Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/06BE02Bratic

PRIMENA SPRING CLOUD PROGRAMSKOG OKVIRA ZA RAZVOJ MIKROSERVISNIH APLIKACIJA

USE OF SPRING CLOUD FRAMEWORK FOR DEVELOPING MICROSERVICE-BASED APPLICATIONS

Stefan Bratić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *Prikaz osnovnih prednosti i mana mikroservisne arhitekture, kao i konkretna primena Spring Cloud radne biblioteke uz pomoć Jhipster generatora na konkretnom projektu.*

Ključne reči: *Mikroservisi, Jhipster, Spring Cloud, Kubernetes, Docker*

Abstract – *Representation of basic advantages and flaws in the microservice architecture, as well as a concrete implementation through use of Spring Cloud Framework and Jhipster generator.*

Keywords: *Microservices, Jhipster, Spring Cloud, Kubernetes, Docker*

1. UVOD

Proces pravljenja softverskih rešenja se sastoji iz niza procesa, koji su neophodni kako bi konačno rešenje ispunilo sve zahteve klijenta. U pomenute procese spadaju planiranje, razvoj, testiranje i isporuka softverskog rešenja.
Sastavni deo procesa planiranja čini odabir softverske arhitekture. Razlog tome je, što svaka softverska arhitektura predstavlja stub izgradnje softvera i utiče na to kako će se odvijati proces razvoja softvera.
Da bi se moglo znati koja je softverska arhitektura najpogodnija, potrebno je znati karakteristike date arhitekture.
U skladu sa tim, glavni fokus ovog rada će biti mikroservisna arhitektura, koja danas ima sve veću primenu.
Teme koje će biti obrađene u radu su sledeće:

- osnovne karakteristike mikroservisne arhitekture
- razlike između mikroservisne i monolitne arhitekture
- osnovne komponente mikroservisne arhitekture
- primena *Spring Cloud* radnog okvira za razvoj mikroservisnih aplikacija

Cilj rada predstavlja detaljan prikaz mikroservisne arhitekture upotrebom postojećeg radnog okvira. Takođe, uradiće se analiza u kojim slučajevima je mikroservisna arhitektura dobro rešenje za razvoj softvera.
Kompletna analiza će biti prikazana kroz konkretno softversko rešenje.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Branko Milosavljević, red. prof.

1.1 Mikroservisna arhitektura

Mikroservisnu arhitekturu čini skup aplikacija zvanih mikroservisi. Mikroservisi su autonomni i izolovani entiteti, čiji je cilj izvršavanje dela funkcionalnosti kompletnog sistema i skladno komuniciranje sa ostalim učenicima u datom sistemu [1].
Glavne odlike mikroservisne arhitekture su:

- autonomnost
- tehnološka heterogenost
- robustnost
- skalabilnost
- pojednostavljen deployment postupak
- organizacijsko poravnjanje
- kompozitna struktura

1.2 Autonomnost

Autonomnost se ogleda u tome što je svaki mikroservis zaseban entitet, koji se posebno *deploy-uje* na platformu ili izvršava kao poseban računarski proces izolovan od ostalih učesnika u sistemu. Svaki vid komunikacije se vrši isključivo preko računarske mreže i ne postoji nikakva sprega na nivou samog koda [1].

1.3 Tehnološka heterogenost

Pošto su mikroservisi definisani kao autonomne i izolovane jedinice, samim tim, programski kod, koji se koristi za razvijanje određenih delova sistema, je odvojen. To daje mogućnost da se delovi sistema razvijaju u različitim programskim jezicima, tehnologijama i da se koriste različiti izvori perzistencije [1].

1.4 Robustnost

Pošto je prethodno ustanovljeno da mikroservisi funkcionišu kao autonomni i izolovani entiteti, oni u skladu sa tim po istim principima reaguju na sistemske greške.

Robustnost mikroservisne arhitekture se ogleda u tome što ne postoji propagacija sistemske greške jednog mikroservisa na ceo sistem. Samim tim, sistem je otporniji na greške u pojedinim delovima sistema, jer je u mogućnosti da i pored greške funkcioniše dalje i usput vrši oporavak delova sistema [1].

1.5 Skalabilnost

Svaki mikroservis čini jedan proces, preodređen da izvršava jednu funkcionalnost. Ograničavanje jedne funkcionalnosti po mikroservisu omogućava bolje

raspoređivanje hardverskih resursa, zato što je omogućeno skaliranje mikroservisa, čija funkcionalnost u toku upotrebe zahteva veće resurse da bi opslužila korisnike [1].

1.6 Pojednostavljen *deployment* postupak

Svaka mikroservisna aplikacija se *deploy-uje* zasebno i promena koda jedne mikroservisne aplikacije ne zahteva lansiranje celokupnog sistema, već samo dela, kojem pripada promenjeni kod.

Pojednostavljen deployment postupak se ogleda u tome što se promene mogu lakše ispratiti i teže mogu izazvati grešku kod ostalih delova sistema [1].

1.7 Organizacijsko poravnanje

Izolovanost mikroservisnih aplikacija daje prostora da se određeni timovi ili članovi tima fokusiraju na samo određen mikroservis ili skup mikroservisa. Ovakav pristup je u skladu sa *Conway's law*, koji glasi: **"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure".**

Prevedeno znači da svaka organizacija, koja definiše neki sistem, će definisati svoju strukturu na osnovu organizaciono-komunikacione strukture koju sama koristi.

To se pokazuje kao dobra praksa, zato što su timovi odgovorni samo za određeni deo koda celog sistema i spram toga moguće je lakše održavati, razvijati sistem i komunicirati sa ostalim timovima u vezi sa integracijom delova sistema [1].

1.8 Kompozitna struktura

Svaka mikroservisna aplikacija čini jednu komponentu sistema sa određenim funkcionalnostima. Mikrosevisna arhitektura omogućava da se skup komponenti orkestrirano koristi u cilju izvršavanja složenih procesa [1].

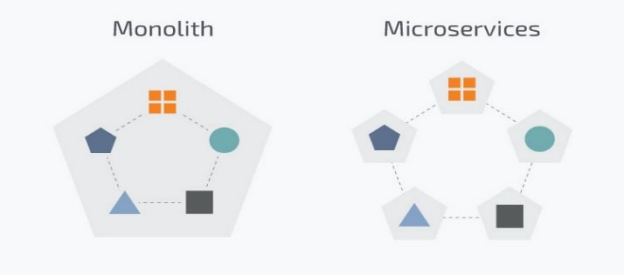
2. KOMPARATIVNI PRIKAZ MIKROSERVISNE I MONOLITNE ARHITEKTURE

Mikroservisna arhitektura sa sobom nosi mnoge prednosti, a takođe i izazove. Ona ne predstavlja rešenje za svaki mogući slučaj. U skladu sa tim, da bi se sagledale činjenice u kojim situacijama je mikroservisna arhitektura pogodna kao rešenje, uradiće se komparativni prikaz mikroservisne i monolitne arhitekture.

Monolitna arhitektura, u odnosu na mikroservisnu arhitekturu, predstavlja jednostavniji način razvijanja softvera, jer sistem nije distribuiran i celokupan kod je u jednoj tehnologiji i programskom jeziku. Međutim, kod aplikacija, koje izlaze van okvira jednog razvojnog tima, je teže održavati stabilnim ako nad tim okvirom koda treba da radi veliki broj ljudi.

Takođe, navedeni način struktuiranja aplikacije može se primenjivati na nivou mikroservisa, jedina razlika je što postoji granulacija funkcionalnosti po entitetima.

Slika 1 daje vizuelni prikaz razlike između monolitne arhitekture, koja sadrži sve funkcionalnosti u okviru jednog entiteta, i mikroservisne arhitekture, gde se svaka funkcionalnost nalazi izolovano u zasebnom entitetu.



Slika 1: *Prikaz razlike mikroservisne u odnosu na monolitnu arhitekturu*

3. *SPRING* RADNI OKVIR

Spring predstavlja radni okvir napisan u *Java* programskom jeziku i razvijen je od strane firme Pivotal. U početku je *Spring* predstavljao biblioteku za uvođenje *Dependency Injection Pattern-a* u softver, a kasnije se namena biblioteke proširila na širok dijapazon problema, koji se tiču razvoja *Enterprise* softvera.

3.1. Spring Cloud

Spring Cloud, kao deo *Spring* radnog okvira, pretežno je namenjen za razvijanje distribuiranih sistema, gde se pojavljuju problemi poput: upravljanja konfiguracijom i distribuiranim sesijama, otkrivanja usluga, dinamičkog rutiranja itd.

Pošto u okviru *Spring Cloud* radnog okvira postoji veliki broj rešenja, prikazaće se oni delovi radnog okvira, koji rešavaju probleme navedenih *Design Pattern-a* u mikroservisnom okruženju.

Ti delovi radnog okvira su:

- Spring Cloud Config
- Spring Cloud Netflix
- Spring Cloud Gateway
- Spring Cloud Security
- Spring Cloud Sleuth

3.2. Spring Cloud Config

Spring Cloud Config, koji implementira *Central Configuration Pattern*, nudi klijentsku i serversku podršku za eksternu konfiguraciju u distribuiranim sistemima, gde se sa konfiguracionim serverom uvodi centralno mesto za upravljanje eksternom konfiguracijom za aplikaciju u okviru svih okruženja [3].

Odlike *Spring Cloud Config Server-a* su:

- HTTP-bazirani API za eksternu konfiguraciju
- enkripcija i dekripcija konfiguracionih vrednosti
- mogućnost korišćenja u okviru *Spring Boot* aplikacije

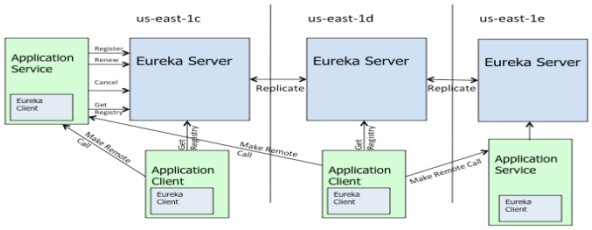
3.3. *Spring Cloud Netflix*

Spring Cloud Netflix je integracija *Netflix OSS* biblioteke, koja rešava probleme distribuiranih sistema, u *Spring* ekosistemu. *Pattern-i*, koje *Spring Cloud Netflix* obuhvata su:

- otkrivanje servisa pomoću *Eureke*
- pametno rutiranje pomoću *Zuul-a*
- prekidanje puta (*Circuit Breaker*) pomoću *Hystrix-a*

- klijent-bazirano balansiranje opterećenja sa *Ribbon-om*

Eureka je *REST*-baziran servis, koji se primarno koristi za *AWS (Amazon Web Services) Cloud* okruženje u svrhu lociranja servisa i balansiranja opterećenja poslatih zahteva. Uloga *Eureke* servisa je da opslužuje zahteve *Eureka* klijenata, kojima je potrebna informacija o adresi željenih servisa [2].



Slika 2: *Prikaz arhitekture Eureka Server-a*

Slika 2 prikazuje da je *Eureka server* napravljen da se sastoji iz više replikacija po različitim geografskim lokacijama i da pruža indirektan pristup klijentima ka željenoj aplikaciji.

Svaki servis, koji želi da bude na raspolaganju drugim servisima, pri pokretanju se registruje kod *Eureka server-a* i na svakih 30 sekundi treba da šalje *Heartbeat* odnosno poruku da je i dalje u funkciji. Takođe, proces registracije se vrši samo kod jednog servera, a potom se ta informacija o registrovanju servisa šalje svim ostalim instancama *Eureka servera* [2].

Zuul predstavlja pristupnu tačku za sve zahteve, koja se upućuje u sistem od strane klijenata, te omogućava dinamička rutiranja, osmatranja, otpornost i sigurnost zahteva. *Zuul* predstavlja konkretnu implementaciju *API* Prolaz *Design Pattern-a*.

Zuul je razvijem od strane kompanije Netflix i koristi se za sledeće stvari:

- autentifikaciju
- razne vrste testiranja
- praćenje i nadgledanje zahteva
- dinamičko rutiranje
- integraciju servisa
- bezbednost
- balansiranje opterećenja
- upravljanje saobraćajem

U *Spring Cloud* ekosistemu, *Zuul* je ugrađen u okviru *Spring* aplikacije, kako bi se pojednostavio njen razvoj.

Česta pojava u distribuiranim sistemima jeste da jedan deo sistema otkaže. U takvim okolnostima, zahtev uglavnom izvrši više skokova na više različitih servisa pre nego što vrati odgovor klijentu. Spram toga, ukoliko na putanji zahtev dobije grešku, ta greška se može propagirati kroz više servisa davajući mogućnost da sistem postane nestabilan, a u najgorem slučaju nedostupan.
Kako bi se taj problem prevazišao, uvodi se *Circuit breaker* biblioteka pod imenom *Hystrix*, koja uvodi sledeće funkcionalnosti:

- kontrola kašnjenja i grešaka zahteva
- zaustavljanja proširenja jedne greške na ceo distribuiran sistem
- oporavak sistema u slučaju greške
- nadgledanje sistema

Važnost otpornosti sistema, posebno u kompleksnim distribuiranim sistema, pokazuje računica da sistem od 30 servisa ima 99.99% dostupnost. Kumulativno se dobija totalna dostupnost sistema $99{,}99\%^{30}=99{,}7\%$. Kada se prevede nedostupnost sistema od 3% na bilion zahteva, na mesečnom nivou to predstavlja oko 3 miliona grešaka što rezultira da sistem ne bude u funkciji 2 ili čak više sati na mesečnom nivou [4].

3.4. Spring Cloud Security

Spring Cloud Security predstavlja skup primitiva da se na deklarativan način dobiju funkcionalnosti, koje su potrebne da bi sistem bio siguran, kao što su identifikacija, autentifikacija i autorizacija [5].

Jedne od glavnih funkcionalnosti su:

- transfer *Single Sign On* tokena sa *Frontend* strane na *Backend* stranu preko *Zuul* proksija
- konfigurisanje autentifikacije u *Zuul* proksiju
- transfer tokena između resursnih servera.

3.5. *Spring Cloud Sleuth*

Spring Cloud Sleuth je implementacija distribuiranog praćenja u *Spring* okruženju. Većina implementacije se svodi na adaptiranje postojećih rešenja za distribuirano praćenje, kao što su: *Zipkin*, *Dapper* i *HTrace*, i njihovo korišćenje u *Spring Cloud*-baziranim sistemima [6].

Osnovna jedinica rada u *Spring Cloud Sleuth-a* predstavlja entitet po imenu *Span*. *Span* predstavlja jedinstveno identifikovani dnevnik vremenski označenih zapisa, koji čuva početak i kraj operacije. Pomenuti zapisi se identifikuju pomoću 64-bitnog identifikatora tzv. *span id* i još jednog 64-bitnog identifikatora zvanog *trace id*, koji predstavlja identifikator *trace-a*, čiji je *span* deo. Pored identifikatora, *span-ovi* mogu sadržati u sebi i druge podatke kao što su: opisi, identifikator procesa tj. IP-adresa i slično [6].

Svaki *span* se programski započinje i završava, te se na taj način čuvaju informacije o vremenu [6].

Korišćenjem *span-ova* i *trace-ova* daje se mogućnost da se operacije koje se dešavaju distribuirano povezuju, te na taj način dovode do lakšeg uočavanja problema u sistemu.

4. ZAKLJUČAK

Na osnovu iznetih podataka o mikroservisnoj arhitekturi, može se zaključiti da je mikroservisna arhitektura po svojoj prirodi pogodna za razvoj sistema, koji su veliki i kompleksni. Veličina sistema se ogleda u količini koda, koja je potrebna da bi se dati sistem razvio. Ukoliko bi se celokupan sistem održavao na jednom mestu, došlo bi do velikog opterećenja pri održavanju i daljem razvoju datog sistema. Stoga, deljenje koda na izolovane celine daje veći stepen indirektnosti delova koda i smanjuje šansu pojave greške u sistemu.

Pored velikih prednosti, koje mikroservisna arhitektura može da pruži, javljaju se drugi problemi sa kojima razvojni tim mora da se nosi. Ti problemi polaze od činjenice da mikroservisna arhitektura razvija sistem kao distribuiran. Samim tim, problem nadgledanja, testiranja, *deploy-ovanja* celokupnog sistema funckioniše drugačije te je potrebno više uložiti vremena za te stvari u toku razvoja. Za skladan i uspešan rad mikroservisne arhitekture neophodno je podržati proces kontinuirane integracije ili kontinuirane isporuke, kako bi se problemi kod određenih delova sistema u najbržem roku uočili.

Mikroservisna arhitektura kao način razvoj projekta ne predstavlja najbolje rešenje za svaku situaciju. Ova arhitektura sa sobom nosi velike prednosti, ali takođe i traži veliku odgovornost i disciplinu razvojnog tima kako bi to imalo smisla. Stoga, možemo zaključiti da mikroservisna arhitektura predstavlja samo jedno od ponuđenih rešenja. Pre nego što projekat pređe u fazu razvoja, potrebno je dobro razmisliti koja će softverska arhitektura biti odabrana, jer arhitektura utiče na brzinu razvoja, rad i sigurnost sistema.

5. LITERATURA

[1] S. Newman, Building Microservices, O'Reilly Media, Inc., 2015.

[2] Pivotal Software, Inc., "Spring Cloud Config," 2019. URL: https://spring.io/projects/spring-cloud-config. (pristupljeno 07. 05. 2019).

[3] "Eureka Netflix," 2019. URL: https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance. (pristupljeno 16. 05. 2019).

[4] "Hystrix wiki," 2019. URL: https://github.com/Netflix/Hystrix/wiki. (pristupljeno 30. 5. 2019).

[5] "Spring Cloud Security," 2019. URL: https://cloud.spring.io/spring-cloud-security/spring-cloud-security.html. (pristpuljeno 30. 05. 2019).

[6] "Spring Cloud Sleuth," 2019. URL: https://spring.io/projects/spring-cloud-sleuth. (pristupljeno 13. 6. 2019).

[7] B. H. S. a. L. A. B. a. M. B. a. P. S. a. M. P. a. D. B. a. S. J. a. C. Shanbhag, „Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,“ 2010.

Kratka biografija:

Stefan Bratić rođen je 08.03.1994. godine u Novom Sadu. Godine 2008. završio je Osnovnu školu "Vuk Karadžić" u Novom Sadu. Srednju ekonomsku školu "Svetozar Miletić" u Novom Sadu završio je 2013. godine. Iste godine upisao je Fakultet tehničkih nauka na Univerzitetu u Novom Sadu, smer Softversko inženjerstvo i informacione tehnologije. Godine 2017. diplomirao je sa temom diplomskog rada *Upotrebna X-pack-a u Elastic stack tehnologiji*. Master Akademske Studije, smer Softversko inženjerstvo i informacione tehnologije, upisao je 2018. godine, gde je tokom te godine, dobivši *Best of South-East* stipendiju, pohađao kurseve na Tehničkom univerzitetu u Gracu i položio sve ispite koji su bili predviđeni planom i programom.

kontakt: stefanbraticns@gmail.com