Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 4.41
DOI: https://doi.org/10.24867/31BE18Rapic

MIGRACIJA MIKROSERVISNE ARHITEKTURE NA BEZSERVERSKO OKRUŽENJE UZ KORIŠĆENJE AWS SERVISA

MIGRATION FROM MICROSERVICE ARCHITECTURE TO A SERVERLESS PLATFORM USING AWS SERVICES

Marko Rapić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *Ovaj rad istražuje prelazak sa mikroservisne arhitekture na bezserversku infrastrukturu u cloud okruženju. Implementacija koristi AWS servise, uključujući API Gateway, DynamoDB i Lambda funkcije, radi poboljšanja performansi i smanjenja troškova. Nova arhitektura omogućava efikasnije upravljanje sistemom i integrisano praćenje putem alata za monitoring.*

Ključne reči: *Bezserversko okruženje, Mikroservisi, AWS*

Abstract – *This paper explores the transition from microservices architecture to serverless infrastructure in a cloud environment. The implementation utilizes AWS services, including API Gateway, DynamoDB and Lambda functions, to improve performance and reduce costs. The new architecture enables more efficient system management and integrated monitoring.*

Keywords: *Serverless environment, Microservices, AWS*

1. UVOD

Sa sve većim zahtevima za skalabilnošću i optimizacijom resursa, bezserverska arhitektura postaje dominantan trend u *Cloud* okruženju. Ovaj pristup omogućava kompanijama da se fokusiraju na razvoj poslovne logike, dok infrastruktura i upravljanje resursima ostaju u nadležnosti *Cloud* platforme.

Rad istražuje proces migracije mikroservisne aplikacije na bezserversku arhitekturu, fokusirajući se na tehničke izazove, najbolje prakse i korake neophodne za uspešnu implementaciju. Migrirana aplikacija prvobitno je razvijena kao *Freelance* platforma za povezivanje slobodnih radnika i poslodavaca, omogućavajući kreiranje i upravljanje projektima, kao i praćenje toka rada.

Rešenje je zasnovano na korišćenjem *Amazon Web Services* (*AWS*) platforme, s posebnim fokusom na *AWS Lambda* funkcije koje omogućavaju izvršavanje bezserverskog koda u oblaku. Ove funkcije su razvijene u *.NET* ekosistemu, koristeći *C#* programski jezik, čime je obezbeđena kompatibilnost sa postojećom bazom koda originalne aplikacije.

Jedinstvenost ovakvog rešenja ogleda se u sposobnosti da iskoristi prednosti bezserverske infrastrukture uz zadržavanje glavnih osobina mikroservisne arhitekture.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Miroslav Zarić, redovni profesor

2. KORIŠĆENE TEHNOLOGIJE

Kao što je već pomenuto, rešenje se zasniva na korišćenju *AWS* platforme i njenih servisa. U ovom poglavlju biće opisani ključni servisi koji su korišćeni za uspešnu implementaciju aplikacije u bezserverskom okruženju.

2.1. *AWS Lambda*

AWS Lambda [1] je servis koji omogućava pokretanje koda bez potrebe za upravljanjem serverima. Ovaj bezserverski pristup znači da se može pokrenuti kod kao odgovor na događaje kao što su promene podataka, *HTTP* zahtevi, ili aktivnosti u drugim *AWS* servisima. *Lambda* automatski skalira aplikaciju pokretanjem koda kao odgovor na svaki okidač, bez obzira na obim saobraćaja.

2.2. *AWS DynamoDB*

DynamoDB [2] je brza, fleksibilna *NoSQL* baza podataka u potpunosti upravljana od strane *AWS*-a. Kao bezserverska baza podataka, ona automatski upravlja resursima, skaliranjem i dostupnošću, omogućavajući programerima da se fokusiraju na razvoj aplikacija bez brige o infrastrukturi. *DynamoDB* podržava *DynamoDB Streams*, što omogućava hvatanje i skladištenje svih promena koje se dešavaju u tabeli, uključujući operacije dodavanja, ažuriranja i brisanja podataka.

2.3. *AWS API Gateway*

AWS Api Gateway [3] servis služi za kreiranje, objavljivanje, održavanje i osiguranje *API*-a. *API Gateway* deluje kao "prednja vrata" za aplikacije, omogućavajući im pristup podacima, poslovnoj logici ili funkcionalnostima iz *backend* servisa. Ovaj servis podržava kreiranje *RESTful* i *WebScoket API*-a, omogućavajući i podršku za realnovremenu dvosmernu komunikaciju.

2.4. *AWS Cognito*

AWS Cognito [4] je servis za upravljanje autentifikacijom i autorizacijom korisnika u veb aplikacijama. Ovaj servis omogućava programerima da lako dodaju registraciju, prijavljivanje i pristupne kontrole u aplikacije. Njegova glavna prednost je jednostavna integracija sa ostalim *AWS* servisima radi poboljšanja sigurnosti i upravljanja korisničkim pristupom.

2.5. *AWS CloudWatch*

AWS CloudWatch [5] je servis za nadgledanje i upravljanje resursima na *AWS*-u. Omogućava prikupljanje, praćenje i

analizu metrike, logova i događaja sa različitih *AWS* servisa i aplikacija u realnom vremenu. *CloudWatch* je ključan za upravljanje i optimizaciju *AWS* okruženja, jer omogućava uvid u stanje i performanse.

2.6. *AWS EventBridge*

AWS EventBridge [6] je servis za upravljanje događajima u realnom vremenu, koji omogućava aplikacijama da se lako integrišu i reaguju na događaje iz različitih izvora. *EventBridge* omogućava izgradnju aplikacija koje odmah reaguju na promene u sistemu putem događaja.

3. SPECIFIKACIJA ARHITEKTURE SISTEMA

U specifikaciji fokus je na nefunkcionalnim zahtevima i arhitekturi sistema nakon migracije. Funkcionalni zahtevi same aplikacije nisu posebno razmatrani, jer se radi o prenetom rešenju gde su oni ostali u potpunosti nepromenjeni.

3.1. Nefunkcionalni zahtevi

U implementaciji nefunkcionalni zahtevi su od ključnog značaja za efikasnost i pouzdanost sistema. Fokus je stavljen na sledeće aspekte:

- Migracija na bezserversku arhitekturu putem *AWS* servisa, omogućavajući skalabilnost i minimiziranje infrastrukturnog održavanja
- *DevOps Pipeline* za *ASP.NET* servis koji ostaje van bezserverskog okruženja, uz automatski *CI-CD* proces za izgradnju, testiranje i isporuku
- Monitoring i logovanje preko *AWS CloudWatch* servisa, obezbeđujući uvid u performanse, stabilnost i sigurnost sistema

3.1. Arhitektura sistema

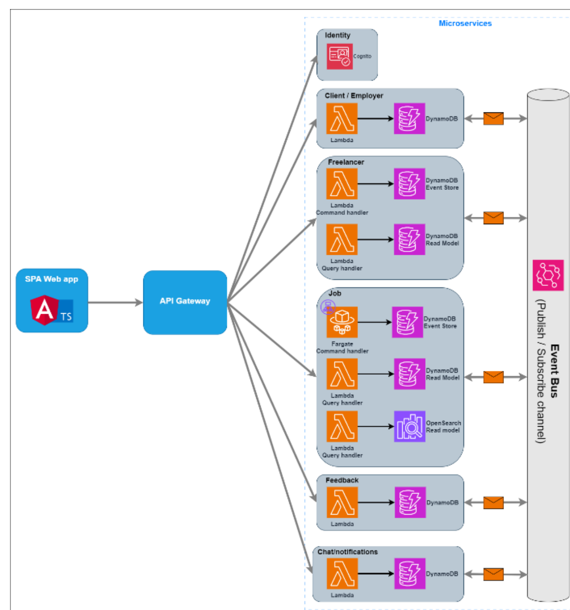
Posmatrajući arhitekturu sistema nakon migracije, ključna razlika u odnosu na originalno rešenje leži u prelasku na bezserversku infrastrukturu. Iako su funkcionalnosti postojećih komponenti ostale nepromenjene, sistem sada koristi *AWS* servise za efikasnije upravljanje resursima i skalabilnost. Na slici 1. prikazana je arhitektura nakon migracije, a nakon slike sledi opis ključnih komponenti sistema.

Komponenta „Identitet i pristup“ (*Identity*) predstavlja centralnu komponentu za upravljanje identitetom i pravom pristupa korisnika i igra ključnu ulogu u osiguravanju autentikacije i autorizacije sistema.

Komponenta „Profil klijenta“ (*Client/Employer*) zadužena je za upravljanje profilima klijenata.

Komponenta „Profil slobodnog radnika“ (*Freelancer*) ima ulogu upravljanja profilima slobodnih radnika unutar sistema. Ova komponenta je odgovorna za igradnju i upravljanje ličnim profilima slobodnih radnika, omogućavajući manipulaciju svim elementima profila.

Komponenta „Posao“ (*Job*) ima za cilj upravljanje celokupnim životnim ciklusom posla na Freelance platformi. Ova komponenta omogućava klijentima da objavljuju poslove i slobodnim radnicima da šalju predloge, što dalje može rezultovati uspešno sklopljenom poslu. Iz navedenog može se zaključiti da ova komponenta ima centralnu ulogu u povezivanju slobodnih radnika i klijenata.



Slika 1. Arhitektura sistema nakon migracije

Komponenta „Povratna informacija“ (*Feedback*) posvećena je dobijanju i deljenju ocena i komentara između klijenata i slobodnih radnika. Ova komponenta omogućava učesnicima obostrano učenje i podelu iskustva.

Komponenta „Notifikacije i chat“ (*Chat/Notifications*) omogućava komunikaciju (između klijenata i slobodnih radnika) i notifikacije u realnom vremenu. Chat funkcionalnost omogućava direktan dijalog i razmenu informacija, dok notifikacije informišu korisnike o važnim događajima na platformi.

EventBus je komponenta koja omogućava komunikaciju između različitih komponenti sistema putem definisanih integracionih događaja. Koristi se za prosleđivanje informacija o događajima bez direktne sprege između komponenti.

API Gataway je komponenta koja predstavlja sloj koji pruža jednostavan i ujedinjen interfejs za klijentske aplikacije. Ona obezbeđuje mapiranje i agregaciju podataka sa više različitih servisa, čime se smanjuje broj zahteva klijenta ka serverskoj strani i olakšava upravljanje.

SPA veb aplikacija je komponenta zadužena za interakciju sa korisnikom.

3. IMPLEMENTACIJA

U ovom poglavlju biće opisana implementacija ključnih servisa *Freelance* platforme nakon migracija na bezserversku infrastrukturu. U opisima implementacija fokus će biti na tehnološkim promenama. Takođe, biće objašnjena interna komunikacija servisa korišćenjem *AWS EventBridge*-a, a na kraju će biti predstavljeno rešenje za monitoring i upravljanje logovima pomoću *AWS CloudWatch*-a.

3.1. Mikroservisi

Tokom procesa migracije, mikroservisi su preneti sa *ASP.NET* radnog okvira na *AWS Lambda* servise, uz korišćenje *DynamoDB*-a za skladištenje podataka. Svaki od mikroservisa zadržao je svoje osnovne funkcionalnosti, ali je u okviru nove arhitekture prilagođen za rad u bezserverskom okruženju.

3.1.1 Identitet i pravo pristupa

U novoj arhitekturi sistema, servis za identitet i pravo pristupa migriran je na *AWS Cognito* servis.

Cognito obezbeđuje gotovo rešenje za upravljanje korisnicima, autentifikaciju i autorizaciju. Role u sistemu sada su definisane kroz *Cognito* grupe, koje omogućavaju razlikovanje između slobodnih radnika i poslodavaca, putem grupa *Freelancer* i *Employer*.

Za prijavu korisnika koristi se *Cognito Hosted UI*, koji nudi unapred pripremljen interfejs za prijavu na sistem, dok je registracija implementirana kao *AWS Lambda* funkcija. Ova *Lambda* funkcija ima zadatak da registruje korisnika u *Cognito* sistemu i doda ga u određenu grupu. Takođe, kreira i domenskog korisnika, bilo da je reč o slobodnom radniku ili poslodavcu.

3.1.2 Upravljanje profilima slobodnih radnika

Originalna implementacija mikroservisa bila je bazirana na *Clean* arhitekturi [7], koja je značajno olakšala proces migracije. *Clean* arhitektura podrazumeva jasno odvajanje logike sistema u nezavisne slojeve, što je omogućilo da se tokom migracije fokusiramo uglavnom na prezentacioni sloj. Konkretno, bilo je potrebno migrirati kod iz svakog kontrolera u *Lambda* funkciju, čime je obezbeđena besprekorna tranzicija ka bezserverskom okruženju.

Pored toga, u mikroservisu već je bio primenjen *CQRS* [8] (*Command Query Responsibility Segregation*) šablon, koji omogućava odvajanje operacija čitanja i pisanja u različite module. Primena *CQRS* šablona u sistemu omogućila je odvajanje modela pisanja i čitanja. Model pisanja predstavlja niz događaja koji se čuvaju u *DynamoDB* tabeli koja predstavlja skadište događaja (engl. *event store*). Za svaki zapisani događaj, podešen je *DynamoDB Stream* koji prati promene u tabeli i aktivira određene *Lambda* funkcije kada dođe do promene. Model čitanja predstavlja agregirane podatke o slobodnim radnicima, koji se čuvaju u posebnoj *DynamoDB* tabeli.

Ovaj *DynamoDB Stream* mehanizam omogućava automatsko obrađivanje događaja i njihovo korišćenje za sinhronizaciju podataka između modela čitanja i pisanja, čime se obezbeđuje konzistentnost i ažuriranost sistema.

3.1.3 Upravljanje poslovima

Poput prethodno opisanog servisa za upravljanje profilima slobodnih radnika, i ovaj servis je zasnovan na čistoj arhitekturi (*Clean Architecture*) i koristi *CQRS* šablon, ali je ovde primenjeno hibridno rešenje. Dok su upiti preobraženi u bezserversko okruženje, komande su zadržane u originalnom *ASP.NET* servisu.

Ovde fokus neće biti na *Lambda* funkcijama koje obrađuju model čitanja, jer im je implementacija identična kao u prethodnom servisu. Umesto toga, pažnja će biti usmerena na dva ključna aspekta: *DevOps* ciklus za *ASP.NET* servis i proces *deploy*-a ovog servisa na *AWS* infrastrukturu.

Prvi *DevOps* ciklus automatski proverava izgradnju (engl. *build*), prolaznost testova i *SonarCloud* [9] skeniranje koda koristeći *GitHub* akcije [10]. Drugi *DevOps* ciklus, koji se inicira ručno, radi *deploy* mikroservisa na *AWS* infrastrukturu.

Osnovu infrastrukture čini *AWS ECS* [11] koji omogućava upravljanje *Docker* [12] kontejnerima. U ovom slučaju, koristi se *AWS Fargate* [13], koji omogućava da se kontejneri izvršavaju bez potrebe za direktnim upravljanjem serverima. *Fargate* samostalno određuje i skalira potrebne resurse za svaki task, što značajno pojednostavljuje upravljanje infrastrukturom. Svaki task u *ECS*-u je definisan putem *ECS task definition*-a, koji opisuje sve potrebne parametre za pokretanje kontejnera, uključujući *Docker* sliku, mrežne portove, resurse kao što su memorija i procesor, i ostale važne konfiguracije.

Ovaj servis dobio je novu funkcionalnost koja omogućava korisnicima naprednu pretragu poslova koristeći *AWS OpenSearch* [15]. Ova pretraga koristi *OpenSearch* kao skladište podataka, dok se ažuriranje podataka automatizuje putem *Lambda* funkcija koje se aktiviraju na osnovu događaja poput kreiranja, ažuriranja ili brisanja poslova u modelu pisanja. Prilikom zahteva za pretragu, *Lambda* funkcija šalje upit *OpenSearch*-u i vraća relevantne rezultate, čime je omogućen brz i efikasan pristup poslovima u sistemu.

3.1.4 *Chat* i notifikacije

Implementacija *chat* sistema i notifikacija zahtevala je rešenje koje omogućava komunikaciju u realnom vremenu. S obzirom na potrebu za migracijom ovih servisa u bezserversko okruženje, kao idealno rešenje izabran je *AWS Api Gateway* u kombinaciji sa *WebSocket* protokolom.

U okviru *WebSocket API Gateway*-a, postoje podrazumevane rute *$connect* i *$disconnect* koje se automatski aktiviraju pri priključenju i prekidu konekcije korisnika. U slučaju ovog servisa, kada se korisnik priključi na *WebSocket*, ruta *$connect* aktivira *Lambda* funkciju koja čuva *connectionId* i *sub* (*JWT* token) korisnika u *DynamoDB* tabeli. Pri prekidu konekcije, ruta *$disconnect* aktivira funkciju koja uklanja ove podatke iz iste tabele. Ruta *sendMessage*, koja je povezana sa odgovarajućom *Lambda* funkcijom, omogućava slanje poruka izmedju korisnika uz pomoć podataka koji se čuvaju u prethodno pomenutoj tabeli.

3.1.5 Povratne informacije

Servis "Povratne informacije" predstavlja relativno jednostavan mikroservis. U procesu migracije u bezserversko okruženje, njegova logika je podeljena na više komandi, pri čemu je svaka komanda povezana sa odgovarajućom *Lambda* funkcijom. Podaci o povratnim informacijama čuvaju se u posebnoj *DynamoDB* tabeli, namenjenoj isključivo za tu svrhu.

Ključni aspekt servisa je obrada domenskog događaja završetka ugovora, koji okida *Lambda* funkciju koja kreira zapis u *DynamoDB* tabeli kako bi pripremila prostor za budući unos povratnih informacija.

3.2 Komunikacija između servisa

Nakon migracije sistema u bezserversko okruženje, pitanje komunikacije između mikroservisa ostalo je podjednako važno kao i u originalnoj arhitekturi.

3.2.1 *AWS EventBridge*

U procesu migracije, za međusobnu komunikaciju putem događaja korišćen je *AWS EventBridge.* U ovom sistemu, događaji stižu iz *DynamoDB Stream*-ova, ali pre nego što budu poslati na *EventBridge*, prolaze kroz posredničku *Lambda* funkciju. Ova funkcija, koja deluje kao dispečer, transformiše događaje u praktičnu strukturu, budući da su izvorni događaji iz *DynamoDB Stream*-a često neprikladni za direktnu obradu u drugim servisima. Tek nakon te transformacije, događaj se šalje na *EventBridge*, gde ga drugi servisi mogu presresti i obraditi.

3.2.2 *AWS API Gateway*

AWS API Gateway služi kao centralna tačka za upravljanje i usmeravanje *HTTP* zahteva u bezserverskom okruženju. U ovoj arhitekturi, *API Gateway* definiše sve rute koje vode do odgovarajućih *Lambda* funkcija, gde svaka funkcija obrađuje konkretan *HTTP* zahtev. Ovaj šablon obezbeđuje fleksibilnost i kontrolu nad komunikacijom između klijentskih aplikacija i mikroservisa, omogućavajući lako upravljanje i monitoring *API* poziva.

U originalnoj *ASP.NET* aplikaciji, *API Gateway* nije služio samo kao posrednik između klijenta i servisa, već je takođe igrao ulogu *API Composer*-a. To je značilo da je *API Gateway*, za neke rute, agregirao podatke iz više servisa i dostavljao ih klijentima kao jedinstven odgovor. Ova funkcija u bezserverskom okruženju implementirana je kroz specijalizovane *Lambda* funkcije. Ove *Lambda* funkcije su odgovorne za skupljanje i agregiranje podataka iz različitih mikroservisa.

3.3 Monitoring – *AWS CloudWatch*

U sklopu migracije, *AWS CloudWatch* se pokazao kao ključni alat za monitoring. Sa svim servisima koji su raspoređeni na *AWS*-u, *CloudWatch* sam po sebi obezbeđuje sveobuhvatno rešenje za praćenje performansi i analizu metrika.

Za svaki od korišćenih *AWS* servisa – kao što su *API Gateway*, *Cognito*, *DynamoDB* i *Lambda* funkcije – *CloudWatch* automatski generiše specifične panele koji predstavljaju vitalne metrike. Ovo je znatno ubrzalo dobijenje uvida u zdravlje i performanse sistema bez potrebe za dodatnim konfiguracijama.

4. ZAKLJUČAK

U ovom radu analiziran je proces migracije postojećeg sistema sa mikroservisne arhitekture na bezserversku infrastrukturu, sa ciljem povećanja skalabilnosti, smanjenja troškova i optimizacije upravljanja resursima. Mikroservisna arhitektura, iako veoma efikasna u radu sa kompleksnim sistemima, sa porastom broja korisnika donosi izazove u održavanju infrastrukture i upravljanju resursima. Ovi izazovi doveli su do potrebe za prelaskom na fleksibilniju arhitekturu koja omogućava lakše prilagođavanje rastućim zahtevima tržišta.

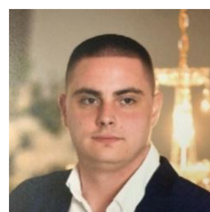
Migracija na bezserversku infrastrukturu baziranu na *AWS* uslugama, kao što su *Lambda* funkcije, *API Gateway* i *DynamoDB*, omogućila je automatsko skaliranje resursa i značajno smanjenje operativnih troškova. Ova arhitektura je pojednostavila upravljanje sistemom, jer *Lambda* funkcije reaguju isključivo na događaje, što eliminiše potrebu za stalnim upravljanjem serverima.

Postavljanjem infrastrukture na *AWS* otvorene su mogućnosti za dalje unapređenje sistema kroz integraciju sa drugim *AWS* uslugama. Na primer, primena *AWS OpenSearch*-a omogućila bi obradu velikih količina podataka u realnom vremenu, dok bi korišćenje mašinskog učenja i veštačke inteligencije moglo doprineti poboljšanju performansi i personalizaciji usluga kroz analizu podataka i predviđanje korisničkih potreba.

5. LITERATURA

[1] *AWS Lambda* servis za pokretanje koda u oblaku u bezserverskom okruženju https://aws.amazon.com/lambda/
[2] *AWS DynamoDB* bezserverska *NoSQL* baza podataka https://aws.amazon.com/dynamodb/
[3] *AWS API Gateway* servis za kreiranje ulazne tačke za aplikacije https://aws.amazon.com/api-gateway/
[4] *AWS Cognito* servis za autentifikaciju i autorizaciju korisnika https://aws.amazon.com/cognito/
[5] *AWS CloudWatch* servis za nadgledanje i upravljanje resursima https://aws.amazon.com/cloudwatch/
[6] *AWS EventBridge* servis za upravljanje dogadjajima u realnom vremenu https://aws.amazon.com/eventbridge/
[7] *Clean* arhitektura https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
[8] *CQRS* šablon https://martinfowler.com/bliki/CQRS.html?ref=blog.funda.nl
[9] *SonarCloud Scan https://github.com/marketplace/actions/sonarcloud-scan*
[10] *GitHub Actions* https://github.com/features/actions
[11] *AWS ECS* servis za orkestraciju kontejnera https://aws.amazon.com/ecs/
[12] *Docker https://www.docker.com/*
[13] *AWS Fargate* servis za beserversko pokretanje kontejnera https://aws.amazon.com/fargate/
[15] *AWS OpenSearch* servis za pretraživanje i analizu velikih količina podataka https://aws.amazon.com/opensearch-service/

Kratka biografija:



Marko Rapić rođen je 26.02.2000. godine u Novom Sadu. Godine 2019. upisao je Fakultet Tehničkih Nauka u Novom sadu, odsek Računarstvo i automatika. Osnovne studije završio je u septembru 2023. Od oktobra 2023. upisuje Master akademske studije.
kontakt: rapic.marko26@gmail.com