvod u tehnologije *Spring Cloud Config* i *Spring Cloud Bus* čija su osnovna programska baza. U trećem poglavlju je opisana tehnologija *Spring Cloud* koji isporučuje mikrousluge krajnjim korisnicima te njezini potprojekti: *Spring Cloud Config* koji služi za konfiguriranje usluga te *Spring Cloud Bus* koji služi za ažuriranje konfiguracija*.* U četvrtom poglavlju se nalazi *logging*, odnosno zapisivanje podataka koji su bitni za otklanjanje poteškoća tijekom izvođenja operacija mikrousluga. Nakon toga slijedi poglavlje u kojemu je objašnjena registracija i pronalazak usluga te ponašanje sustava i oporavak od ispada usluga. Pri tome su korišteni *Eureka*, *Hystrix*, *Ribbon* i *Feign*. U šestom poglavlju su objašnjene tehnologije za pristup registriranim uslugama. Tu su korišteni *Zuul* i *Spring Cloud Gateway*. U zadanjem poglavlju je objašnjeno konkretno pokretanje napravljenog sustava i svih njegovih komponenti. Prikaz cijelog sustava koji će biti detaljno objašnjen u nastavku ovoga rada prikazan je na slici u nastavku.

*Slika 1. Skica sustava*

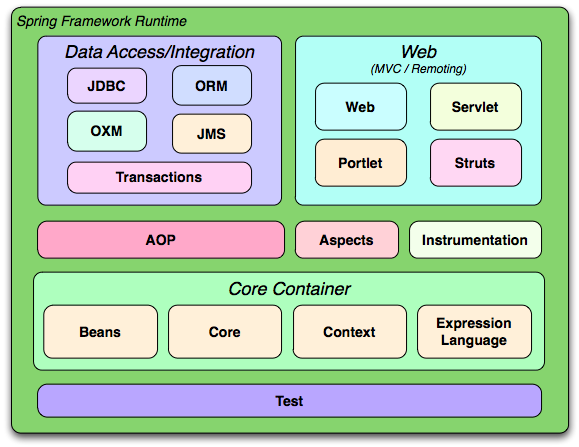
# Spring Framework i Spring Boot

U ovom će poglavlju biti ukratko opisani Spring Framework i Spring Boot

## Spring Framework

*Spring Framework* vrlo je popularan radni okvir koji služi za jednostavno podizanje Java aplikacija koje su namijenjene za komercijalne svrhe (*eng. Java enterprise application*). *Spring Framework* osigurava svu potrebnu opsežnu infrastrukturu za korištenje programskog jezika Java u komercijalnom okruženju. Pojavio se 2003. godine, a zahvaljujući konceptu otvorenog koda (*eng. open-source*) te velikoj zajednici aktivnih korisnika koji kontinuirano šalju povratne informacije (*eng. feedback*) o iskustvu korištenja Springa u najrazličitijim stvarnim scenarijima, neprestano evoluira i razvija nove sestrinske projekte kao što su *Spring Cloud* te *Spring Cloud Config* i *Spring Cloud Bus*.

Arhitektura *Spring Frameworka* modularna je što omogućuje korisnicima veliku fleksibilnost pri odabiru potrebne strukture vlastitih aplikacija. Postoji dvadesetak modula koji su povezni u veće cjeline. Grafički prikaz osnovnih modula prikazan je na sljedećoj slici (Slika 1).

**

*Slika 2. Spring Framework moduli*

Moduli su grupirani u sljedeće grupe:

* *Core Container* - sadrži fundamentalne dijelove radnog okvira uključujući zamjenu kontrole (*eng. Inversion of Control, IoC*) te injektirajuće ovisnosti (*eng. Dependency Injection)*
* *Data Access/Integration* - sadrži module za pristup i manipulaciju podatcima (*eng. Java DataBase Conectivity - JDBC, Object-relational mapping - ORM, ...*)
* *Web* - sadrži osnovne značajke za web-orijentiranu integraciju aplikacija te podršku za MVC (*eng. Model-View-Controller)* arhitekturni stil web aplikacija
* *AOP (eng. Aspect Oriented Programming)* i *Instrumentation*
* *Test* - podrška za testiranje programa koristeći *JUnit* ili neki drugi testni okvir

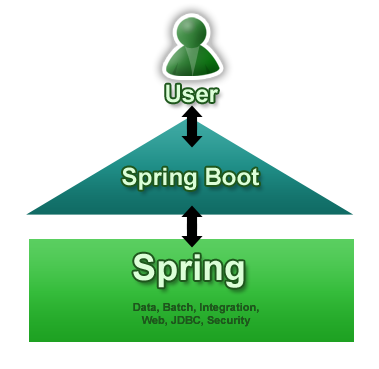
U nastavku slijedi popis principa koje karakteriziraju *Spring Framework* radni okvir i koji ga čine iznimno privlačnim odabirom za izgradnju komercijalnih aplikacija.

1. *Spring* pruža mogućnost izbora na svakom nivou te potiče odlučivanje o konkretnim implementacijama što je kasnije moguće u kodu.
2. Prilagođava se različitim perspektivama. Nema jednoznačno određen recept kako bi se trebalo radit već je fleksibilan i podupire širok spektar aplikacijskih potreba.
3. Brine se o dizajnu aplikacijskog programskog sučelja (*eng Application Program Interface, API).* Nastoji tako oblikovati *API* da je što intuitivniji i da je održiv kroz razne verzije.
4. Postavlja iznimno visoke standarde za kvalitetu kodu. Stavlja naglasak na značajne, točne i precizne *Javadoc* komentare.

*Spring Framework* radni okvir svrstava se trenutno u jedan od najpopularnijih radnih okvira za razvijanje svih vrsta, ali ponajviše web aplikacija. Njegovoj iznimnoj popularnosti doprinose sve, gore navedene karakteristike, koje omogućuju razvijateljima aplikacija da se isključivo brinu o poslovnoj logici njihove aplikacije, a sve ostale manje zanimljive stvari prepuste *Springu* koji će to odraditi za njih i pružiti fleksibilnost kod odabira arhitekture aplikacije.

## Spring Boot

Iako za *Spring framework* možemo reći da uvelike olakšava pokretanje i konfiguraciju Java aplikacija, *Spring Boo*t dodatno pojednostavljuje uporabu *Springa* te za korisnika predstavlja malenu pristupnu površinu koja izvlači najbitnije vrijednosti iz velikog *Spring* okruženja. To nam najzornije predočuje sljedeća slika (Slika 3).



*Slika 3. Spring Boot*

Primarni i ciljevi *Spring Boota* su sljedeći:

* pružanje iznimno brzog i široko dostupnog početnog iskustva konfiguracije za sve projekte vezane uz Spring Framework radni okvir.
* nudi kreativne i inventivne postavke prema jasno određenom konceptu, ali vrlo se brzo prilagođava najraznovrsnijim korisničkim zahtjevima za konfiguriranje aplikacije.
* nudi širok spektar nefunkcijskih odlika i prednosti poput ugrađenih poslužitelja, sigurnosti, metrike, mogućnosti vanjske konfiguracije...

*Spring Boot* polazi od jasno definiranog i ustaljenog koncepta za podizanje produkcijski spremnih aplikacija. Favorizira konvenciju ispred konfiguracije i dizajniran je za što brže i jednostavnije pokretanje.

## Spring Cloud

*Spring Cloud* nastao je kao projekt istog razvojnog tima kao i *Spring Framework* i *Spring Boot*, a pruža sve potrebne alate i biblioteke koje razvojnim programerima omogućuju brzo podizanje i izgradnju obrazaca u raspodijeljenim sustavima (*eng. distributed systems*). Raspodijeljeni sustav je model u kojem su komponente distribuirane u mreži te međusobnu komunikaciju i međudjelovanje usklađuju slanjem i primanjem poruka. *Spring Cloud* gradi se na *Spring Bootu* pružajući mnoštvo biblioteka koje poboljšavaju rad aplikacije. Korisniku se nudi osnovno, zadano ponašanje koje omogućuje brzo podizanje ali se također nudi i mogućnost konfiguracije i proširenja za stvaranje prilagođenog rješenja.

U nastavku se nalazi popis najznačajnijih karakteristika *Spring Clouda*:

* distribuirana konfiguracija
* prijava i otkrivanje usluga
* pozivi od usluge do usluge (*eng. service-to-service*)
* balansiranje opterećenja
* prekidači
* globalne brave
* distribuirana komunikacija
* odabir vodstva

*Spring Cloud* ima vrlo deklarativan pristup te se samom promjenom c*lass-patha* i/ili dodavanjem anotacije mogu jednostavno dodati nove značajke. Glavni moto koji stoji iza *Spring Cloud* sustava jest: "Izgradnja aplikacija u oblaku trebala bi biti jednostavnija od izgradnje monolitnih, jednoprocesorskih aplikacija.", te se toga nastoji pridržavati. U tome mu uvelike olakšavaju potprojekti: *Spring Cloud Config* i *Spring Cloud Bus* U nastavku slijedi njihov opis.

## Spring Cloud Config

*Spring Cloud Config* pruža podršku na klijentskoj i poslužiteljskoj strani za eksternaliziranu konfiguraciju u raspodijeljenim sustavima. Uporabom *Cloud Config Server* poslužitelja korisniku se na jednom mjestu nudi mogućnost upravljanje postavkama aplikacije za čitavi raspodijeljeni sustav. Koncept se, kako na klijentu, tako i na poslužitelju, jednako dobro preslikava na aplikacije pisane za *Spring* okruženje, ali i za bilo koje drugo okruženeu ili jezik. Korištenjem *Spring Cloud Configa* alternativne implementacije i promjene koje one uzrokuju za samu konfiguraciju aplikacije se na vrlo jednostavan način dodaju i uključuju u *Spring* konfiguraciju aplikacije.

Slijedi popis najznačajnijih karakteristika *Spring Cloud Config* poslužiteljskog dijela:

* HTTP aplikacijsko programsko sučelje (*eng. API*) temeljeno na resursima (parovi ime-vrijednost ili ekvivalentni *YAML* sadržaj)
* šifriranje i dešifriranje vrijednosti entiteta (simetrično ili asimetrično)
* jednostavna ugradnja u program *Spring Boot* koristeći *@EnableConfigServer* anotaciju

Najznačajnije karakteristike *Spring Cloud Config* klijentskog dijela:

* povezivanje na *Spring Cloud Config Server* poslužitelj
* inicijalizacija *Spring* okruženja (*eng. Environment*) s udaljenim izvorima entiteta
* šifriranje i dešifriranje vrijednosti entiteta (simetrično ili asimetrično)

Konfiguracijske datoteke spremaju se uporabom *EnvironmentalRepository* strategije. Zadana implementacija *EnvironmentalRepository* strategije u pozadini koristi sustav *Git*, što je vrlo pogodno za upravljanje nadogradnjama i fizičkim okruženjima, kao i za reviziju promjena konfiguracijskih datoteka.

## Spring Cloud Bus

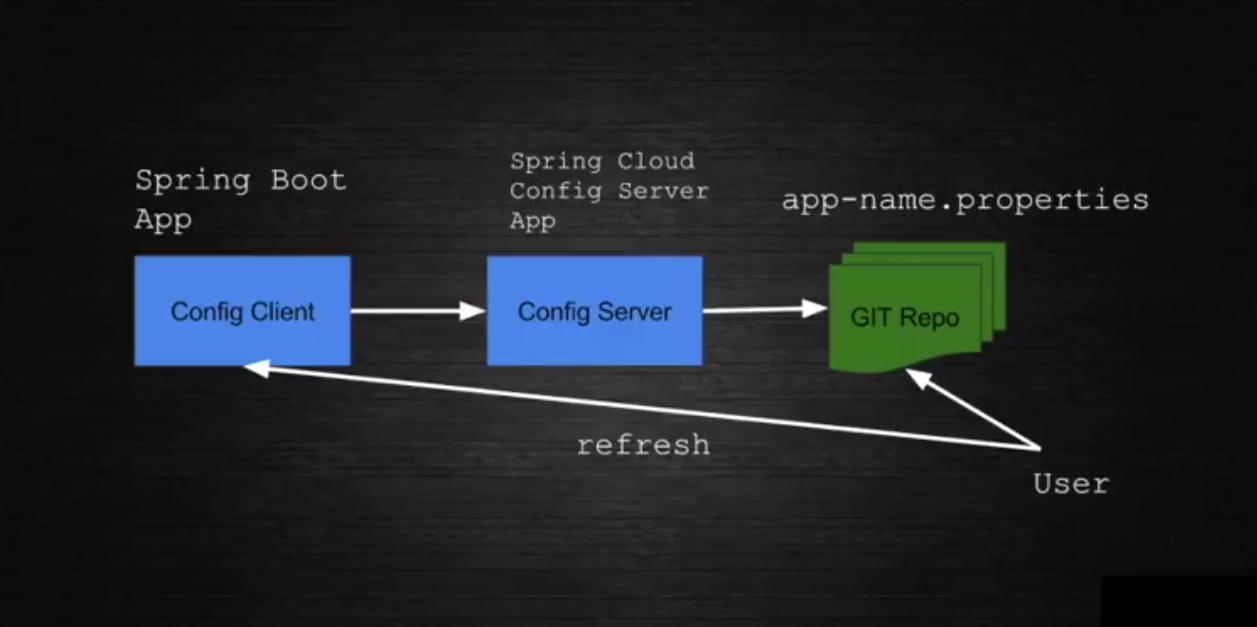
*Spring Cloud Bus* povezuje čvorove distribuiranog sustava s laganim posrednikom poruka. To se može upotrijebiti za emitiranje promjena stanja (npr. promjena konfiguracije) ili drugih upravljačkih uputa u cloud okruženju. Ključna je ideja da *Spring Cloud* Bus služi kao raspodijeljeni aktuator promjene stanja za aplikaciju *Spring Boot* koja je skalirana, ali se također može koristiti i kao komunikacijski kanal između aplikacija. Jedina implementacija trenutačno je s AMQP posrednikom kao transportnim posrednikom, ali isti osnovni skup značajki nalazi se na *class-pathu* drugih transportnih posrednika.

## Implementacija

Ovo poglavlje predstavlja upute za programere koji žele implementirati tehnologije *Spring Cloud Config* i Spring *Cloud* *Bus* u vlastito programsko rješenje u raspodijeljenom okruženju. Također, ovdje se nalaze i upute za korištenje praktičnog dijela projekta odnosno već implementiranih tehnologija *Spring Cloud Config* i *Spring Cloud* Bus na primjeru mikrousluga u raspodijeljenom okruženju. U prvom potpoglavlju nalaze se upute za *Spring Cloud Config*, a u drugom za *Spring Cloud Bus*.

### Spring Cloud Config

Kao što je već spomenuto u prethodnom poglavlju, *Spring Cloud Config* pruža podršku i za klijentsku i za poslužiteljsku stranu, pa će, u skladu s tim, i upute biti podijeljene u dva dijela. U nastavku je prikazana arhitektura programskog rješenja koje koristi *Spring Cloud Config* tehnologiju.



*Slika 4. Spring Cloud Config arhitektura*

### Poslužiteljski dio

*Spring Cloud Config* poslužitelj zapravo je *SpringBoot* aplikacija koja koristi određene ovisnosti (*eng. dependencies*) koje je potrebno uključiti na *class-path* projekta. Iako postoji više načina za inicijalizaciju *SpringBoot* aplikacije, preporučljivo se koristiti *Spring* inicijalizatorom kojem se može pristupiti na sljedećoj adresi: <https://start.spring.io/> . *Spring* inicijalizator omogućuje korisnicima da odluče koje sve ovisnosti njihov projekt treba imati te naziv projekt i potom se generira .*zip* datoteka koja se vrlo jednostavno može uvesti u neku od razvojnih okolina poput *Eclipse-a.* Ovisnost koju treba uključiti u projekt za *Spring Cloud Config* poslužitelj jest: *Config Serever*. Uključivanjem spomenute ovisnosti te dodavanjem anotacije *@EnableConfigServer* povrh *main* klase (označene sa *@SpringBootApplicaton*) programer daje dostatne upute za pokretanje te sve ostalo prepušta da *Spring* odradi za njega. Prije pokretanja *main* klase i podizanja servera nužno je napraviti još dvije stvari. Potrebno je kreirati dvije datoteke u direktoriju *java/src/resources*: *application.properties* i *bootstrap.properties*. Obje datoteke su konfiguracijskog karaktera te sadrže upute za pokretanje *Spring Cloud Config* poslužitelja. Kako smo naveli u prijašnjem poglavlju, *Spring Cloud Config* poslužitelj uvodi raspodijeljenost između implementacije pojedinih aplikacija i njihove konfiguracije, a to čini tako da komunicira sa *Git* repozitorijem u kojem se nalaze konfiguracijske datoteke od aplikacija. Iz toga razloga u *bootstrap.properties* datoteci nužno je naznačiti lokaciju *Git* repozitorija, a to se čini postavljanjem sljedećeg retka: spring.cloud.config.server.git.uri={URI do GIT repozitorija} . *Git* repozitorij može biti lokalni direktorij ili određeni *URI*. U datoteci pod nazivom *application.properties* može se postaviti vrata na kojima će *Spring Cloud Config* poslužitelj osluškivati zahtjeve postavljanjem idućeg retka: server.port={Broj vrata}. Ako se ne postavi, *Spring* će poslužitelju dodijeliti pretpostavljeni broj vrata.

### Klijentski dio

Klijentsku stranu u ovom slučaju također čine *SpringBoot* aplikacije koje svoje konfiguracijske datoteke neće dohvaćati iz lokalnog direktorija *java/src/resources* već sa *Spring Cloud Config* poslužitelja koji pohranjuje konfiguracijske datoteke u *Git* repozitoriju. Kako bi aplikacija to mogla ostvariti mora na svoj *class-path* dodati sljedeće ovisnosti: *Config Client te Actuator.* Uz spomenute ovisnosti potrebno je razrede koji promjenom konfiguracijskih datoteka mijenjanju i svoj vlastiti način izvršavanja označiti s anotacijom *@RefreshScope* koja kazuje *Springu* da je taj razred svojevrsni pretplatnik na promjene iz konfiguracijskih datoteka. Budući da se *application.properties* seli iz lokalnog direktorija u raspodijeljeni *Git* repozitorij nužno je *Springu* naglasiti gdje se nalazi *Spring Cloud Config* poslužitelj te kako će se konfiguracijska datoteka zvati u *Git* repozitoriju. To se naznačava u *bootsrap.properties* datoteci u direktoriju *java/src/resources* sljedećim retkom: spring.application.name={Ime aplikacije} te spring.cloud.config.uri={URI do poslužitelja}. Time će *Spring* kod pokretanja aplikacije znati da konfiguracijsku datoteku potraži na poslužitelju do kojeg vodi *URI* te pod nazivom *{ime aplikacije}.properties* .

### Spring Cloud Bus

U prethodnom poglavlju objašnjeno je kako funkcionira te kako se implementira raspodijeljeno pristupanje konfiguracijskim datotekama na *Spring Cloud Config* poslužitelju koji te datoteke dohvaća iz *Git* repozitorija. U ovom poglavlju bit će objašnjena tehnologija *Spring Cloud Bus* koja služi za dojavljivanje promjena u konfiguracijskim datotekama njihovim aplikacijama bez potrebe za ponovnim pokretanjem. Promjene u konfiguarcijskim datotekam na poslužiteljskoj strani dojavljuju se putem transportnog posrednika koristeći transportni protokol AMQP (eng. Advanced Message Queuing Protocol). U praktičnom dijelu korišten je *RabbitMQ* transportni posrednik čija instalacija je nužna prije pokretanja. Preporuča se preuzimanje i instalacija putem *Dockera* o čemu se više može pronaći na sljedećoj poveznici: <http://www.baeldung.com/spring-cloud-bus>. Dodatci u implementaciji su nužni i na klijentskoj i poslužiteljskoj strani, pa će, sukladno tomu i upute biti podijeljene u dva dijela.

### Poslužiteljski dio

U *Spring Cloud Config* poslužiteljsku aplikaciju potrebno je na *class-path* dodati sljedeće ovisnosti: *Monitor* (*spring-cloud-starter-monitor)* i *AMQP (spring-cloud-starter-stream-rabbit)*. *AMQP* ovisnost služi za rad s transportnim posrednikom *RabbitMQ*-em čije postavke je potrebno postaviti u datoteku *application.properties* na sljedeći način:

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

*Monitor* ovisnost služi za izlaganje */monitor* krajnje točke (*eng. endpoint*) na poslužitelju. Slanjem POST zahtjeva na spomenutu krajnju točku na Spring Cloud Config poslužitelju šalju se poruke o promjeni konfiguracije na transportni posrednik te samim time aplikacije koje su označene anotacijom *@RefreshScope* osvježavaju svoje postavke. Automatizirano slanje POST zahtjeva na */monitor* krajnju točku ostvareno je korištenjem *Git* *Webhook* usluge. To je usluga koja šalje POST zahtjeve na zadani odredišni *URI* prilikom svakog *commit* ili *push* događaja. Na taj način, kada korisnik promijeni postavke u konfiguracijskoj datoteci na *Git* repozitoriju te potom te promjene pohrani (*eng. commit*), automatski se obavještavaju aplikacije protokolom AMQP da osvježe svoje konfiguracijske postavke bez potrebe za ponovnim pokretanjem.

### Klijentski dio

U klijentskoj aplikaciji potrebno je samo uključiti ovisnost *Cloud Bus AMQP (spring-cloud-starter-bus-amqp)* te u konfiguracijsku datoteku koja se nalazi na serveru ({ime aplikacije}.properties) dodati postavke za *RabbitMQ* transportni posrednik na sljedeći način:

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

Koristeći već prije spomenutu anotaciju *@RefreshScope* aplikacija će automatski ažurirati svoj rad prema promjenama u konfiguracijskim datotekama bez potrebe za ponovnim pokretanjem.

# Logging u mikrouslugama

Važan aspekt prilikom rada s mikrouslugama jest praćenje njihovog ponašanja (engl. *monitoring*) nakon implementacije. Praćenje je vrlo važna stavka, stoga su razvijeni razni alati i tehnike za rješavanje određenih problema.

*Logging*, odnosno zapisivanje podataka, je osobito važan za otklanjanje poteškoća otkrivenih tijekom izvođenja operacija usluga. Baš kao u monolitnim aplikacijama, dnevnici zapisa (engl. *logs*) pružaju vrijedne informacije o tome što se događa tijekom rada usluge. No, za razliku od monolitnih aplikacija, ta se informacija obično distribuira između pojedinačnih dnevnika usluga i može biti nemoguća za ručnu obradu. Svaka usluga ima svoje vlastite datoteke zapisnika, a praćenje zapisivanja podataka iz jedne log datoteke u drugu jednostavno nije opcija. Tu u pomoć dolaze alati poput Splunka ili ELK stacka. Ovi alati prikupljaju dnevnike iz više izvora i pružaju snažnu funkciju pretraživanja i filtriranja.

Još jedan aspekt koji treba uzeti u obzir je kako slijediti tok izvršavanja zahtjeva dok putuju od usluge do usluge. Iako Splunk i ELK omogućuju temeljito pretraživanje, teško je povezati rezultate pretraživanja bez dodatnih informacija. Uobičajeni način da se to postigne jest dodijeliti jedinstveni identifikator svakom zahtjevu koji dolazi u sustav. Kako se zahtjev širi kroz sustav, njegov identifikator ostaje isti. Taj podatak omogućuje alatima za bilježenje povezivanje rezultata pretraživanja, dajući pregled izvještaja dnevnika povezanih s tim zahtjevom.

Srećom, razvijeni su različiti alati koji pružaju tu funkcionalnost. Spring Cloud nudi Sleuth, koji se neprimjetno integrira u Spring aplikaciju. Pomoću Sleutha lako možemo dodati mogućnosti prijave i praćenja našim mikrouslugama.[1]

Također, tu je i Zipkin, distribuirani sustav praćenja koji pomaže pri prikupljanju vremenskih podataka za svaki zahtjev razmnožen između neovisnih usluga. Ima jednostavnu upravljačku konzolu gdje možemo pronaći vizualizaciju vremenskih statistika generiranih naknadnim uslugama.

## Sleuth

Jedan od problema s kojima se programeri susreću kada se njihova mikrousluga širi, odnosno raste, jest praćenje zahtjeva koji se šire od mikrousluge do mikrousluge. Pokušavanje razumijevanja kako zahtjev putuje kroz aplikaciju može biti prilično komplicirano, pogotovo ako se nema uvid u implementaciju same usluge koja se poziva. Spring Cloud Sleuth je alat u sklopu Spring Clouda koji je predviđen za rješavanje tog problema. On uvodi jedinstvene identifikatore (ID) u zapisivanje podataka koji su konzistentni među mikrouslugama i njihovim pozivima, stoga je moguće otkriti kako zahtjev putuje od jedne do druge mikrousluge. Spring Cloud Sleuth dodaje dvije vrste ID-ova, jedan koji se naziva *trace* ID i drugi koji se naziva *span* ID. *Span* ID predstavlja osnovnu jedinicu posla, primjerice slanje HTTP zahtjeva. *Trace* ID sadrži skup *span* ID-ova, koji tvore strukturu sličnu stablu. *Trace* ID ostaje isti prilikom poziva druge mikrousluge od strane prethodne. U nastavku slijedi jednostavni primjer korištenja Spring Cloud Sleutha za praćenje zahtjeva.

Potrebno je otići na <http://start.spring.io/> i izraditi novu Spring Boot aplikaciju koja ovisi o Sleuthu (spring-cloud-starter-sleuth) te generirati projekt za preuzimanje koda. Dobro je dati aplikaciji ime i također potrebno za smisleno praćenje od strane Sleutha. Potrebno je stvoriti datoteku pod nazivom bootstrap.yml u src/main/resources. U toj datoteci treba dodati svojstvo (engl. *property*) spring.application.name i postaviti ga onako kako se želi nazvati aplikaciju. Ime dano aplikaciji pojavit će se kao dio praćenja koje radi Sleuth.

Sada se mogu dodati neki zapisnici u aplikaciju da bi se vidjelo kako će izgledati trag (engl. *trace*). Potrebno je otvoriti datoteku aplikacije (engl. *application file*) za aplikaciju (gdje se nalazi glavna metoda) i stvoriti metodu nazvanu home koja vraća podatak tipa String.

public String home() {

return "Hello World";

}

Neka se ta metoda poziva pritiskom korijena (engl. *root*) web-aplikacije. Potrebno je dodati oznaku @RestController na razrednoj razini, a zatim @RequestMapping ("/") na metodu home.

@SpringBootApplication

@RestController

public class SleuthSampleApplication {

public static void main(String[] args) {

SpringApplication.run(SleuthSampleApplication.class, args);

}

@RequestMapping("/")

public String home() {

LOG.log(Level.INFO, "you called home");

return "Hello World";

}

}

Ako se sada pokrene aplikacija i otvori <http://localhost:8080> trebalo bi biti ispisano Hello World. Do ove točke sve što imamo je osnovna Spring Boot aplikacija. Potrebno je dodati zapisivanje u aplikaciju da bi se vidjeli podaci o praćenju od Sleutha. Dodaje se sljedeća varijabla u razred aplikacije.

private static final Logger LOG = Logger.getLogger(SleuthSampleApplication.class.getName());

Potrebno je postaviti ispravan naziv klase. U home metodi treba dodati sljedeći log dio:

@RequestMapping("/")

public String home() {

LOG.log(Level.INFO, "you called home");

return "Hello World";

}

Ako se sada pokrene aplikacija i otvori <http://localhost:8080> trebao bi se vidjeti sljedeće u konzoli:

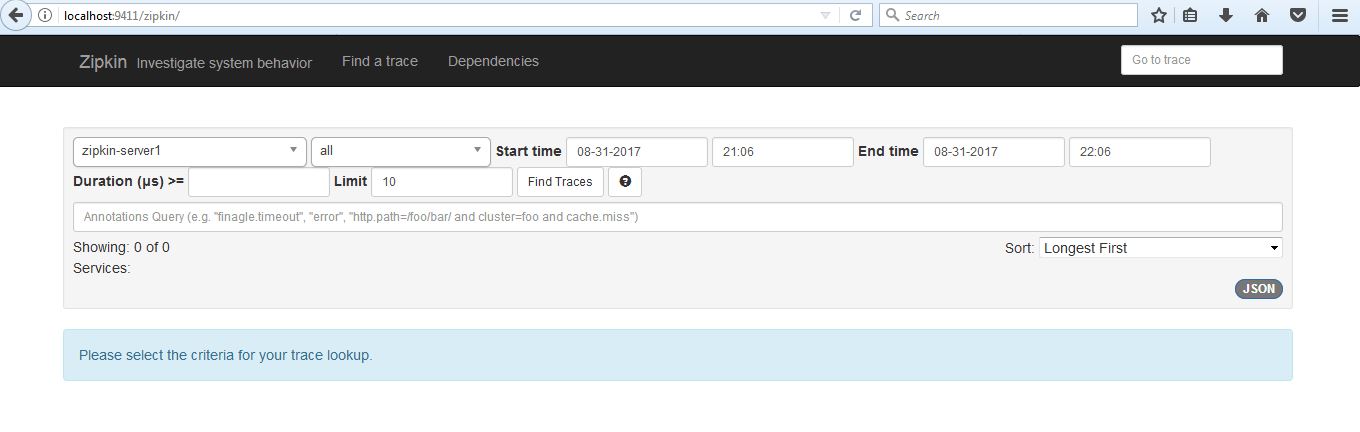
2016-06-15 16:55:56.334 INFO **[**slueth-sample,44462edc42f2ae73,44462edc42f2ae73,false] 13978 --- **[**nio-8080-exec-1] com.example.SleuthSampleApplication : calling home

Dio koji Sleuth dodaje je **[**slueth-sample, 44462edc42f2ae73, 44462edc42f2ae73, false]. Prvi parametar je naziv aplikacije (spring.application.name u bootstrap.yml). Drugi parametar je *trace* ID. Treći je *span* ID. Konačno, zadnji parametar označava treba li se *span* izvesti u Zipkin.

Osim dodavanja dodatnih podataka o praćenju u zapisivanje ( izjava o zapisima), Spring Cloud Sleuth također pruža neke važne prednosti prilikom pozivanja drugih mikrousluga. Stvarni problem ovdje nije identificiranje dnevnika zapisa unutar jedne mikrousluge, već praćenje niza zahtjeva na više mikrousluga. Mikrousluge obično međusobno komuniciraju sinkronizirano pomoću REST API-ja i asinkrono putem spremišta poruka (engl. *message hub*). Sleuth može pružiti informacije o praćenju u oba slučaja. [2]

## Zipkin

Zipkin je distribuirani sustav praćenja koji pomaže u prikupljanju podataka o vremenu potrebnim za rješavanje problema s latencijom u arhitekturi mikrousluga. Upravlja i prikupljanjem i traženjem ovih podataka. Zipkinov dizajn temelji se na Dapperu tvrtke Google.



*Slika 5. Zipkin web UI*

Aplikacije izvještavaju Zipkin o vremenskim podacima te ih on obrađuje. Zipkin korisničko sučelje također predstavlja dijagram ovisnosti koji pokazuje koliko je praćenih zahtjeva prošlo kroz svaku aplikaciju. Ako se pojave problemi s latencijom ili pogreške, moguće je filtrirati ili poredati sve tragove (engl. *traces*) na temelju aplikacije, duljine tragova, bilješke ili vremenske oznake. Nakon što se odabere trag, može se vidjeti postotak ukupnog vremena praćenja za svaki raspon koji omogućuje prepoznavanje problema.

Zipkin se sastoji od četiri dijela:

* Kolektor (engl. *Collector*) - kada komponenta pošalje podatke o tragovima, oni dolaze u Zipkin kolektor daemon, provjeravaju se, pohranjuju i indeksiraju za traženje od strane Zipkinovog kolektora;
* Spremište (engl. *Storage*) – pohranjuje i indeksira podatke pretraživanja u pozadini;
* Pretraživanje (engl. *Search*) - pruža jednostavan JSON API za pronalaženje i dohvaćanje tragova pohranjenih u pozadini. Primarni korisnik ovog API-ja je Web UI;
* Web UI – korisničko sučelje za prikaz tragova.

Za jednostavno pokretanje Zipkin servera na Windows računalima dovoljno je skinuti JAR s [maven repozitorija](https://search.maven.org/remote_content?g=io.zipkin.java&a=zipkin-server&v=LATEST&c=exec) te pokrenuti ga u Command Promptu.

Za pokretanje Zipkin servera kao Spring Boot aplikaciju potrebno je dodati sljedeću Zipkin *starter* ovisnost (engl. *dependency*) u Spring Boot projekt te anotaciju @EnableZipkinServer iznad klase. [3]

<dependency>

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

    <artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

Kako bi Sleuth slao podatke na Zipkin potrebno je u Spring Boot aplikaciju dodati sljedeću ovisnost u pom.xml :

<dependency>

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

    <artifactId>spring-cloud-zipkin-sleuth</artifactId>

</dependency>

Također, potrebno je odrediti koliko često želimo slati podatke Sleuthom na Zipkin. To se može napraviti na dva načina.

Prvi način je da u aplikaciju dodamo sljedeći dio koda:

@Bean

public AlwaysSampler defaultSampler() {

return new AlwaysSampler();

}

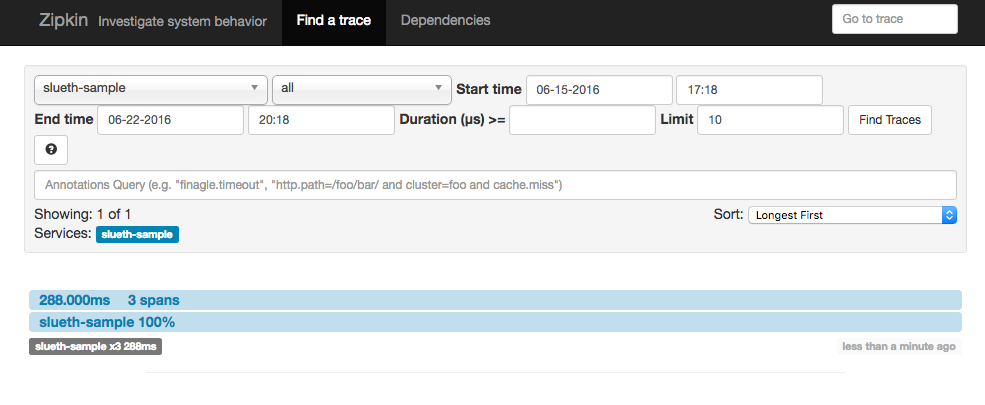
Time kažemo da želimo da se šalju svi podaci.

Drugi način za slanje zapisa je dodavanjem spring.sleuth.sampler.percentage u application.properties. Time određujemo postotak podataka koji će biti poslani na Zipkin. Zadana vrijednost je 0.1, odnosno 10%.

Sada ako se ponovno pokrene aplikacija i otvori <http://localhost:8080> trebalo bi se vidjeti sljedeće u konzoli:

2016-06-15 16:55:56.334 INFO **[**slueth-sample,44462edc42f2ae73,44462edc42f2ae73,false] 13978 --- **[**nio-8080-exec-1] com.example.SleuthSampleApplication : calling home

Sada je zadnji parametar true što znači da je podatak, odnosno *span* poslan na Zipkin. Ako se otvori [http://localhost:9411](http://localhost:9411/) trebao bi se vidjeti Zipkin web UI u kojem se mogu prikazati rezultati odabirom željene usluge, ispravnim vremenskim intervalom te pritiskom na tipku *Find Traces*. Zapisnici su prikazani u vremenskoj ovisnosti te ih je moguće pretraživati preko *traceID*-a.[2]



*Slika 6. Zipkin web UI nakon ispunjavanja željenih polja te pritiska tipe Find Traces*

## ELK stack



*Slika 7. Dijelovi ELK stacka*

ELK stack je zbirka od tri open-source proizvoda od Elastica: Elasticsearch, Logstash i Kibana. Elasticsearch je NoSQL baza podataka koja se temelji na tražilici Lucene. Logstash je alat koji prihvaća ulaze iz različitih izvora, izvršava različite transformacije i izvozi podatke na različita mjesta . Kibana je sloj vizualizacije koji radi na vrhu Elasticsearch.

Zajedno, ova tri različita proizvoda otvorenog koda najčešće se koriste u analizi dnevnika u IT okruženjima (iako postoje mnogi slučajevi korištenja za pokretanje ELK stacka, uključujući poslovnu inteligenciju, sigurnost i sukladnost te web analitiku). Logstash prikuplja i analizira dnevnike, a zatim Elasticsearch indeksira i pohranjuje podatke. Kibana zatim prikazuje podatke u vizualizacijama koje pružaju djelotvorne uvide u nečije okruženje.

Elasticsearch često se opisuje kao poslužitelj za pretraživanje i NoSQL baza podataka. To znači da pohranjuje podatke na nestrukturiran način i da ne možete koristiti SQL da biste ga upitali. Za razliku od većine NoSQL baza podataka, Elasticsearch se snažno usredotočuje na mogućnosti pretraživanja i značajke - toliko, zapravo, da je najlakši način za dobivanje podataka od Elasticsearcha traženjem pomoću REST API-ja. Za dodavanje podataka se koriste PUT i POST metode, dok se za brisanje podataka koristi DELETE. Za dohvaćanje podataka se koristi metoda GET.

Svrha ELK stacka je pohranjivanje, vizualizacija i analiza zapisa i drugih podataka u vremenskoj seriji. Logstash je sastavni dio tijeka podataka od izvora do Elasticsearcha i dalje. Ne samo da dopušta povlačenje podataka iz raznih izvora, već pruža alate za filtriranje i oblikovanje podataka kako bi se lakše radilo s njima. Logstash ima jednostavnu DSL konfiguraciju koja omogućuje određivanje ulaza (engl. *inputs*), izlaza (engl. *outputs*) i filtara zajedno s njihovim specifičnim opcijama. Poredak je bitan, pogotovo kod filtara i izlaza, jer se konfiguracija u osnovi pretvara u kôd i izvršava. Konfiguracije obično imaju tri dijela: ulazi, izlazi i filtri. Može se imati više primjeraka svakog od tih dijelova, što znači da se može grupirati povezane dodatke u konfiguracijskoj datoteci umjesto da ih se grupira prema vrsti. Pomoću ulaza moguće je uvesti podatke od različitih izvora i raditi s njima šta god želimo. Izlazi služe za dostavljanje događaja na različite lokacije i usluge. Također tu je i nekoliko vrlo moćnih filtera s kojima se može manipulirati, izmjeriti i stvarati događaje. Snaga tih filtera čini Logstash vrlo svestranim i vrijednim alatom.

Kibana je open-source alat za vizualizaciju i istraživanje podataka koji se koristi za analize dnevnika i vremenskih nizova te praćenje aplikacija. Kibana nudi usku integraciju s Elasticsearchom, što ju čini zadanim izborom za vizualizaciju podataka pohranjenih u Elasticsearchu. Kibana je također popularna zbog svojih moćnih i jednostavnih značajki kao što su histogrami, grafikoni linija, kružni grafikoni, toplinske mape i ugrađena geoprostorna podrška. Kibana nudi intuitivne grafikone i izvješća koja se mogu koristiti za interaktivno kretanje ogromnim količinama podataka zapisnika. Može se dinamično povlačiti vremenske prozore, zumirati na određene podskupove podataka, smanjiti prikaz kako bi se vidjela veća slika i pregledala izvješća za analizu podataka.[4]

Kako bi se sve uspješno pokrenulo potrebno je ići određenim redoslijedom.

Najprije treba skinuti Elasticsearch arhivu. Nakon toga se može direktno pokrenuti instanca.

Primjer naredbi za to:

curl -O <https://download.elasticsearch.org/elasticsearch/elasticsearch/elasticsearch-1.7.1.tar.gz>

tar zxvf elasticsearch-1.7.1.tar.gz

./elasticsearch-1.7.1/bin/elasticsearch

Elasticsearch je sada pokrenut. To je moguće provjeriti slanjem GET zahtjeva na adresu <http://localhost:9200>.

Nakon toga treba skinuti Kibanu.

curl -O <https://download.elasticsearch.org/kibana/kibana/kibana-4.1.1-darwin-x64.tar.gz>

tar zxvf kibana-4.1.1-darwin-x64.tar.gz

./kibana-4.1.1-darwin-x64/bin/kibana

Sada je pokrenuta i Kibana što se može provjeriti na <http://localhost:5601>.

Nakon toga slijedi skidanje Logstasha.[6]

curl -O <https://download.elasticsearch.org/logstash/logstash/logstash-1.5.3.tar.gz>

tar zxvf logstash-1.5.3.tar.gz

Kako bi Logstash mogao sakupiti zapise potrebno ih je spremiti u *log* datoteku, a za to je potrebno u application.properties Spring Boot aplikacije dodati logging.file = application.log. Sada će se zapisi spremati u datoteku application.log.

Potrebno je konfigurirati Logstash, a za to treba stvoriti datotetku logastash.conf koja će se sastojati od tri dijela kao što je već objašnjeno ranije. To ulaz, filter i izlaz koji se mogu sastojati od više dijelova.

Primjer ulaza:

input {

file {

type => "java"

path => "/path/to/application.log"

codec => multiline {

pattern => "^%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME}.\*"

negate => "true"

what => "previous"

}

}

}

Koristi se *file* plugin i tip je namješten na java. Zatim se navodi apsolutni put do *log* datoteke. Koristi se *multiline codec* jer više linija može odgovarati jednom događaju. Ako zapis ne počinje oblikom *pattern* onda on pripada prošlom zapisu.

Primjer filtera:

filter {

#If log line contains tab character followed by 'at' then we will tag that entry as stacktrace

if [message] =~ "\tat" {

grok {

match => ["message", "^(\tat)"]

add\_tag => ["stacktrace"]

}

}

#Grokking Spring Boot's default log format

grok {

match => [ "message",

"(?<timestamp>%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME}) %{LOGLEVEL:level} %{NUMBER:pid} --- \[(?<thread>[A-Za-z0-9-]+)\][A-Za-z0-9.]\*\.(?<class>[A-Za-z0-9#\_]+)\s\*:\s+(?<logmessage>.\*)",

"message",

"(?<timestamp>%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME}) %{LOGLEVEL:level} %{NUMBER:pid} --- .+? :\s+(?<logmessage>.\*)"

]

}

#Parsing out timestamps which are in timestamp field thanks to previous grok section

date {

match => [ "timestamp" , "yyyy-MM-dd HH:mm:ss.SSS" ]

}

}

U ovom filteru prvo se označava zapis ako sadrži *stacktrace*. Zatim se zapis parsira u zadani format te specificira vremenska oznaka i format kako bi ih Kibana mogla koristiti za traženje po vremenu.

Primjer izlaza:

output {

stdout {

codec => rubydebug

}

elasticsearch {

host => "localhost"

}

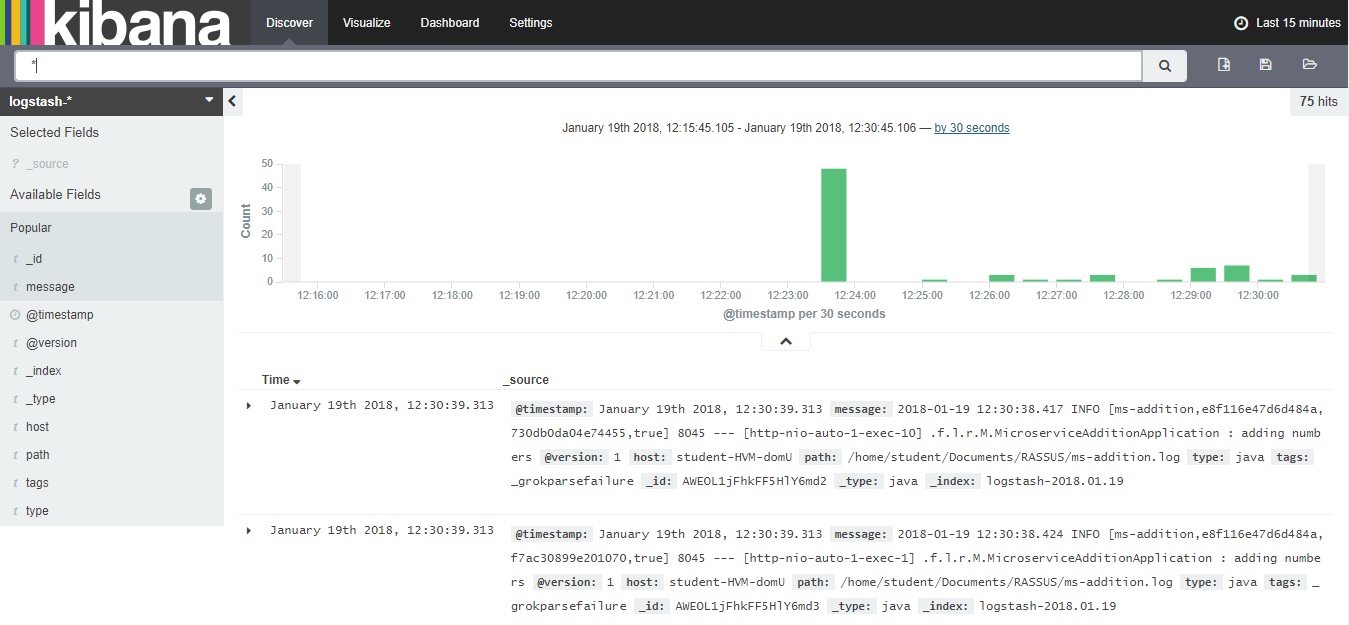
}

U ovom slučaju se koriste dva izlaza. Jedan je za standardni ispis u konzoli, a drugi za slanje zapisa na Elasticsearch server.

Nakon što je Logstash konfiguriran, možemo pokrenuti Logstash Agenta naredbom:

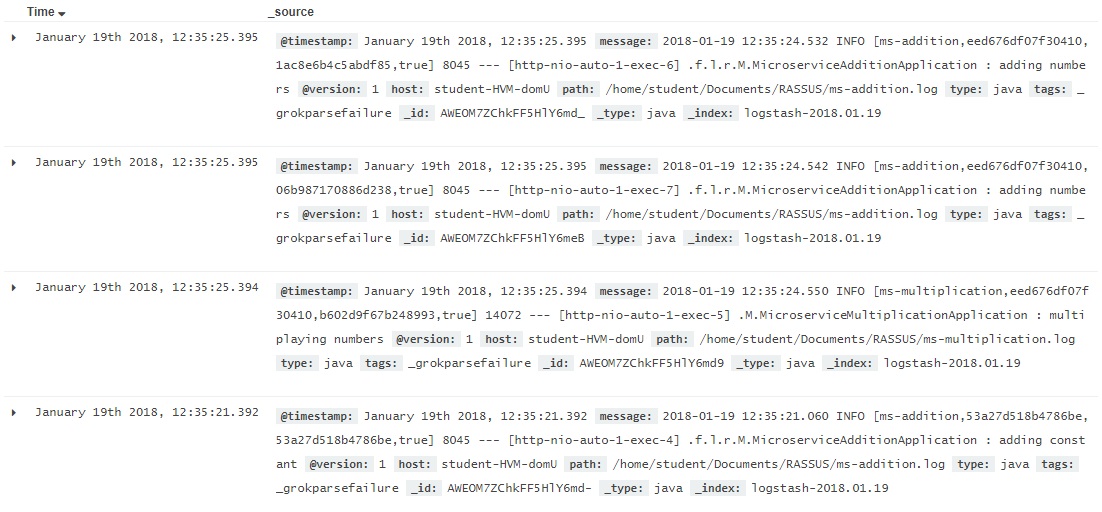
./logstash-1.5.3/bin/logstash agent -v -f /path/to/logstash.conf

Logstash bi sada trebao slati zapise na Elasticsearch. Nakon toga treba posjetiti Kibana web UI koji smo ranije pokrenuli. Prvo treba odrediti Elasticsearch indeks koji Kibana treba pratiti. Logstash stvara indekse oblika logstash-YYYY.MM.DD. U Postavkama (engl. *Settings*) Kibane potrebno je označiti obje ponuđene opcije, odabrati dnevni interval te upisati oblik. Indeks se stvara pritiskom na *Create Index.*  Zatim klik na *Discover* . Tu se vide zapisi i pružaju razne mogućnosti za rad s tim zapisima kao što se može vidjeti na slici ispod.[5]



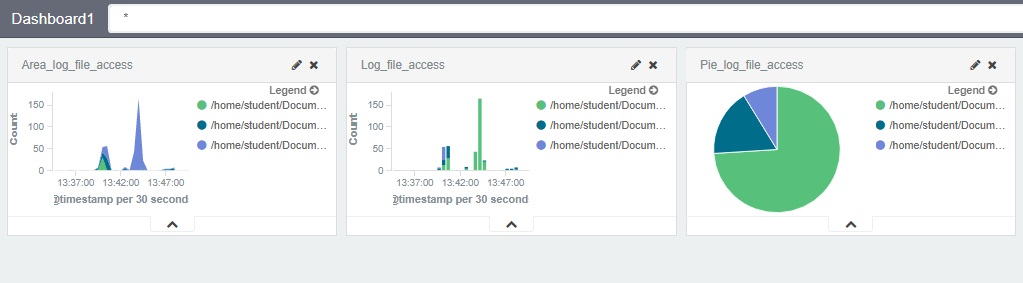
*Slika 8. Kibana web UI*

U *Discoveru* se ispod grafički prikazanog dolaska zapisa u određenom vremenskom intervalu prikazanog na slici iznad nalaze ti zapisi filtrirani i parsirani na željeni način. Svaki zapis ima oznake *@timestamp*, *message*, *@version*, *host*, *path*, *type*, *tags*, *\_id*, *\_type* i *\_index* kao što se može vidjeti na slici.



*Slika 9. Primjer zapisa na Kibani*

Uz *Discover* tu su još i *Visualize* pomoću kojega se može vizualno prikazati sakupljene podatke u raznim oblicima te *Dashboard* u koji se mogu sakupiti vizualni prikazi. Primjer *Dashboarda* sa tri različita načina prikaza pristupa log datotekama u vremenu prikazan je u nastavku.



*Slika 10. Primjer Dashboarda s grafovima*

## Upute za pokretanje

1. Pokrenuti Zipkin preko jar datoteke
2. Pokrenuti mikrousluge
3. Pokrenuti Elasticsearch pomoću naredbe:

./elasticsearch-1.7.1/bin/elasticsearch

1. Pokrenuti Logstash pomoću naredbe:

./logstash-1.5.3/bin/logstash agent -v -f /path/to/logstash.conf

1. Pokrenuti Kibanu pomoću naredbe:

./kibana-4.1.1-darwin-x64/bin/kibana

# Registracija i pronalazak usluga, rasterećenje sustava i oporavak od ispada usluga

Raspodijeljeni sustavi često sadrže velik broj grupa poslužitelja različitih funkcionalnosti, a svaka se grupa sastoji od niza poslužitelja koji obavljaju isti posao. Dupliciranje poslužitelja povećava ukupnu kvalitetu pružanja usluge jer povećava robusnost, približava uslugu velikom broju korisnika diljem svijeta što smanjuje vrijeme odziva i raspoređuje posao na više poslužitelja s istom namjenom. Bez sofisticiranih alata nemoguće je pratiti i pravovremeno reagirati na svaku promjenu na ovako velikom broju poslužitelja. Jedna od usluga gledanja video sadržaja koja je dostupna širom svijeta zasnovana je na vrlo složenom raspodijeljenom sustavu naziva se *Netflix.* Neki od alata za unaprjeđenje i stalni nadzor pružanja usluga *Netflixa* su *Eureka, Hystrix, Ribbon*  i *Feign.* U nastavku slijedi kratka analiza svakog od ovih alata i njihova uloga u ovom, mnogima nezamislivo velikom sustavu.

## Eureka

*Eureka* je usluga zasnovana na REST-u (*Representational State Transfer*) i primarna joj je uloga otkrivanje usluga u AWS (*Amazon Web Services*) oblaku u svrhu raspoređivanja opterećenja (*load balancing*) i prikrivanja nedostupnosti poslužitelja koji sudjeluju u pružanju usluge krajnjem korisniku. Ova se usluga naziva *Eureka* poslužitelj. Uz poslužitelj dolazi i *Eureka* klijent koji olakšava interakcije s poslužiteljem. Klijent sadrži i ugrađeni jednostavni raspoređivač opterećenja koji koristi *round-robin* raspoređivanje. U nastavku slijedi postupak korištenja *Eureka* poslužitelja i klijenta.

### Eureka poslužitelj

Prvo je potrebno implementirati i konfigurirati *Eureka* poslužitelja. Razred poslužitelja potrebno je označiti anotacijama *@SpringBootApplication* jer se radi o aplikaciji u *Spring Bootu,* te *@EnableEurekaServer*  za omogućavanje *Eureka* poslužitelja. Primjer jednostavnog Eureka poslužitelja:

@SpringBootApplication

@EnableEurekaServer

public class EurekaServerApplication **{**

public static void main**(**String**[]** args**)** **{**

SpringApplication**.**run**(**EurekaServerApplication**.**class**,** args**);**

**}**

**}**

Konfiguracijska datoteka *Eureka* poslužitelja je u *.properties* (može biti i *.yml*) formatu, a zove se *eureka-server.properties*. Primjer konfiguracijske datoteke:

server.port=8761

eureka.client.registerWithEureka=false

eureka.client.fetchRegistry=false

Vrata su početno postavljena na vrijednost 8761. Zbog toga što se ova aplikacija koristi isključivo kao poslužitelj, ugrađeni klijent konfigurira se tako da se ne registrira na poslužitelj te ne dohvaća registrirane *Eureka* klijente.

### Eureka klijent

Na postavljeni poslužitelj sada valja registrirati dostupne aplikacije koje predstavljaju *Eureka* klijente. Anotacija specifična za *Eureku* koja se koristi u označavanju razreda aplikacije jest *@EnableEurekaClient* koja *Spring Bootu* govori da za otkrivanje usluga koristi *Eureku*. *Eureka* klijent na primjeru mikrousluge zbrajanja:

@SpringBootApplication

@EnableEurekaClient

@RestController

public class MicroserviceAdditionApplication **{**

@Value**(**"${constantToAdd: 500}"**)** //vrijednost iz konfiguracijske datoteke

private static double c**;** //500 je vrijednost ako nije postavljena

public static void main**(**String**[]** args**)** **{**

SpringApplication**.**run**(**MicroserviceAdditionApplication**.**class**,** args**);**

**}**

//metoda koja vraća zbroj dvaju brojeva predanih u URL-u

@RequestMapping**(**value **=** "/add"**,** method **=** RequestMethod**.**GET **)**

public double add**(**@RequestParam**(**"a"**)** double a**,** @RequestParam**(**"b"**)** double b**)** **{**

**return** a**+**b**;**

**}**

//metoda koja vraća zbroj broja predanog u URL-u i konstante iz konfig. dat.

@RequestMapping**(**value **=** "/addConstant"**,** method **=** RequestMethod**.**GET **)**

public double add**(**@RequestParam**(**"a"**)** double a**)** **{**

**return** a**+**c**;**

**}**

**}**

Konfiguracijska datoteka slična je poslužiteljskoj, uz poneke dodatke kao što je ime same usluge po kojoj ju drugi poslužitelji i aplikacije mogu zatražiti od *Eureka* poslužitelja. Primjer:

U ovoj datoteci definiran je *Eureka* poslužitelj na koji se klijent prijavljuje te ime instance s kojim se prijavljue, konstanta koja se koristi pri pozivu */addConstant*, te parametri vezani za osvježavanje stanja na *Eureka* poslužitelju.

Nakon pokretanja *Eureka* klijenta, na lokaciji [*http://localhost:8761*](http://localhost:8761)vidljiv je status njegove registracije na  *Eurekinoj* kontrolnoj ploči.

Ovako registriranu aplikaciju preko njenog REST sučelja koriste drugi klijenti nakon dobivanja njene lokacije upitom na *Eureka* poslužitelj.

spring.application.name='ms-addition'

server.port=0

eureka.client.serviceUrl.defaultZone=${EUREKA\_URI:http://localhost:8761/eureka}

eureka.instance.leaseRenewalIntervalInSeconds=1

eureka.instance.leaseExpirationDurationInSeconds=1

eureka.instance.metadataMap.instanceId=${spring.application.name}:${spring.application.instance\_id:${random.value}}

eureka.instance.instanceId=${spring.application.name}:${spring.application.instance\_id:${random.value}}

constantToAdd=500

*Eureka* se koristi u okruženju u kojemu servisi često mijenjaju svoje lokacije i iz različitih razloga često mogu biti nedostupni. [7][8]

## Hystrix

*Hystrix* je radni okvir koji omogućuje da raspodijeljeni sustavi koji se sastoje od većeg broja usluga koje međusobno surađuju budu tolerantni na ispade pojedinih usluga i na pojavu smetnji u komunikaciji između usluga. Njime se izoliraju usluge koje ne funkcioniraju i tako se sprječava propagacija grešaka u sustavu. Funkcionalnost *Hystrixa* postiže se omatanjem poziva prema drugim uslugama u razrede koji nasljeđuju *HystrixCommand<String>* i predstavljaju naredbe. U nastavku slijede različiti načini otkrivanja i izolacije od pogrešaka.

Jedan od načina jest korištenje sata određivanjem vremenskog intervala u kojemu odgovor pozvane usluge mora doći natrag kako bi ju se smatralo aktivnom. U slučaju da odgovor ne stigne u vremenu zadanom u konfiguraciji naredbe, baca se iznimka *HystrixRuntimeException*, i to nakon isteka zadanog vremena, a ne nakon primitka zakašnjelog odgovora.

Drugi je način postavljanje ograničenja broja dretvi kojima jedna usluga komunicira s ostalima. Ovo ograničenje omogućuje odbacivanje zahtjeva prema usluzi kada njihov broj nadmaši određenu vrijednost, što ju štiti od preopterećenja.

U slučaju kada je neka usluga postala potpuno nefunkcionalna cilj je na određeni vremenski period u potpunosti prekinuti slanje upita prema njoj kako bi se mogla oporaviti od greške i kako bi se pri tom potrošila minimalna količina resursa. Obrazac koji se u tom slučaju koristi jest *Short Circuit Breaker* kojemu je cilj na zadano vrijeme prekinuti komunikaciju sa zadanom uslugom. Konfiguraciju je moguće postaviti tako da se zada broj neuspješnih zahtjeva nakon kojih se usluga smatra nedostupnom i vrijeme nakon kojega će se prema njoj ponovno početi slati upiti.

Ovim metodama omogućuje se rano otkrivanje pogrešaka u uslugama raspodijeljenog sustava i pravovremenog pokretanja postupka oporavka, nadzor i obavještavanje o stanju usluga, minimizacija degradacije korisničkog iskustva i slično.[9][10]

Primjer korištenja *Hystrixa* bit će prikazan u poglavlju u kombinaciji s *Feignom* i *Ribbonom*.

## Ribbon

*Ribbon* je biblioteka korištena u oblaku za internu komunikaciju među procesima. Njegova je primarna svrha pružanje algoritama za raspodjelu opterećenja na klijentskoj strani. *Ribbon* sadrži i integraciju s *Eurekom* pa ju zbog toga nije potrebno odvojeno koristiti. On sadrži i funkcionalnost za otkrivanje grešaka i poslužitelja koji su nedostupni. Neki od algoritama koje *Ribbon* podržava su *RoundRobinRule,*  *AvailabilityFilteringRule* i *WeightedResponseTimeRule,* a moguće je i definirati nova proizvoljna pravila.

Na *Ribbonovom* raspoređivaču opterećenja moguće je postaviti parametre *Rule, Ping* i *ServerList*. Parametar *Rule* označava algoritam koji će se koristiti pri raspoređivanju upita. Primjer algoritma je već spomenuti *WeightResponseTimeRule* koji se temelji na vremenu odziva pojedinog poslužitelja. Poslužitelj s kraćim vremenom odziva ima manju težinu pa se na njega upiti češće šalju. Ova se težina određuje dinamički načinom određenim u parametru *Ping.* Ovaj parametar određuje način na koji će biti određeno vrijeme odgovora poslužitelja. Postavljanjem ovog parametra na *PingUrl* znači da će se to određivati slanjem *ping*  poruka na njegov URL. *ServerList* je lista poslužitelja koja predstavlja poslužitelje kojima pojedina vrsta zahtjeva može biti poslana, a može biti određena statički ili dinamički. U ovom primjeru, prema svim poslužiteljima u ovoj listi šalje se *ping* poruka za određivanje brzine primanja odgovora, svakome se pojedinačno dodjeljuje težina ovisno o brzini primitka odgovora te se prema tomu raspoređuje opterećenje.

*Ribbon* API-jem moguće je odrediti i dostupnost poslužitelja konstantnim slanjem zahtjeva u određenim vremenskim intervalima i tako izbjeći slanje zahtjeva prema nedostupnim poslužiteljima. U slučaju da su svi poslužitelji koji mogu obraditi neki zahtjev nedostupni, *pingUrl()* vratit će iznimku *java.lang.IllegalStateException* s porukom „No instances are available to server the request.“. *Ribbon* sadrži i implementaciju obrasca *Circuit Breaker*, kao i *Hystrix*. [11][12]

Primjer korištenja *Ribbona* bit će prikazan u sljedećem poglavlju u kombinaciji s *Feignom* i *Hystrixom*.

## Feign

Feign je HTTP klijent kojemu je cilj pojednostavljenje komunikacije s REST API-jem. Programer treba definirati samo sučelje prema REST poslužitelju korištenjem anotacija *@RequestLine, @Param, @Headers* i sl. Anotacija *@RequestLine* označava HTTP metodu i put do resursa koji može sadržavati parametre označene anotacijom *@Param*. Pomoću *@Headers* moguće je definirati format zahtjeva, odgovora, pristupne podatke i sl. [13][14]

### Primjer Feigna s Ribbonom i Hystrixom

U ovom projektu, *Feign* je korišten kao usluga za računanje udaljenosti između dviju točaka. Ta usluga za potrebe izračuna koristi mikrousluge zbrajanja i množenja. Ovakvim se rasporedom usluga olakšava demonstracija funkcioniranja *Feigna* s *Ribbonom* i *Hystrixom.*

Kao što je spomenuto ranije u ovom poglavlju, potrebno je definirati sučelje prema poslužitelju kojeg *Feign* klijent koristi. U ovom slučaju poslužitelj je mikrousluga zbrajanja, a kôd slijedi u nastavku.

@FeignClient**(**value **=** "ms-addition"**)**

public interface AdditionClient **{**

@RequestMapping**(**"/add"**)**

double add**(**@RequestParam**(**"a"**)** double a**,** @RequestParam**(**"b"**)** double b**);**

**}**

Metodu definiranu u sučelju koristi aplikacija *Feign* klijenta:

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients

@Controller

public class FeignDistanceApplication **{**

@Autowired

private AdditionClient additionClient**;**

@Autowired

private MultiplicationClient multiplicationClient**;**

public static void main**(**String**[]** args**)** **{**

SpringApplication**.**run**(**FeignDistanceApplication**.**class**,** args**);**

**}**

//d= sqrt( (bx-ax)^2 + (by-ay)^2 )

@RequestMapping**(**path **=** "/distance"**,** produces **=** "text/plain"**)**

@ResponseBody

public String multi**(**@RequestParam**(**"ax"**)** double ax**,** @RequestParam**(**"ay"**)** double ay**,** @RequestParam**(**"bx"**)** double bx**,** @RequestParam**(**"by"**)** double by**)** **{**

double xDiff **=** additionClient**.**add**(**bx**,** **-**ax**);** //pozivi usluge zbrajanja

double yDiff **=** additionClient**.**add**(**by**,** **-**ay**);**

double xSquare **=** multiplicationClient**.**multiply**(**xDiff**,** xDiff**);** //mnozenje

double ySquare **=** multiplicationClient**.**multiply**(**yDiff**,** yDiff**);**

double sum **=** additionClient**.**add**(**xSquare**,** ySquare**);**

**return** String**.**valueOf**(**Math**.**sqrt**(**sum**));** **}}**

U slučaju ispada neke od mikrousluga, primjerice one zadužene za množenje, potrebno je spriječiti propagaciju te pogreške. To se ostvaruje dodavanjem *fallback* metode u sučelje REST poslužitelja koja se izvršava ako niti jedna instanca mikrousluge zadužene za taj posao nije dostupna. Implementacija *fallback* metode nalazi se u razredu koji implementira sučelje prema REST poslužitelju, a sama definicija koja ju označava *fallback* metodom nalazi se u anotaciji *@FeignClient* sučelja. Primjeri su prikazani u nastavku:

@Component

public class MultiplicationClientFallback **implements** MultiplicationClient**{**

public double multiply**(**double a**,** double b**)** **{**

**return** **-**1**;**

**}**

**}**

@FeignClient**(**

value **=** "ms-multiplication"**,**

fallback **=** MultiplicationClientFallback**.**class

**)**

public interface MultiplicationClient **{**

@RequestMapping**(**"/multi"**)**

double multiply**(**@RequestParam**(**"a"**)** double a**,** @RequestParam**(**"b"**)** double b**);**

**}**

Dio konfiguracijske datoteke vezan sa *Feignovo* komuniciranje s *Eurekom* ne razlikuje se od konfiguracije ranije prikazanog *Eureka* klijenta. Da bi *fallback* funkcionirao kako treba, u konfiguracijsku datoteku valja uključiti *Hystrix*:

feign.hystrix.enabled=true

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=1000

Nasuprot prethodnom scenariju kada nema dostupnih mikorusluga traženog tipa, moguće ih je na raspolaganju imati nekoliko. Za raspoređivanje opterećenja s klijentske strane koristi se *Ribbon* za čije je funkcioniranje u konfiguracijsku datoteku potrebno dodati sljedeće:

|  |
| --- |
| ribbon.eureka.enabled=true |
|  |

|  |
| --- |
| ribbon.MaxAutoRetries=0 |
|  |

|  |
| --- |
| ribbon.MaxAutoRetriesNextServer=0 |
|  |

ribbon.ReadTimeout=2000

Ovako postavljen *Feign* klijent raspoređuje opterećenje na različite instance mikrousluga istog tipa u pomoć *Ribbona*, te sprječava propagaciju pogrešaka uz pomoć *Hystrixa.*

## Ovisnosti

<dependency>

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

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

# Pristup mikrouslugama

Prilikom implementacije sustava s mikrouslugama jedan od bitnijih problema koje trebamo razriješiti je način komunikacije među uslugama i klijentskim aplikacijama:

* Radi lakše održivosti sustava potrebno je ostvariti slabu povezanost među uslugama i klijentskim aplikacijama. Ako dođe do promjene na pojedinoj usluzi ili ako dođe do ispada usluge to ne bi smjelo utjecati na rad drugih usluga i na rad klijentskih aplikacija.
* Održavanje podataka o postojećim uslugama, provjera dostupnih resursa koje može nuditi više usluga ili otkrivanje novih dostupnih usluga su poslovi o kojima klijentska aplikacija ne bi trebala brinuti
* Klijentu prilikom upućivanja zahtjeva nije bitno od koje usluge je primio traženi resurs ili da li je pružatelj usluge promijenio lokaciju(server) i zbog toga bi ta komunikacija trebala biti transparentna
* Usluge mogu biti razdijeljene na više manjih usluga i mogu implementirati različite protokole za komunikaciju što ne bi smjelo promijeniti način na koji klijent može pristupiti traženim resursima
* Klijentski zahtjev može zatražiti resurs koji je raspodijeljen na više usluga, a koji bi trebalo biti moguće dohvatiti upućivanjem jednog zahtjeva
* Više usluga može nuditi isti resurs. Kako bi se spriječilo opterećenje pojedine usluge potrebno je ostvariti mehanizam podjele zahtjeva na usluge.

Kako bismo ostvarili navedene zahtjeve nameće nam se rješenje u vidu ostvarivanja jedinstvenog sučelja preko kojeg će klijent pristupati uslugama sustava. To sučelje predstavlja jedinstveni pristupnik uslugama koji će prema klijentu biti rijetko promjenjiv ili gotovo nepromjenjiv. Klijentski zahtjevi se šalju na sučelje koje zatim prosljeđuje zahtjeve uslugama koje raspolažu traženim resursima. Ukoliko postoji usluga koja pruža traženi resurs taj resurs će preko pristupnika biti vraćen klijentu. Ako dođe do promjene ili ispada usluge klijent će i dalje moći poslati zahtjev na pristupnik, a da pri tom ne moramo mijenjati logiku klijentske aplikacije. Pristupnik je taj koji će se brinuti o ponovnom slanju zahtjeva na neku drugu uslugu koja pruža traženi resurs. Budući da svi klijentski zahtjevi prolaze kroz pristupnik moguće je raditi stvari kao što su rasterećenje pružatelja usluga, provjera zahtjeva radi sigurnosti, logiranje i praćenje sustava na krajnjim čvorovima sustava. Za implementaciju pristupnika nam mogu poslužiti gotovi alati kao što su Zuul i Spring Cloud Gateway.

## Zuul

Zbog prirode usluga koje Netflix kompanija pruža korisnicima(video sadržaj) morali su se često suočavati s problemima opterećenja sustava zbog čega su morali tražiti rješenje problema koje će im omogućiti brze izmjene sustava kako bi se mogli prilagoditi takvim događajima kada je to potrebno. Rješenje koje su razvili je Zuul pristupnik uslugama.

Kada klijent pošalje zahtjev na Zuul pristupnik, zahtjev se obrađuje kroz niz filtara. Filtar je temeljna komponenta koja se sastoji od:

* Tipa filtra – obično izražava fazu u kojoj se filtar izvodi tijekom životnog ciklusa obrade zahtjeva, ali može biti bilo koji niz znakova
* Prioriteta filtra
* Uvjeta filtra – filtar se neće izvoditi ukoliko uvjet nije zadovoljen
* Akcije – akcija koja će se izvršiti ukoliko je uvjet zadovoljen

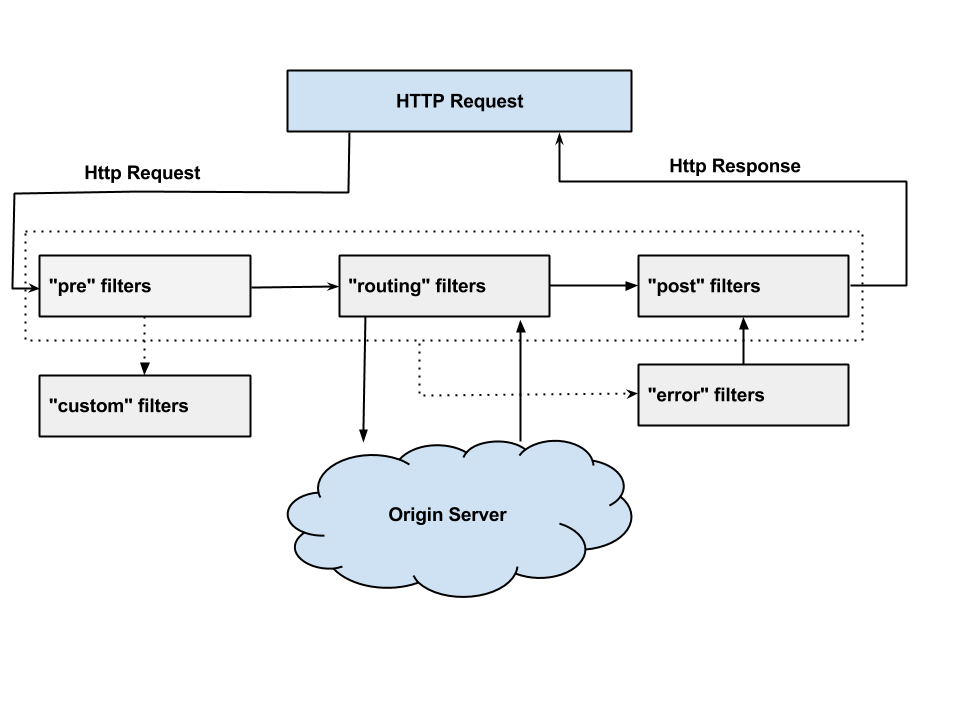
Zuul pruža radni okvir u kojem se filtri mogu dinamički učitavati, prevoditi i izvoditi. Filtri ne komuniciraju međusobno, već dijele stanje kroz kontekst zahtjeva koji je jedinstven za svaki pojedinačni zahtjev.

Iako je moguće definirati bilo koji tip filtra i pozvati ga eksplicitno, postoji nekoliko standardnih tipova filtara koji odgovaraju fazama životnog ciklusa obrade zahtjeva(Slika 7):

* PRE – filtri koji se izvode prije obrade i prosljeđivanja zahtjeva na servis(autentifikacija, logiranje,...)
* ROUTING – filtri kojima se zahtjev obrađuje i kojima se grade zahtjevi prema dostupnim servisima
* POST – filtri koji se izvode nakon pripremanja zahtjeva za servise(postavljaju standardna zaglavlja, skupljanje statistike, prosljeđivanje odgovora od servisa do klijenta...)
* ERROR – filtri koji se izvode kada se u bilo kojoj fazi dogodi greška

Ovakva struktura pristupnika omogućava mnogo funkcionalnosti kao što su autentifikacija zahtjeva, rasterećenje usluga, logiranje sustava, kontrola sigurnosti, brza prilagodba sustava u slučaju pojave problema i slično. [17]

Trenutno je najavljen i u izradi Zuul 2 koji, za razliku od prve verzije Zuula, radi asinkrono čime žele postići veću propusnost sustava, bolja svojstva skalabilnosti i bolju otpornost na greške sustava.



Slika 11. Životni ciklus obrade zahtjeva

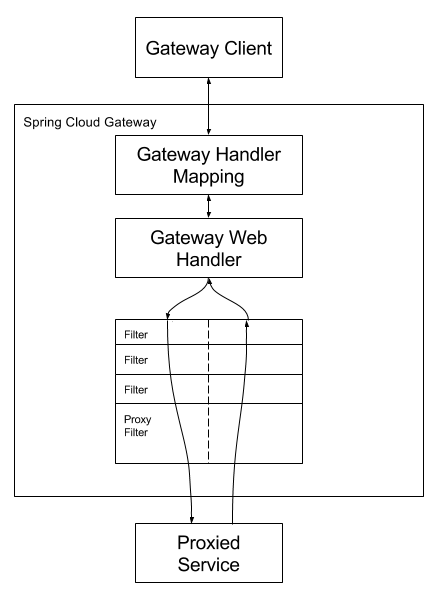
## Spring Cloud Gateway

Spring Cloud Gateway(SCG) je pristupnik uslugama koji je izgrađen na Spring ekosistemu uključujući Spring 5, Spring Boot 2 i Project Reactor. SCG pokušava pružiti jednostavan način postavljanja ruti za preusmjeravanje klijentskih zahtjeva na servise te ujedno pruža mogućnosti za nadziranje sustava, kontrolu sigurnosti i mehanizme zaštite i oporavka uslijed pojavljivanja grešaka u sustavu.

Temeljni koncepti SCG-a su rute, predikati i filteri:

* Rute – Osnovni elementi za izgradnju logike pristupnika. Ruta je definirana pomoću IDa, destinacijskog URIa, listom predikata i listom filtera
* Predikati – uspoređuju zahtjeve sa zadanim parametrima te služe za mapiranje zahtjeva i ruta.
* Filteri – služe za izmjenu parametara zahtjeva prije prosljeđivanja(postavljanje polja zaglavlja, podešavanja putanje na koju se zahtjev prosljeđuje, ...)

Na slici je vidljiv dijagram SCG-a. Prilikom zaprimanja zahtjeva od klijenta taj zahtjev se pokušava mapirati na jednu od definiranih ruta. To će se desiti ukoliko zaprimljeni zahtjev zadovoljava sve predikate za nekoju od definiranih ruta. Ako zahtjev odgovara ruti onda se



Slika 12. Dijagram Spring Cloud Gatewaya

prosljeđuje Gateway Web Handleru(GWH). GWH je zadužen za slanje zahtjeva u lanac filtara koji se obrađuju prije prosljeđivanja zahtjeva servisima. Filtri prilagođavaju zahtjev prije prosljeđivanja. Nakon što se dobije odgovor od servisa, nad odgovorom se opet može pokrenuti izvršavanje filtara prije nego se odgovor vrati nazad klijentu. Na ovakav način se postiže jasna odvojenost između dijelova pristupnika koji provjeravaju valjanost zahtjeva i dijelova koji prosljeđuju zahtjev na servise. Tako se na primjer predikati mogu brinuti o provjeri sigurnosti zahtjeva zaprimljenih od klijenta, a filteri se mogu brinuti o prilagođavanju zahtjeva kako bi odgovarao sučelju servisa.[15][16]

SCG nudi mogućnost konfiguracije ruta, predikata i filtara pomoću YAML konfiguracijskih datoteka. Prilikom konfiguracije je potrebno definirati id rute, odredišni URI te opcionalno predikate i filtre koji se primjenjuju nad tom rutom. Primjer konfiguracije rute koja mapira sve zahtjeve koji su pristigli iza zadane oznake vremena dan je sljedećim isječkom YAML datoteke:

spring:

cloud:

gateway:

routes:

# =====================================

- id: after\_route

uri: http://example.org

predicates:

- After=2017-01-20T17:42:47.789-07:00[America/Denver**]**

Predikati koji mogu biti definirani u YAML datoteci su:

* After – mapira na rutu sve zahtjeve primljene iza zadane oznake vremena
* Before – mapira na rutu sve zahtjeve primljene prije zadane oznake vremena
* Between – mapira na rutu sve zahtjeve primljene između zadanih oznaka vremena
* Cookie – mapira zahtjev na rutu ukoliko vrijednost zadanog kolačića odgovara danom regularnom izrazu
* Header – mapira zahtjev na rutu ukoliko vrijednost zadanog zaglavlja odgovara danom regularnom izrazu
* Host – mapira zahtjev na rutu ukoliko vrijednost Host zaglavlja odgovara danom uzorku
* Method – mapira zahtjev na rutu ukoliko vrijednost HTTP metode odgovara traženoj metodi
* Path – mapira zahtjev na rutu ukoliko putanja zahtjeva odgovara traženoj putanji. Moguće je mapirati sve putanje s određenim prefiksom i pohraniti pojedine dijelove putanje kao parametre zahtjeva.
* Query – mapira zahtjev na rutu ukoliko se traženi parametar nalazi u putanji ili ukoliko parametar u putanji odgovara traženoj vrijednosti
* RemoteAddr – mapira zahtjev na rutu ukoliko IP adresa klijenta odgovara traženoj IP adresi koja je zadana u CIDR obliku.

Filtri koji mogu biti definirani u YAML konfiguraciji su:

* AddRequestHeader – dodaje zaglavlje prije prosljeđivanja servisu
* AddRequestParameter – dodaje parametar prije prosljeđivanja servisu
* AddResponseHeader – stavlja zaglavlje na odgovor koji je dobiven od servisa
* Hystrix
* PrefixPath – dodaje prefiks na putanju na koju prosljeđuje zahtjev
* RequestRateLimiter – služi za definiranje količine pristupa koji može biti omogućen korisniku u zadanom vremenu
* RedirectTo – vraća klijentu odgovor sa zadanim URL-om na koji klijent treba dalje poslati zahtjev i sa statusom koji treba biti pripadati grupi statusa **3XX**
* RemoveNonProxiHeaders – uklanja ne-proxi zaglavlja prilikom prosljeđivanja zahtjeva servisu
* RemoveRequestHeader – uklanja zadano zaglavlje prilikom prosljeđivanja zahtjeva servisu
* RemoveResponseHeader - uklanja zadano zaglavlje prilikom vraćanja odgovora klijentu
* RewritePath – Mjenja putanju na koju prosljeđuje zahtjev prema servisima
* SecureHeaders – dodaje preporučena sigurnosna zaglavlja u odgovor klijentu
* SetPath – postavlja putanju na koju prosljeđuje zahtjev
* SetResponseHeader – mijenja vrijednost zaglavlja ukoliko odgovor sadrži traženo zaglavlje
* SetStatus – postavlja status odgovora

Bitno je napomenuti da se može kombinirati više predikata ili filtara na pojedinoj ruti. Vidimo da nam SCG daje velike mogućnosti predikata i filtara da na jednostavan način možemo postaviti konfiguraciju ruta kojom možemo pokriti većinu slučajeva koji se pojavljuju u praksi.

Kako je mapiranje ruta u SCG-u napravljeno na Spring WebFlux HandlerMapping infrastrukturi, SCG je po prirodi asinkron. To znači da ima bolje predispozicije za skalabilnost sustava od Zuul pristupnika. Zbog toga, a I zbog jednostavnosti integracije SCG-a sa Springom, SCG se isplati dalje istraživati.[18][19]

## Implementacija Spring Cloud Gateway servisa

Alati koji su korišteni prilikom implementacije SCG servisa su:

* Eclipse razvojna okolina
* Spring Boot 2.0.0.M7
* Gradle(Buildship 2)
* Java 1.8

Novi projekt možemo generirati preko stranice „<https://start.spring.io/>“. Kako bi smo dodali SCG u projekt potrebno je uključiti *spring-cloud-starter-gateway* u naš projekt dodavanjem *compile('org.springframework.cloud:spring-cloud-starter-gateway')* u gradle.build document:

dependencies {

//spring cloud gateway

compile('org.springframework.cloud:spring-cloud-starter-gateway')

//

compile('org.springframework.cloud:spring-cloud-starter-netflix-ribbon')

compile group: 'org.springframework.cloud', name: 'spring-cloud-starter-netflix-eureka-client'

compile group: 'org.springframework.boot', name: 'spring-boot-starter-actuator'

compile group: 'org.springframework.cloud', name: 'spring-cloud-starter-config'

}

Nakon što smo dodali SCG u projekt potrebno je locirati klasu iz koje pokrećemo aplikaciju i dodati anotaciju *@EnableAutoConfiguration* kako bi se omogućila automatska konfiguracija Beanova potrebnih za ispravan rad SCG servisa:

@SpringBootConfiguration

@EnableAutoConfiguration

@EnableEurekaClient

**public** **class** GatewaySampleApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(GatewaySampleApplication.**class**, args);

}

}

Dodatne stvari na koje treba paziti su te da je SCG napravljen tako da zahtjeve obrađuje reaktivno što omogućuje asinkronu obradu zahtjeva. Zbog toga se SCG pokreće na Netty reaktivnom okruženju(umjesto na Tomcat okruženju) ako drugačije nije postavljeno i stoga se savjetuje da se okolina ne mijenja. Ukoliko je u projekt uključen *spring-cloud-starter-web* projekt doći će do konflikta te je potrebno ukloniti ovisnost o tom projektu.

Na kraju je potrebno definirati rute i pravila za mapiranje dolaznih zahtjeva na servise što je moguće napraviti pomoću YAML dokumenta. Unutar „resources“ datoteke kreiramo *application.yml* dokument. Unutar dokumenta definiramo rute prema sljedećoj strukturi:

|  |
| --- |
|  |
|  |
|  | spring:  cloud: |
|  | gateway: |
|  | routes: |
|  | - id: |
|  | addConstant |
|  | uri: |
|  | lb://ms-addition |
|  | predicates: |
|  | - Path=/addConstant |
|  | - Query=a |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  | - id: |
|  | multiply |
|  | uri: |
|  | lb://ms-multiplication |
|  | predicates: |
|  | - Path=/multiply |
|  | - Query=a |
|  | - Query=b |
|  | filters: |
|  | - RewritePath=/multiply(?<segment>.\*), /multi$\{segment} |
|  |  |

U konfiguraciji vidimo da su rute locirane pod spring:cloud:gateway:routes granom u koju pohranjujemo listu ruta(znak „-“ u ravnini s „routes“ označava element liste). Ruta se sastoji od svoga identifikatora, URIa koji nam označava adresu na koju prosljeđujemo zahtjev, liste predikata koji služe za definiranje pravila za uspoređivanje dolaznih zahtjeva sa rutom i od filtara koji nam služe za modificiranje zahtjeva prilikom prosljeđivanja zahtjeva prema servisu ili prilikom vraćanja odgovora klijentu. Moguće je na jednoj ruti definirati više filtara i predikata. Prosljeđivanje na servise koje smo dohvatili s eureka servera radimo tako da za uri stavimo naziv pod kojim se servis registrira na Eureki i prefiksiramo ga s „lb://“. Time zahtjeve prosljeđujemo na *load balancer* koji se brine o tom da se zahtjevi rasporede na servise koji su registrirani pod zadanim imenom kako bi se sustav mogao rasteretiti prilikom većih opterećenja. Neke od opcija koje možemo postaviti za predikate i filtre su prikazane u gornjem primjeru. Iz danog primjera je vidljivo da pomoću predikata možemo odrediti da prihvaćamo zahtjeve s određenom putanjom resursa i s određenim imenima *query* parametara te da također možemo filtrirati putanju resursa i promjeniti je u neku drugu putanju i to sve na vrlo jednostavan način. Ovime možemo osigurati trasparentnost servisa i osigurati da se API prema klijentu ne mora nužno mijenjati ukoliko smo nešto promijenili na servisima.

## Generiranje jar datoteke

Da bi se mogla izgenerirati jar datoteka koju ćemo moći na kraju pokrenuti potrebno je dodati sljedeći *jar* tip zadatka u gradle.build dokument:

jar {

manifest {

attributes 'Main-Class': 'com.example.demo.GatewaySampleApplication'

}

from { configurations.compile.collect { it.isDirectory() ? it : zipTree(it) } }

}

Atribut „Main-Class“ traži puni naziv klase(s nazivom paketa u kojem se nalazi) iz koje pokrećemo Spring Boot aplikaciju. „from“ redak traži sve datoteke o kojima ovisi aplikacija i dodaje ih u jar.

Zatim je potrebno u terminalu pozicionirati se u direktorij projekta i pokrenuti naredbu „*gradlew.bat build“* (za Windows OS) čime će se izgenerirati „build“ datoteka pod kojom možemo pronaći generirani jar. [20]

# Pokretanje

* sa sljedeće [poveznice](https://docs.docker.com/engine/installation/#supported-platforms) skinuti Docker (Stable)
* iz terminala pokrenuti sljedeću naredbu: docker pull rabbitmq:3-management
* iz terminala pokrenuti sljedeću naredbu: docker run -d --hostname my-rabbit --name some-rabbit -p 15672:15672 -p 5672:5672 rabbitmq:3-management
* raspakirati zip arhivu sa izvšnim jar datotekama
* u terminalu se pozicionirati u direktorij gdje se nalaze raspakirane izvršne jar datoteke i pokrenuti sljedeću naredbu : java -jar SpringCloudConfig-server-0.0.1-SNAPSHOT.jar
  + sada je konfiguracijski poslužitelj pokrenut na portu 8980
* GitHub repozitorij sa konfiguracijskim datotekama nalazi se na sljedećoj [poveznici](https://github.com/jakovzivkovic/rassus-config-files)
* za preuzimanje programa za izlaganje porta na Internet otvoriti sljedeću [poveznicu](https://ngrok.com/download)
* u terminalu se pozicionirati u direktorij u kojem se nalazi preuzeti program i izvršiti sljedeću naredbu: ./ngrok http 8980
* pomoću dobivenog linka ažurirati GitHub Webhook i dodati /monitor na kraj
* pozicionirati se u direktorij sa izvršnim jar datotekama te pokrenuti sljedeće naredbe u zasebnim terminalima:
  + java -jar EurekaServer-0.0.1-SNAPSHOT.jar
    - na portu 8761 može se provjeriti je li uspješno pokrenuto (Eureka server)
  + java -jar MicroserviceAddition-0.0.1-SNAPSHOT.jar
    - na portu 8761 provjeriti je li uspješno prijavljena mikrousluga ms-addition
    - ispitati rad mikrousluge pokretanjem sljedećeg upita:
      * htttp://localhost:{dodijeljeni port}/add?a=2&b=3
      * dodijeljeni port moguće očitati iz terminala ili Eureke
  + java -jar MicroserviceMultiplication-0.0.1-SNAPSHOT.jar
    - na portu 8761 provjeriti je li uspješno prijavljena mikrousluga ms-multiplication
    - ispitati rad mikrousluge pokretanjem sljedećeg upita:
      * htttp://localhost:{dodijeljeni port}/multi?a=2&b=3
      * dodijeljeni port moguće očitati iz terminala ili Eureke
  + java -jar FeignDistance-0.0.1-SNAPSHOT.jar
    - na portu 8761 provjeriti je li uspješno prijavljena mikrousluga feign-distance
    - ispitati rad mikrousluge pokretanjem sljedećeg upita:
      * htttp://localhost:{dodijeljeni port}/distance?ax=2&bx=3&ay=1&by=5
      * dodijeljeni port moguće očitati iz terminala ili Eureke
  + java -jar GatewayApp-0.0.1-SNAPSHOT.jar
    - na portu 8761 provjeriti je li uspješno prijavljena mikrousluga spring-cloud-gateway
    - ispitati rad pokretanjem sljedećeg upita:
      * htttp://localhost:**8030**/distance?ax=2&bx=3&ay=1&by=5
  + java –jar zipkin-server-2.4.3-exec.jar
    - na portu 9411 provjeriti je uspješno pokrenut Zipkin server
* preuzeti Elasticsearch, Logstash, Kibana sa [poveznice](https://download.elasticsearch.org/)
* raspakirati arhive te pokrenuti sljedeće naredbe:
  + ./elasticsearch-{version}/bin/elasticsearch
    - na portu 9200 provjeriti je li uspješno pokrenuto
  + ./kibana-{version}/bin/kibana
    - Na portu 5601 provjeriti je li uspješno pokrenuto
  + ./logstash-{version}/bin/logstash agent -v -f /path/to/logstash.conf

# Zaključak

Razvijanje sustava pomoću mikrousluga nam omogućava brzu prilagodbu sustava korisničkim zahtjevima te poboljšava otpornost na ispade usluga. Dodavanje mikrousluga se svodi na izgradnju kompaktnijih i jednostavnijih cjelina, no cijena toga je veliki broj komponenti koje trebamo pratiti. Servisi pomoću kojih registriramo usluge ili pomoću kojih konfiguriramo usluge su nam stoga vrlo bitni radi lakšeg održavanja i upravljanja uslugama. Zbog velikog broja različitih mikrousluga preporuča se njihove konfiguracije staviti na jedno mjesto od kuda ih možemo jednostavno podešavati bez potrebe za njihovim ponovnim pokretanjem. Za prikaz dostupnih usluga dobra je praksa koristiti registar putem kojega možemo doznati lokaciju potrebne mikrousluge koja se dinamički mijenja. Prilikom razvijanja sustava koji se sastoji od više komponenti dobro je voditi dnevnik zapisa o izvođenju svake komponente. To pruža uvid u globalno stanje te olakšava pronalaženje pogrešaka prilikom izvođenja raspodijeljenog sustava.

# Literatura

1. <http://www.devramble.com/2017/07/microservices-platform-with-spring-cloud-part-4/>
2. <http://ryanjbaxter.com/cloud/spring%20cloud/spring/2016/07/07/spring-cloud-sleuth.html>
3. <https://howtodoinjava.com/spring/spring-cloud/spring-cloud-zipkin-sleuth-tutorial/>
4. <https://logz.io/learn/complete-guide-elk-stack/>
5. <http://knes1.github.io/blog/2015/2015-08-16-manage-spring-boot-logs-with-elasticsearch-kibana-and-logstash.html>
6. <https://blog.codecentric.de/en/2014/10/log-management-spring-boot-applications-logstash-elastichsearch-kibana/>
7. <https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance>
8. <http://www.baeldung.com/spring-cloud-netflix-eureka>
9. <http://www.baeldung.com/introduction-to-hystrix>
10. <https://github.com/Netflix/Hystrix>
11. <https://github.com/Netflix/ribbon>
12. <http://www.baeldung.com/spring-cloud-rest-client-with-netflix-ribbon>
13. <http://www.baeldung.com/intro-to-feign>
14. <https://github.com/OpenFeign/feign>
15. <http://cloud.spring.io/spring-cloud-gateway/single/spring-cloud-gateway.html>
16. <http://cloud.spring.io/spring-cloud-gateway/#quick-start>
17. <https://medium.com/netflix-techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c>
18. <http://www.baeldung.com/spring-cloud-bootstrapping>
19. <https://github.com/spring-cloud/spring-cloud-gateway>
20. <http://www.baeldung.com/spring-cloud-gateway>