UNIVERZITET CRNE GORE
ELEKTROTEHNIČKI FAKULTET

Nikola Ostojić

PRIMJENA KONCEPTA MIKROSERVISNE ARHITEKTURE I KONTEJNERIZACIJE U WEB APLIKACIJI ZA PLASMAN POLJOPRIVREDNIH PROIZVODA

-MASTER RAD-

Podgorica, 2024. godine

UNIVERZITET CRNE GORE
ELEKTROTEHNIČKI FAKULTET

Nikola Ostojić

PRIMJENA KONCEPTA MIKROSERVISNE ARHITEKTURE I KONTEJNERIZACIJE U WEB APLIKACIJI ZA PLASMAN POLJOPRIVREDNIH PROIZVODA

-MASTER RAD-

Podgorica, 2024. godine

PODACI I INFORMACIJE O STUDENTU

Ime i prezime:	**Nikola Ostojić**
Datum i mjesto rođenja:	17.12.1998. godine, Podgorica
Naziv završenog osnovnog studijskog programa i godina završetka studija:	Studijski program Primijenjenog računarstva, Elektrotehnički fakultet, Univerzitet Crne Gore, 180 ECTS kredita, 2020. godine.

INFORMACIJE O MASTER RADU

Naziv master studija:	Postdiplomske master studije primijenjenog računarstva
Naslov rada:	Primjena koncepta mikroservisne arhitekture i kontejnerizacije u web aplikaciji za plasman poljoprivrednih proizvoda
Fakultet na kojem je rad odbranjen:	Elektrotehnički fakultet

UDK, OCJENA I ODBRANA MASTER RADA

Datum prijave magistarskog rada:	15.12.2023. godine
Datum sjednice Vijeća na kojoj je prihvaćena tema:	08.02.2024. godine
Komisija za ocjenu/odbranu rada:	Prof. dr Milutin Radonjić, predsjednik Prof. dr Nikola Žarić, mentor Doc. dr Nevena Radović, član
Mentor:	Prof. dr Nikola Žarić
Datum odbrane:	_____

Ime i prezime autora: Nikola Ostojić, BApp

ETIČKA IZJAVA

U skladu sa članom 22 Zakona o akademskom integritetu i članom 18 Pravila studiranja na master studijama, pod krivičnom i materijalnom odgovornošću, izjavljujem da je master rad pod naslovom

"Primjena koncepta mikroservisne arhitekture i kontejnerizacije u web aplikaciji za plasman poljoprivrednih proizvoda"

moje originalno djelo.

U Podgorici, Novembra 2024. godine.

Podnosilac izjave,

NOstojić

Nikola Ostojić, BApp

PREDGOVOR

Ovim radom želim da izrazim duboku zahvalnost svima koji su me podržavali i inspirisali tokom mog školovanja i istraživanja. Najprije, želim da posvetim ovaj rad svojoj porodici, naročito roditeljima, sestrama i bratu, čija je podrška bila neprocenjiva tokom svih mojih napora i uspjeha. Posebnu zahvalnost želim da iskažem svojim kolegama - prijateljima iz kompanije u kojoj radim, čiji su savjeti, iskustvo i podrška bili od presudnog značaja u realizaciji ovog projekta. Takođe, želim da se zahvalim svom mentoru, Prof. dr Nikoli Žariću, čiji su stručnost i vođstvo bili ključni u odabiru teme i usmjeravanju mog istraživanja. Na kraju, želim da se zahvalim svima koji su na bilo koji način doprinijeli mom školovanju i profesionalnom razvoju.

Nikola Ostojić, Bapp

IZVOD RADA

U današnjem digitalnom okruženju, upotreba informacionih tehnologija postaje sve zastupljenija, posebno u sektorima kao što je poljoprivreda, konkretno plasman poljoprivrednih proizvoda. Ovaj rad istražuje primjenu koncepta mikroservisne arhitekture i kontejnerizacije u razvoju web aplikacije za plasman poljoprivrednih proizvoda.

U teorijskom dijelu, obrađuje se koncepte mikroservisne arhitekture i kontejnerizacije, ističući njihove prednosti u pogledu skalabilnosti, fleksibilnosti i izolacije resursa.

U praktičnom dijelu, koristi se kombinacija mikroservisa i kontejnera kako bi se implementirala aplikaciju za plasman poljoprivrednih proizvoda. Kroz ovaj proces, istražuje se kako ovi koncepti olakšavaju razvoj, implementaciju i održavanje aplikacije. Detaljno je objašnjeno kako se kontejnerizacija primjenjuje na samu aplikaciju, omogućavajući laku distribuciju i upravljanje aplikacijom dok istovremeno omogućavaju efikasnu upotrebu resursa.

Na osnovu naših istraživanja, zaključujemo da primjena mikroservisne arhitekture i kontejnerizacije može značajno poboljšati performanse, skalabilnost i bezbjednost web aplikacija za plasman poljoprivrednih proizvoda. Ovaj rad pruža osnovu za dalje istraživanje i implementaciju ovih tehnologija u sličnim kontekstima.

Ključne riječi: mikroservisna arhitektura, kontejnerizacija, web aplikacija, skalabilnost, bezbjednost.

ABSTRACT

In today's digital environment, the use of information technologies is becoming increasingly prevalent, particularly in sectors such as agriculture, specifically the sale of agricultural products. This paper explores the application of microservices architecture and containerization in the development of a web application for the sale of agricultural products.

In the theoretical part, we discuss the concepts of microservices architecture and containerization, emphasizing their advantages in terms of scalability, flexibility, and resource isolation.

In the practical part, we use a combination of microservices and containers to implement an application for the marketing of agricultural products. Through this process, we explore how these concepts facilitate the development, implementation, and maintenance of the application. We explain in detail how containerization is applied to the application itself, enabling easy distribution and management of the application while efficiently utilizing resources.

Based on our research, we conclude that the application of microservices architecture and containerization can significantly improve the performance, scalability and security of web applications for the marketing of agricultural products. This paper provides a basis for further research and implementation of these technologies in similar contexts.

Key words: microservices architecture, containerization, web application, scalability, security.

Sadržaj:

1. UVOD

Primjena koncepta mikroservisne arhitekture i kontejnerizacije u web aplikacijama postaje sve prisutnija u svijetu razvoja softvera zbog svoje efikasnosti. Mikroservisna arhitektura omogućava razdvajanje funkcionalnosti web aplikacije u manje, autonomne servise, što olakšava razvoj, testiranje i održavanje sistema. Svaki mikroservis može biti razvijen, testiran i implementiran nezavisno od drugih servisa, što omogućava timovima da rade paralelno i brže dostavljaju funkcionalnosti. Kontejnerizacija dodatno pojačava ovu fleksibilnost omogućavajući smiještanje svakog mikroservisa i njegovih zavisnosti u zaseban, izolovani kontejner. Ovo omogućava konzistentno okruženje između razvoja, testiranja i produkcije, što olakšava prenosivost aplikacija između različitih okruženja i smanjuje mogućnost pojave problema usled razlika u konfiguraciji ili okruženju.

Kroz primjenu mikroservisne arhitekture i kontejnerizacije, web aplikacije postaju agilnije i pouzdanije. Svaki mikroservis može biti skaliran nezavisno od drugih, što omogućava efikasno upravljanje resursima i optimalno korištenje infrastrukture. Takođe, kontejneri pružaju izolaciju između servisa, čime se smanjuje rizik da prekid ili oštećenje jednog servisa imaju veći uticaj na ostale servise. Osim toga, kontejneri omogućavaju brzo i efikasno upravljanje resursima, automatsko otkrivanje i pokretanje novih instanci servisa prema potrebi, što je posebno korisno u situacijama sa promjenjivim opterećenjem ili potrebom za visokom dostupnošću. Sve ove karakteristike čine mikroservisnu arhitekturu i kontejnerizaciju idealnim izborom za razvoj modernih, skalabilnih i sigurnih web aplikacija.

Predmet istraživanja ovog rada je analiza primjene mikroservisne arhitekture u kontekstu web aplikacije za plasman poljoprivrednih proizvoda. Mikroservisna arhitektura pruža modularni pristup razvoju aplikacije, omogućavajući razdvajanje funkcionalnosti u manje, autonomne servise. Ovaj pristup olakšava razvoj, testiranje i održavanje sistema, jer svaki mikroservis može biti nezavisno razvijen i implementiran. U kontekstu aplikacije za plasman poljoprivrednih proizvoda, mikroservisi mogu obuhvatiti funkcionalnosti kao što su upravljanje proizvodima, narudžbinama, plaćanjima i dostavom, čime se omogućava fleksibilnost i prilagodljivost sistema potrebama korisnika i poslovnih zahtjeva. U ovom radu, mikroservisi su dizajnirani za kreiranje oglasa, omogućavajući poljoprivrednicima da promovišu svoje proizvode. Jedan mikroservis je namijenjen korisnicima aplikacije (poljoprivrednicima), omogućavajući im da objave oglase za svoje proizvode.

Drugi mikroservis je namijenjen potrošačima, tj. korisnicima koji će moći da naruče proizvode putem oglasa koje su kreirali poljoprivrednici.

Fokus istraživanja je na proučavanju efikasnosti kontejnerizacije u razvoju web aplikacija, u ovom radu koristimo aplikacije za plasman poljoprivrednih proizvoda. Kontejnerizacija, kroz alate poput Docker-a, omogućava smiještanje mikroservisa i njihovih zavisnosti u izolovane kontejnere. Ovaj pristup pojednostavljuje upravljanje mikroservisima, osigurava konzistentno okruženje između razvoja, testiranja i produkcije, te olakšava skaliranje i distribuciju aplikacije. U radu je analizirano kako kontejnerizacija doprinosi agilnosti i pouzdanosti web aplikacije.

Mikroservisna arhitektura omogućava kreiranje modularne aplikacije koja se sastoji od nekoliko specijalizovanih mikroservisa, svaki odgovoran za određenu funkcionalnost. Kontejnerizacija olakšava upravljanje ovim mikroservisima, omogućavajući brzo pokretanje, skaliranje i migraciju kontejnera između različitih okruženja. Ova kombinacija tehnologija omogućava brži razvoj aplikacije, lakšu integraciju novih funkcionalnosti i efikasnije ispunjavanje zahtjeva korisnika.

Istraživačka pitanja koja su razmatrana u ovom master radu su:

1. Kako odabir odgovarajućih programskih jezika za svaki mikroservis utiče na skalabilnost, efikasnost i responzivnost web aplikacije za plasman poljoprivrednih proizvoda?
    - Ovo istraživačko pitanje istražuje ključni uticaj na to kako odabir jezika za svaki mikroservis ima na performance aplikacije, s naglaskom na brzinu odziva I efikasnost u distribuciji poljoprivrednih proizvoda. Kroz analizu uticaja različitih jezika na karakteristike svakog mikroservisa, istražuje se kako ovi izbori mogu direktno uticati na funkcionalnost i performance cjelokupne web aplikacije, pružajući smjernice za optimizaciju izbora jezika radi postizanja najboljih performansi u ovom specifičnom domenu.

2. Analiza prednosti i mana mikroservisne arhitekture u različitim softverskim okruženjima.
    - Ovo istraživačko pitanje istražuje mogućnosti optimizacije performansi, održivosti i drugih ključnih karakteristika softverske arhitekture kroz primjenu mikroservisne arhitekture. Fokusira se na analizu uticaja mikroservisa na brzinu izvođenja aplikacija, poboljšanje održivosti softvera kroz efikasno upravljanje greškama, kao i

na izazove i prednosti u vezi s aspektima sigurnosti i zaštite podataka. Takođe su izvršena poređenje prednosti i mana u slučajevima gdje se koristi ova arhitektura.

3. Koje su prednosti i izazovi kontejnerizacije u olakšavanju implementacije mikroservisne arhitekture?
    - Ovo istraživačko pitanje fokusira se na identifikaciju ključnih koristi i potencijalnih prepreka prilikom primjene kontejnerizacije za podršku mikroservisima. Kroz analizu, istražuje se kako kontejneri olakšavaju razvoj, testiranje, implementaciju i skaliranje mikroservisa. Ovaj rad pruža detaljan uvid u prednosti, ali i ograničenja kontejnerizacije prilikom podrške mikroservisima, nudeći smjernice za optimalno iskorištenje ovih tehnologija u izgradnji stabilnih aplikacija.

2. MIKROSERVISI

2.1. Nastanak i razvoj mikroservisa

Prije desetak godina, grupa softverskih arhitekata se okupila i definisala pojam mikroservis kako bi mogao da se definiše stil arhitekture softvera koji je vremenom sve više evoluirao.[1] Kod tradicionalnih monolitnih aplikacija, u jednoj aplikaciji su enkapsulirane sve funkcije. Navedeni pristup može da ima veliki broj prednosti, a tu se prvenstveno misli na testiranje i brži razvoj, laku primjenu, ali samo kada je relativno manja aplikacija. Međutim, moderni softverski sistemi postali su dosta složeni i navedeni arhitekturni stil je doveo do određenih slabosti, a tu se misli na i lošu pouzdanost.

U poslednjih nekoliko godina, mikroservisi postaju široko prihvaćena paradigma u razvoju softverskih sistema, a veliki broj firmi, PayPal, Amazon, X i ostale, počinju da implementiraju mikroservise u *cloud-u* ili su obavile prelaz sa zastarjele monolitne na mikroservisnu arhitekturu. Kod modernih softverskih sistema, potrebna je visoka dostupnost, visoka konkurentnost, niska povezanost, visoka skalabilnost i visoka kohezija. Tokom 2011. godine, prvi put je uveden pojam „mikro usluge“ u arhitekturi kao način uz pomoću kojeg mogu da se opišu zajedničke ideje učesnika u obrascima softverske arhitekture.[2] Mikroservisi jesu arhitekturni stil u razvijanju modernih softverskih aplikacija. Sam stil se zasniva na razgrađivanju monolitne aplikacije na slabije povezane i posebne mikroservise, koji na nezavisan način mogu da se testiraju, razvijaju, raspoređuju i proširuju.

Kod mikroservisa mogu da se izdvoje naredne karakteristike:[3]

- Nezavisna primjena;
- Nezavisan razvoj;
- Visoka konkurentnost;
- Nezavisna realizacija;
- Niska povezanost;
- Visoka dostupnost;
- Visoka kohezija;

[1] Ronnie, Mitra; Nadareishvilli, Irakli: *Microservices: Up and Running: A Step-by-Step Guide to Building a Microservices Architecture*, O'Reilly Media, New York, 2020, p. 2.
[2] Safina, Larisa; Mazzara, Manuel; Montesi, Fabrizio; Rivera. Victor: *Data-driven workflows for microservices (genericity in jolie)*, AINA, Crans – Montana, 2016, p. 430 – 437.
[3] Gross, Hans; Melideo, Matteo; Sillitti, Alberto: Self certification and trust in component procurement, *Journal of Science of Computer Programming*, 2005, pp. 141 – 156.

- Visoka skalabilnost.

Mikroservisi su malo složeniji jer funkcionalno razlažu veće sisteme u skup nezavisnih usluga.[4] Oni razmjenjuju podatke i komuniciraju preko jednostavnog HTTP (*Hypertext Transfer Protocol - Protokol za prenos hiperteksta*) protokola, ali i mehanizama kao što su magistrale za poruke ili REST API (*Representational State Transfer Application Programming Interface - Interfejs za programsko povezivanje zasnovan na prenosu reprezentativnog stanja*). Veoma često tokom razvijanja softvera, programeri neće koristi obrasce (*Pattern*) softverske arhitekture i tada će doći do neorganizovanog koda koji neće imati jasnu ulogu i tokom vremena taj kod će postati veoma težak za održavanje. Ukoliko se prati paradigma mikroservisa, sistem je strukturiran sastavljanjem manjih nezavisnih građevinskih blokova koji će isključivo komunicirati pomoću poruke.[5]

Sve komponente koje se u aplikacijama razvijaju bez unaprijed isplanirane arhitekture, često će biti čvrsto povezane, i biće izuzetno teške za određenu promjenu. U navedenim aplikacijama, teško se primjenjuje određeni arhitekturni obrazac, bez razumijevanja funkcionalnosti svih modula i komponenti sistema, te je neophodno upotrebljavati obrasce softverske arhitekture za vrijeme razvoja određene aplikacije. Softverska arhitektura predstavlja skup odluka u organizaciji softvera. Svaka struktura, sastoji se od elemenata softvera, ali i njihovih međusobnih odnosa. Podaci o arhitekturi softvera će pomoći prilikom definisanja osnova za ponašanje određene aplikacije. Prisutno je nekoliko različitih tipova softverskih arhitektonskih obrazaca, ali jedna od najbitnijih jeste mikroservisna arhitektura. Ova arhitektura počela je da evoluira na razvoju monolitnih aplikacija koje upotrebljavaju obrazac za razvijanje distribuiranih aplikacija, tj. slojevite arhitekture kroz SOA (*Service–Oriented Architectures - Servisno orijentisane arhitekture*) obrazac. Mikroservisni arhitektonski obrazac predstavlja obrazac koji se upotrebljava kod aplikacija koje su napravljene u SOA ili monolitnoj arhitekturi.[6] Svaka servisna komponenta sadrži individualne module, tj. mikroservise kao zasebne jedinice, a na ovaj način će se omogućiti brže postavljanje cijele aplikacije, što će dovesti do nivoa primjenljivosti.

Na Slici 1. prikazana je klasična monolitna arhitektura softverskog sistema. Ova organizacija uključuje nekoliko slojeva, počevši od korisničkog interfejsa (prezentacionog sloja), preko poslovne

[4]Dragoni, Nicola; Giallorenzo, Saverio; Lluch – Lafuente, Alberto; Mazzara, Manuel: *Microservices – Yesterday, today and tomorrow*, Springer, New York, 2017, p. 195 – 2016.
[5]Pardon, Guy; Buijze Allard; Dustdar, Schahram: *Practical Microservices Architectural Patterns*, Apress, Kerala, 2019, p. 31.
[6] Sillitti, Alberto; Vernazza, Tullio; Succi, Giancarlo: *Service oriented programming: a new paradigm of software reuse*, Springer, Berlin, 2002, p. 269 – 280.

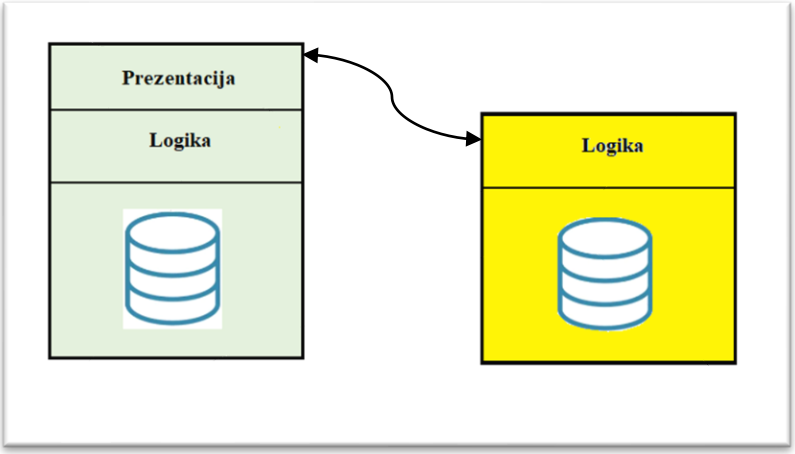
logike, pa sve do sloja za upravljanje podacima (alatima za perzistenciju). Slojevi su jasno odvojeni i predstavljeni po funkcionalnoj odgovornosti. Na Slici 1. je izvršena vertikalna podjela numerisana od 1 do 4, gdje svaki broj predstavlja različit modul unutar sistema. Svaki od ovih modula može imati funkcionalnosti koje se odnose na bilo koji sloj, omogućavajući im da sarađuju kako bi se realizovala određena funkcionalnost. Baza podataka je centralizovana, odnosno zajednička za sve module, i omogućava pristup neophodnim resursima, poput memorije i drugih sistema za skladištenje podataka. Ovakva struktura osigurava međusobnu zavisnost modula, što je karakteristika monolitnih sistema.

Slika 1: *Monolitna arhitektura*[7]

Glavni izazov kod monolitnih aplikacija je njihova sposobnost da evoluiraju i ostanu održive tokom vremena. Najveći problemi se pojavljuju prilikom uvođenja promjena, jer svaka modifikacija može da utiče na cjelokupan sistem. Na Slici 2. je prikazana mikroservisna arhitektura, koja predstavlja alternativu monolitima, gdje su uloge komponenti jasnije definisane i izolovane. U mikroservisnoj arhitekturi, svaka usluga je zasebna jedinica sa sopstvenim alatima za postojanost podataka (bazama podataka), što omogućava veću fleksibilnost u radu. Komunikacija između ovih usluga obavlja se putem prosleđivanja poruka, što olakšava interakciju među njima i minimizuje zavisnost. U poređenju sa monolitnim sistemima, gdje se svaka promjena mora implementirati kroz sve slojeve (od interfejsa do baze podataka), mikroservisna arhitektura omogućava horizontalno skaliranje i nezavisno implementiranje promjena na nivou pojedinačnih servisa. Time se postiže

[7] Mazzara, Manuel; Khanda, Kevin; Mustafin, Ruslan; Rivera, Victor; Safina, Larisa; Sillitti, Alberto: *Microservices Science and Engineering*, Innopolis University, Innopolis, 2017, p. 2.

bolja modularnost, lakše održavanje i fleksibilnija primjena novih funkcionalnosti bez uticaja na cjelokupan sistem.



Slika 2: *Mikroservisna arhitektura*[8]

Složenost razvoja i održavanja aplikacije će se pomjeriti na nivo koordinacije usluga (ovaj termin je često poznat pod imenom orkestracija).[9] Potrebno je da se adresira veći broj dodatnih međuzavisnosti zbog distribuirane prirode navedene vrste pristupa (na primjer, sertifikacija i povjerenje). Nije unaprijed definisano ograničenje veličine koja će definisati da li je servis mikroservis ili nije. Zaista, mikroservis je na određeni način pogrešna definicija. Od svakog mikroservisa se očekuje da implementira neku svoju jedinstvenu poslovnu sposobnost, da nema ograničenu funkcionalnost sistema, da donosi prednosti u pogledu proširenja i održavanja usluga, a kao takav, mikroservis je idealan za velike sisteme.[10] Svaki mikroservis predstavlja jednu poslovnu sposobnost, koja se ažurira i isporučuje nezavisno, bavi se otkrivanjem grešaka ili dodavanjem manjih poboljšanja, imajući određen uticaj na druge usluge, kao i na njihova izdavanja. U uobičajenoj praksi, očekuje se da jedna usluga može da bude razvijena i upravljana od strane jednog tima.

Izuzetno je privlačna ideja da tim radi na jednom mikroservisu: kako bi se izgradio sistem sa slabo povezanim i modularnim dizajnom, mora da se obrati pažnja na organizacionu strukturu, ali i njene komunikacione obrasce, prema Konvejevom zakonu.[11] Ukoliko jedan tim kreira organizaciju i radi na jednom servisu, ovakva struktura će učiniti komunikaciju efikasnijom ne samo na nivou

[8] Mazzara, Manuel; Khanda, Kevin; Mustafin, Ruslan; Rivera, Victor; Safina, Larisa; Sillitti, Alberto: *Microservices Science and Engineering*, Innopolis University, Innopolis, 2017, p. 3.
[9] Mazzara, Manuel;.Govoni, Sergio; *A Case Study of Web Services Orchestration*, Springer, Berlin, 2005, p. 1 – 16.
[10] Nadareishvili, Irakli; Ronnie, Mitra; McLarty, Matt; Amundsen, Mike: *Microservice Architecture – Aligning principles, practices and culture*, O'Reilly, Boston, 2016, p. 4.
[11] Conway, Melvin: How do committees invent, *Datamation*, vol. 14, no. 4, 1968, pp. 28–31.

tima, nego i unutar cijele organizacije, poboljšavajući dizajn u smislu modularnosti. Glavna karakteristika mikroservisa jeste da timovi budu mali, a komunikacija će biti efikasnija stvaranjem manjih međufunkcionalnih timova koji su u stanju da kontinuirano rade na istoj usluzi i da u potpunosti budu odgovorni za tu uslugu. Timovi se organizuju oko servisa, koji su organizovani oko poslovnih sposobnosti.

Optimalnu veličinu tima za mikroservis je najbolje opisalo poznato pravilo „dva pica tima“ Džefa Bezosa, koje sugeriše da veličina tima ne bi smjela da bude veća od onoga što dvije pice mogu da hrane. Samo pravilo neće dati tačan broj, ali se najčešće procijenjuje da je poželjno da to bude od šest do osam ljudi. Nedostatak ovakvog pristupa jeste što nije uvijek praktičan sa finansijske strane, jer angažovanje tima programera za jednu uslugu može da dovede do visokih troškova razvoja i održavanja.

Mora se biti oprezan prilikom projektovanja visokog nivoa struktura organizacije koji će upotrebljavati mikroservise – povećanje broja usluga može na negativan način da utiče na cjelokupnu efikasnost organizacije, ukoliko se ne preduzmu određene akcije.

2.2. Razlozi korišćenja mikroservisa

Mikroservisi omogućavaju veliki broj prednosti. Cilj detaljnog razumijevanja izazova i benefita koje nudi mikroservisna arhitektura jeste bolja odluka da li ova vrsta arhitekture predstavlja dobrog kandidata za sistem koji planira da se razvije, s obzirom na njegove karakteristike i zahtjeve. Prednosti mikroservisa mogu da se podijele u nekoliko domena, kao što su:[12]

- Tehnički domen;
- Organizacioni domen;
- Poslovni domen.

Tehnički domen

Kada je riječ o tehničkom domenu, razvojna zajednica se složila oko narednih benefita mikroservisa: tehnološka otpornost, heterogenost, nezavisnost i skalabilnost.

Tehnološka heterogenost (*Techology Heterogeneity*) zasniva se na tome da su mikroservisi na međusoban način razdvojeni, predstavljaju nezavisne komponente koje komuniciraju uz pomoć određenog komunikacionog mehanizma, najčešće protokola kao što je HTTP. Ovo znači da domen organizacije, koji ima obavezu u razvijanju mikroservisne arhitekture, nema obavezu da implementira svu arhitekturu u identičnoj tehnologiji koja će obuhvatiti cjelokupan sistem.

[12] Newman, Sam: *Izgradnja mikrosistema – Dizajn sitno granulisanih sistema*, O'Reilly Media, Boston, 2015, p. 5.

Mikroservise krasi prednost da zbog njihove međusobne nezavisnosti, svaki individualni mikroservis može da se implementira u različitoj tehnologiji. Danas, ne postoji tehnološki „*silver bullet*“ koji će riješiti sve potencijalne probleme i koji će pružiti najbolje moguće performanse, pa iz tog razloga ima smisla da se određene karakteristike sistema optimizuju, tako što će se upotrebljavati tehnologija koja je prikladna za pojedini zadatak. Ovo znači da je moguće da se svaki mikroservis implementira u tehnologijama koje će biti prikladne za njegov zadatak sa minimalnim uticajem na sistem u kojem može da se nađe, pošto će ostati isti protokol komunikacije.[13]

U ovakvoj situaciji, mikroservis može da se posmatra kao crna kutija. Zbog svoje prirode, mirkoservisna arhitektura predstavlja teorijski manje pouzdanu arhitekturu od drugih sličnih arhitektura. Ukoliko se arhitektura proteže kroz veći broj raznih servera, onda je prisutna i opasnost od kvara na hardveru. Kod distribuiranih sistema, prisutna je opasnost od ispada u mreži uz pomoću koje komuniciraju komponente sistema. Zaštitni zid (*Bulkhead*) predstavlja glavni pojam kod robusnosti i otpornosti. U slučaju da ispadne jedna komponenta distribuiranog sistema, ukoliko je na pravilan način uspostavljena pregradna zaštita, onda ispad ne može da se proširi i da na negativan način utiče na rad drugih komponenti u sistemu. Mikroservisi čine prirodnu granicu koja mora da se ponaša kao zaštitni zid ukoliko dođe do ispada ili kada mikroservis detektuje grešku u radu, onda bi uz pomoć komunikacionog protokola morala ponovo da se pošalje standardna vrijednost, koja će se u sistemu tretirati kao kod za grešku.

Nakon detektovanja prestanka rada servisa pomoću koda za grešku od strane sistema, u optimalnom slučaju će na neko vrijeme da se smanji dostupnost usluge. Ovo predstavlja bitnu tehnološku prednost nad monolitnom arhitekturom, kod koje katastrofalna greška često može da znači i pad cijelog sistema. Skalabilnost je takođe jedna od prednosti mikroservisne arhitekture. Kod monolitne arhitekture, kada dođe do potrebe za skaliranjem, neophodno je da se sistem kao cjelina skalira.

Sa druge strane, kod mikroservisne arhitekture se ovaj problem rješava tako što se skalira onaj servis koji je preopterećen zbog porasta, na primjer prometa. Mikroservisi se mogu optimizovati tako da manje zahtjevni servisi budu postavljeni na hardver sa slabijim performansama, dok se kritični i resursno zahtjevni mikroservisi, za koje se očekuje veće opterećenje, implementiraju na hardveru sa boljim performansama. Pored toga, neki mikroservisi mogu se postaviti na različitim lokacijama kako bi bili bliži korisnicima, čime se smanjuje vrijeme odziva i poboljšava kvalitet usluge. Jedna od najvećih prednosti mikroservisne arhitekture je autonomnost –

[13] Sanchez, Carlos; Vilarino, Pablo: *PHP Microservices*, Packt Publishing, Birmingham, p. 11.

mikroservis može biti izolovan na određenoj platformi (*Platform as a Service*) ili kao proces na operativnom sistemu.[14]

U arhitekturi softvera, izolovanost mikroservisa povlači činjenicu da mikroservisi nisu zavisni međusobno, ali zajedno oni sačinjavaju jedan funkcionalan sistem. Međusobna komunikacija se odvija uz pomoć komunikacionih kanala, kojima se poruke oblikuju putem protokola komunikacionog kanala. Mikroservisi mogu da se mijenjaju bez uticaja na ostatak sistema, da mijenjaju ili nadograđuju unutrašnju logiku. Kako bi okolini mogao da bude dostupan mikroservis, on treba da implementira određenu vrstu interfejsa, a u praksi se veoma često upotrebljava API (Application Programming Interface). API interfejst treba da se implementira ukoliko se mikroservisna arhitektura prostire na nekoliko međusobno udaljenih računara.

Organizacioni domen

Mikroservisna arhitektura ne donosi samo tehničke benefite, već i organizacione. Ovdje se posebno može primijeniti Konvejev zakon, koji kaže: *"Sistemi koje organizacije dizajniraju predstavljaju odraz njihovih unutrašnjih komunikacionih struktura."* Drugim riječima, struktura tima direktno utiče na strukturu softverskog sistema – ukoliko je komunikacija među timovima kompleksna, i sistem će odražavati tu složenost.

Ovaj princip ima poseban značaj u mikroservisnoj arhitekturi, jer način na koji su mikroservisi organizovani omogućava da se veliki, kompleksni projekti razdvoje na manje, samostalne module. Time se smanjuje potreba za centralizovanim upravljanjem, što olakšava komunikaciju i pojednostavljuje odlučivanje. Nezavisni timovi, fokusirani na konkretne zadatke, mogu efikasnije da rade i donose odluke na lokalnom nivou, bez složenih lanaca odobravanja. Tako se postiže veća fleksibilnost i autonomija unutar organizacije.

Poslovni domen

Organizacione i tehničke prednosti dovešće zajedno do poslovnih prednosti: rizik poslovnih projekata će se smanjiti, biće manje intenzivna komunikacija i koordinacija unutar i između timova, time će se koncentracija usmjeriti prema realizovanju zahtjeva projekata. Mikroservisna arhitektura će dati značaj poslovnoj prednosti – omogućiće se brži razvoj i smanjiće se rizici projektnih zahtjeva. Paralelni rad koji omogućava mikroservisna arhitektura na direktan način će uticati na poslovni aspekat projekta, u isto vrijeme projekat će rasti i širiti se u svim pravcima. Ukoliko jedan od timova naiđe na probleme, ti problemi će ostati samo na pojedinom mikroservisu dok se ne riješe, dok će se rad na svim drugim aspektima projekta nastaviti kontinuirano. Ukoliko se

[14] Newman, Sam: *Izgradnja mikrosistema – Dizajn sitno granulisanih sistema*, O'Reilly Media, Boston, 2015, p. 120.

arhitektura pažljivo odabere za potrebe projekta koji se razvija, mikroserisi će podržati agilni rad. Ovo je veoma dobar razlog, sa poslovne strane, da se upotrebljava mikroservisna arhitektura.

Izazovi i Rizici

Mikroservisna arhitektura ne predstavlja jedinstveno rješenje koje samo treba da se realizuje i problem će biti eliminisan, a održavanje sistema više neće predstavljati nikakav izazov. Postupak razdvajanja sistema u mikroservisne sisteme izaziva povećanu složenost sistema, zato što razvojni timovi imaju zadatak da razviju distribuirani sistem – testiranje je dosta složenije na distribuiranim sistemima, pa vrlo vjerovatno treba da se implementira određeni komunikacioni mehanizam između servisa.[15] Zatim, složenost sistema može dovesti do izazova na tehničkom nivou, kao što je visoki vremenski odziv komponenti ili kvarovi na pojedinačnim mikroservisima. Isto tako, razdvajanje sistema na mikroservise može stvoriti probleme u integraciji, pa je bitno dobro razumijevanje kako aplikacija može da komunicira sa okolinom.[16] Mogu da se jave i određeni problemi prilikom razdvajanja funkcionalnosti na neki mirkoservis. Jedan od potencijalnih izazova koje mikroservisna arhitektura uvodi jeste nepouzdana komunikacija (*Unreliable Communication*).

Mikroservisi sa okolinom i međusobno komuniciraju uz pomoć mreže, koja može biti nepouzdana i nesigurna. Poruke koje se šalju preko mreže koja nije pouzdana mogu da se pošalju bez garancije da će stići do ciljanog odredišta na vrijeme. Pored toga što je nepouzdana i nesigurna sama mreža, mikroservisi mogu i da postanu nefunkcionalni.

Tehnička rješenja kao unapređenje opreme mogu smanjiti vjerovatnoću problema, ali ne mogu potpuno otkloniti rizik. Bolja oprema može osigurati veću dostupnost mikroservisa, ali se ovim povećavaju troškovi. Rizici, poput nestabilnosti mreže, ostaju prisutni bez obzira na tehnička poboljšanja, pa će se uslijed pokušaja da se ti rizici minimalizuju, dodatno povećati troškovi održavanja sistema. Ovi troškovi, zajedno sa mogućnošću nestabilnosti i prekida u radu mreže, predstavljaju izazov koji mikroservisna arhitektura treba da rješava na više nivoa – ne samo kroz tehnička unapređenja već i kroz optimizaciju procesa.

Tehnološki pluralizam predstavlja problem koji može nastati zbog tehnološke heterogenosti, koja je prethodno navedena kao jedna od prednosti mikroservisne arhitekture. Iako mikroservisi ne moraju da dijele istu razvojnu tehnologiju – svaki mikroservis može biti razvijen u različitim tehnologijama u skladu sa specifičnim zahtjevima i performansama – ta raznovrsnost može postati problematična kada se poveća broj mikroservisa.

---

15 Namiot, Dmitri; Sneps – Sneppe, Manfred: On Micro – services Architecture, *International Journal of Open Information Technologies*, vol. 2, no. 9, 2014, pp. 24 – 27.

16 Thönes, Johannes: Microservices, *IEEE software*, vol. 32, no. 1, 2015, p. 116.

Ako arhitektura sistema sadrži desetine ili čak stotine mikroservisa, upotreba različitih tehnologija može značajno povećati složenost sistema. To vodi ka problemima u održavanju i integraciji, zbog čega postaje ključno uspostaviti balans između fleksibilnosti u izboru tehnologija i dugoročne održivosti sistema.

U pojedinim trenucima više neće biti razvojnog tima ili inženjera koji će u potupnosti razumjeti cijeli sistem.

I pored toga što sve ovo zvuči problematično, u praksi je dokazano da zbog same karakteristike razvoja mikroservisne arhitetkure dosta rijetko dolazi do potrebe da grupa ili individualac moraju da razumiju cijeli sistem. Rješenje koje se predlaže jeste uniformisanje tehnologija mikroservisa kroz sistem kako bi se umanjila kompleksnosti, a kako bi se u isto vrijeme zadržale visoke performanse. Karakteristika mikroservisne arhitekture jeste da su jednako važne arhitektura i organizacijska struktura. Prema Konvejevom zakonu, navedena karakteristika stvara zavisnost između organizacije i arhitekture. Navedena zavisnost će stvoriti problem kada je neophodno da se napravi ponovno fakturisanje arhitekture, pošto onda mora doći i do promjena nad organizacijom. Navedena pojava će otežati arhitekturalne promjene, a navedeni problem će se javiti kod svih projekata koji su propraćeni kroz Konvejev zakon.

2.3. Karakteristike servisnih arhitektura

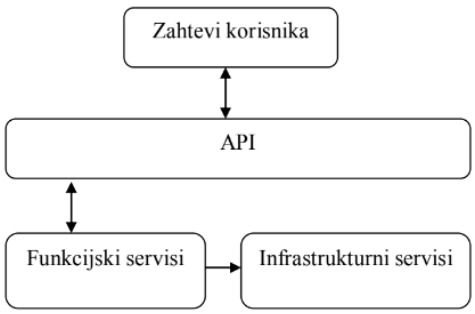
Servisnim arhitekturama se smatraju SOA i mikroservisna arhitektura. One prikazuju bitnost servisa, kao najbitniju karakteristiku arhitekture. Servis će obavljati i implementirati određeni poslovni zadatak organizacije, kao i ostale funkcije koje nisu vezane za poslovanje. Obije navedene arhitekture jesu dva potpuno različita arhitekturalna stila, ali imaju neke identične karakteristike koje proizilaze iz činjenice da se oba arhitekturalna stila orijentišu na servise. Ono što je zajedničko za sve servisne arhitekture jeste to da su generalno distribuirane.[17] Drugim riječima, komponentama može da se pristupi sa određene udaljenosti uz pomoć određenog protokola pristupa: AMQUP (*Advanced Message Queuing Protocol*), SOAP (*Simple Object Access Protocol*), MSMQ (*Microsoft Message Queueing Protocol*), REST, itd.

U samoj praksi, veoma često se javljaju REST i SOAP protokoli pristupa. Distribuirane arhitekture omogućavaju bitne prednosti nad monolitnim arhitekturama – nezavisnije su, nude daleko veću kontrolu nad testiranjem i razvojem. Modularnost takođe predstavlja jednu od važnijih

[17] Richards, Mark: *Microservices vs. service-oriented architecture*, O'Reilly Media, California, 2015, p. 22.

karakteristika servisnih arhitektura. Ovo je praksa koja opisuje enkapsulaciju djelova aplikacije u servise koji mogu pojedinačno da se razviju, dizajniraju, testiraju i puste u rad sa malom ili gotovo nikakvom zavišnošću od ostalih servisa u aplikaciji.

Međutim, navedene prednosti mogu da dovedu do određenih problema i izazova koji kasnije mogu da se pojave. Razmjena koja će se dobiti sa navedenim prednostima jeste da će doći do povećanja troškova i kompleksnosti razvoja, pa treba osigurati da se distribuirana arhitektura na pravilan način implementira, kako bi korisnost mogla višestruko da se vrati u odnosu na cijenu i kompleksnost koju može da stvori. Na osnovu taksonomije servisa mikroservisne arhitekture i SOA mogu da se objasne različitosti dva stila arhitekture. Na najvišem nivou, može da se da podjela servisa na dvije bazične vrste: poslovno područje (*Business Area*) i tip servisa (*Service Type*). Tip servisa se prvenstveno odnosi na vrstu uloge koju servis može da zauzme u sveukupnoj arhitekturi. Na primjer, određeni servisi mogu da implementiraju poslovnu funkcionalnost, dok pojedini servisi mogu da implementiraju funkcionalnost koja nije veza za poslovanje, kao što su bilješke i zapisi, sigurnosti i revizije. Sa druge strane, klasifikacija poslovnog područja se odnosi na servis koji je već implementiran da nudi usluge u nekoj organizacionoj jedinici. Na primjer, klasifikacija servisa prema poslovnom području može da bude primanje narudžbina, izvještavanje, obavljanje transakcija i slično. Mikroservisna arhitektura posjeduje taksonomiju servisa prilikom ispitivanja klasifikacija tipa servisa – uglavnom se sastoji od dvije vrste servisa: infrastukturni servisi (*Infrastructure Services*) – omogućavaju podršku nefunkcionalnim zadacima kao što su autorizacija, autentifikacija, posmatranje, revizija, bilježenje događaja; funkcijski servisi (*Functional Services*) koji omogućavaju određene poslovne funkcije ili operacije. Bitna karakteristika navedene dvije vrste servisa jeste ta da infrastrukturni servisi ne samo da su vidljivi javno, nego mogu da se tretiraju kao djeljivi, privatni servisi koji su dostupni interno drugim servisima. Funkcijski servisi su javno vidljivi i dostupni, ali se generalno ne dijele među internim servisima.

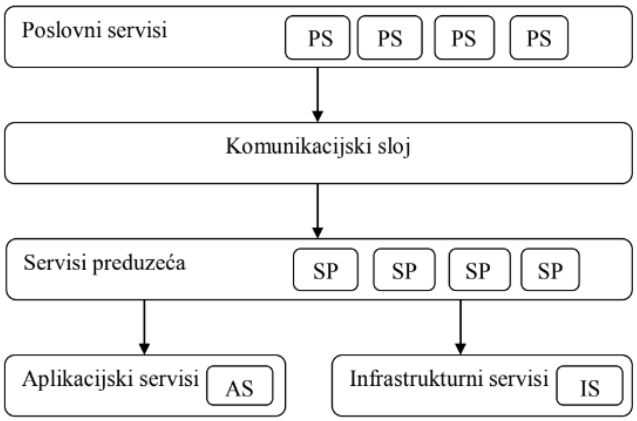


Slika 3: *Taksonomija mikroservisne arhitekture*[18]

[18] Richards, Mark: *Microservices vs. service-oriented architecture*, O'Reilly Media, California, 2015, p. 23.

Različitu taksonomiju servisa ima servisno – orijentisana arhitektura – postoji formalna i distinktna servisna taksonomija kada se tumači klasifikacija tipa servisa u odnosu na mikroservisnu arhitekturu. Ona definiše četiri grupe servisa, u pogledu tipa servisa i njegove uloge u cjelokupnoj arhitekturi sistema. Uprkos ovome, arhitekta sistema ne mora da se pridržava ove standardne klasifikacije servisne taksonomije. On može da implementira ličnu klasifikacijsku taksonomiju servisa po potrebama i volji, ali je neophodno da napravi temeljnu dokumentaciju i dobru definiciju servisne taksonomije arhitekture sistema. Kod standardne taksonomije, postoje četiri grupe servisa:

- Servisi preduzeća (*Enterprise Services*);
- Poslovni servisi (*Business Services*);
- Infrastrukturni servisi (*Infrastructure Services*);
- Aplikacijski servisi (*Application Services*).



Slika 4: *Taksonomija servisa SOA*[19]

Poslovni servisi jesu apstraktni servisi na visokom nivou koji se bave definisanjem jezgra poslovnih operacija, a koji se obavljaju na nivou kompanija. Samim tim što su apstraktni, oni neće implementirati određenu logiku, kao ni protokol, nego će uključiti naziv servisa, kao i očekivan ulaz i izlaz. Za svoju reprezentaciju, poslovni servisi veoma često upotrebljavaju BPEL (*Business Process Execution Language*), XML (*eXtensible Markup Language*), WSDL (*Web Services Definition Language*). Komunikacijski sloj služi kao posrednik između različitih servisa u arhitekturi SOA (*Service-Oriented Architecture*), omogućavajući njihovu međusobnu interakciju i razmjenu podataka. On upravlja protokolima za prenos podataka, omogućavajući besprijekornu komunikaciju između poslovnih, aplikacijskih, infrastrukturnih i servisa preduzeća, bez obzira na njihovu fizičku lokaciju ili tehničku implementaciju. Servisi preduzeća predstavljaju konkretne

[19] Richards, Mark: *Microservices vs. service-oriented architecture*, O'Reilly Media, California, 2015, p. 24.

servise na nivou preduzeća koji implementiraju funkcionalnost, a koja će se definisati određenim poslovnim procesom.

Servisi na nivou preduzeća mogu imati veze jedan-na-više ili jedan-na-jedan i mogu se razvijati uz pomoć različitih programskih platformi ili jezika, ili mogu biti gotovi proizvodi tj. softver treće strane (*Third party softvare*). Ovi servisi su često distribuirani širom organizacije i obuhvataju širok spektar funkcionalnosti. Na primjer, servisi kao što su „Validiraj narudžbinu“ ili „Kreiraj klijenta“ obezbjeđuju ključne operacije koje se koriste u različitim djelovima preduzeća. Na primjer, auto osiguravajuća kuća može da objavi servise koji će računati cijenu rate osiguranja vozila. Ovi servisi mogu da se pozovu preko korisničkog interfejsa ili kroz interfejs na nivou preduzćea. Kao primjeri mogu da se navedu: Izračunaj ratu, Dodaj vozača, Dodaj vozilo, itd. Infrastrukturni servisi predstavljaju servise koji implementiraju nefunkcionalne zadatke, kao što su sigurnosni zadaci, revizije, zapisivanje događaja, itd., kao i kod mikroservisne arhitekture. Navedeni servisi mogu da se pozovu samo uz pomoć servisa na nivou preduzeća ili aplikacijskih servisa. Najbitnija razlika između mikroservisne arhitekture i SOA je ta što je mikroservisna arhitektura fokusirana na pojedinačnim projektima, dok SOA naglasak stavlja na strukturu cijelog preduzeća, čime je izvjesna razlika koja je na organizacionom nivou.

Fokus rada SOA jeste da se razvojni timovi podijele, tako da će jedan ili više timova da budu zaduženi za razvoj servisa na strani servera, dok neki drugi timovi trebaju da implementiraju zahtjeve korisnika. Kod mikroservisne strukture je drugačije, jedan tim je zadužen za implementaciju i na strani korisnika i na strani servera, a sve sa ciljem da se ubrza razvoj funkcionalnosti i da se ostvari dobra komunikacija. Kod SOA, nova funkcionalnost će uključiti promjene nekoliko servisa i zahtjevaće komunikaciju između većeg broja timova, a kod mikroservisne arhitekture ovaj način pokušava da se izbjegne.

Tabela 1: Razlike između mikroservisne arhitekture i SOA[20]

	Mikroservisi	**SOA**
Korisnički interfejs	Sadrže interfejs	Univerzalan interfejs za sve servise
Isporuka	Mogu individualno da se isporuče	Monolitna isporuka određenog broja servisa

[20] Wolff, Eberhard: *Microservices: Flexible Software Architecture*, Addison-Wesley Professional, Boston, 2016, p. 80 – 81.

Organizacija	Razvoj servisa po tipovima po domenima	Razvoj servisa po organizacijskim jedinicama
Fleksibilnost	Fleksibilnost se ostvaruje brzim razvojem i isporukom međusobno nezavisnih mikroservisa	Fleksibilnost koja se ostvaruje uz pomoć orkestracije
Opseg	Arhitektura na nivou pojedinog projekta	Arhitektura na nivou preduzeća

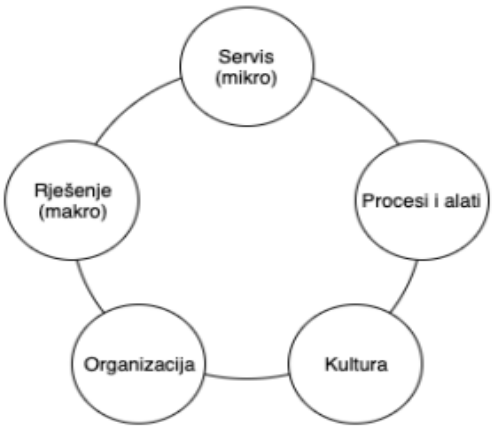
2.4. Dizajn mikroservisne arhitekture

Od razvojnih inženjera i arhitekti sistema, mikroservisna arhitektura traži da razmišljaju drugačije. Kada bi fokus bio na pojedinačnim servisima, onda bi došlo do mogućih komplikacija u kasnijim fazama razvoja. Neophodno je da se pri izradi koncepta dizajna, u obzir uzmu sve karakteristike sistema i kako svi sistemi mogu međusobno da sarađuju. Rezultat navedenog jeste ponašanje sistema koje pridaje veću korisnost od one koja će se dobiti zbrajanjem korisnosti svake određene pojedinačne karakteristike. Sistem koji se sastoji od komponenti koje sarađaju vrijedi daleko više od grupa komponenti koje rade samo za sebe. Prilikom izrade aplikacije, sistem mikroservisa obuhvata sve aspekte organizacije.

Znači da struktura ovakvog sistema može da uključi ljude koji rade u organizaciji, samu organizaciju, kao i rezultat koji je ostvaren. Isto tako, izuzetno su bitni i faktori koji se odnose na sistem u radu: koordinacija, optimizacija servisa, operacijske prakse, upravljanje greškama, itd. Kada su implementirani svi aspekti na pravi način, onda će se dobiti sistem koji će se ponašati na željen i predvidljiv način.[21] Međutim, u organizaciiji se nalazi previše promjenjivih da bi na praktičan način mogao da se proizvede dobar konceptualni dizajn. U samoj praksi, često mogu da se sretnu različite komplikacije, naročito kada je neophodno da se implementira promjena koja će sa sobom povući još promjena, iz razloga jer su djelovi sistema međusobno povezani i isprepletani. Kako bi na jednostavniji način mogao da se razumije kompleksan sistem, moguće je napraviti model kao što naučnici prave sisteme i pojave koji su i više nego kompleksni da se zamisle.

[21] Garrigos, Irene; Wimmer, Manuel: Current Trends in Web Engineering, Springer International Publishing, Rome, 2018, p. 32 – 48.

Sam model omogućava da se na precizan način shvati kako se jedan sistem ponaša, kako lakše da se predvidi ponašanje posle određene promjene, da se konceptualizuje dizjan i prepoznaju djelovi.



Slika 5: *Model mikroservisnog sistema*[22]

Ovaj model se sastoji od pet djelova: servis (mikro nivo), rješenje (makro nivo), alati i procesi, kultura i organizacija. Servisi predstavljaju bitan dio svake servisne strukture. Kod mikroservisne strukture, servisi čine gradivne blokove od koji će biti napravljen cijeli sistem. Ukoliko se uz pomoću servisa odredi pravilan opseg, dizajn, svrha i granularnost može da se indukuje veoma kompleksno ponašanje sistema koji će se sastojati od određenog broja komponenti koje će biti jednostavne same po sebi. Sistem mikroservisa neće činiti samo sistem međusobno povezanih servisa koji će razmjenjivati informacije porukama za vrijeme izvođenja. Alati i procesi su zaslužni za ponašanje sistema, a oni se izvode i upotrebljavaju u sistemu, tako da sistem izvršava svoj posao za koji je i napravljen.

Takođe se uključuju alati i procesi koji su vezani uz razvoj programskog proizvoda, isporučivanja programskog koda, upravljanje i održavanje programskih proizvoda. Izbor pravih alata i procesa izuzetno je bitan za proizvodnju dobrog ponašanja sistema mikroservisa. Na primjer, usvajanje agilnih metoda razvoja programskog jezika i alata (na primjer, Docker), mogu da povećaju promjenljivost sistema.[23]

Kultura može da se izdvoji kao neopipljivi faktor sistema, a iz tog razloga i jeste jedan od najbitnijih faktora. Izuzetno je teško da se mjeri kultura organizacije – prisutne su formalne metode,

[22] Nadareishvili, Irakli; Ronnie, Mitra; McLarty, Matt; Amundsen, Mike: *Microservice Architecture – Aligning principles, practices and culture*, O'Reilly, Boston, 2016, p. 27.
[23] Nadareishvili, Irakli; Ronnie, Mitra; McLarty, Matt; Amundsen, Mike: *Microservice Architecture – Aligning principles, practices and culture*, O'Reilly, Boston, 2016, p. 28.

modeliranja i anketiranja, međutim veliki broj društva mjeri kulturu organizacije, ali i timova koji u organizaciji rade. Kultura organizacije može da se uvidi u dnevnim interakcijama između članova timova, klijenata kojima teže i timskim rezultatima. Ona se zasniva na uvjerenjima, karakteru, nepisanim i pisanim pravilima, običajima koji se kroz vrijeme usvajaju. Organizacijska kultura je izuzetno bitna pošto oblikuje odluke na najnižem nivou poslovanja, organizacije koje imaju pozitivne rezultate posle implementiranja mikroservisne arhitekture govore kako je organizacijski dizajn bitan kod uspjeha.[24]

Veliki broj faktora utiče na način rada u organizaciji, međutim najuticajniji jesu ljudi sa kojima se radi, kao i način na koji se sa njima komunicira. Organizacijski dizajn, sa aspekta mikroservisne arhitekture jeste smjer, struktura kojom se kreće pod vođstvom kompozicije, autoriteta i granularnosti timova. Organizacija je takođe izuzetno osjetljiv dio sistema. Svaki arhitekta mikroservisnog sistema mora da razumije implikacije ukoliko dođe do izmjena organizacijskog svojstva, ali u vidu mora da ima da dobar dizajn sistema često predstavlja nusprodukt dobrog organizacijskog dizajna. Ako se uporedi sa servisima, rješenje predstavlja makro rezultat koji može da bude kombinacija mikroservisa koji će zajedno činiti sistem. Prilikom dizajniranja specifičnog mikroservisa, odluke koje se donose biće ograničene potrebom kako bi mogao da se ostvari određeni izlaz ovog mikroservisa, odnosno ponuda usluge ovog servisa. Prilikom dizajniranja arhitekture, potrebno je da se razmišlja o potrebama ove arhitekture, ali i o njenim izlazima i ulazima koje će proizvesti nekoliko servisa. Na ovom makro nivou arhitekture, dizajner sistema može da indukuje ponašanje sistema koje je unaprijed već zamišljeno. Makro nivo jeste konačan proizvod mikroservisne arhitekture i predstavlja rezultat svih ranijih aspekata: alati i procesi, servis, kultura i organizacija.

2.5. Sigurnost u mikroservisnoj arhitekturi

U mikroservisnoj arhitekturi, implementacija sigurnosti je ključna za zaštitu sistema. Bez obzira na to da li se radi o ljudima, uređajima ili komponentama, svaki entitet koji pristupa mikroservisima mora biti autentifikovan i autorizovan. Autentifikacija je proces kojim se potvrđuje identitet entiteta, dok se autorizacija odnosi na provjeru da li entitet ima odgovarajuća prava za pristup određenom servisu ili resursu. Međutim, nije efikasno implementirati zaštitu za svaki pojedinačni mikroservis zasebno. Umjesto toga, potrebno je pristupiti sigurnosti na nivou cijele

[24] Nadareishvili, Irakli; Ronnie, Mitra; McLarty, Matt; Amundsen, Mike: *Microservice Architecture – Aligning principles, practices and culture*, O'Reilly, Boston, 2016, p. 29.

arhitekture. To znači da treba postojati centralizovani mehanizam za autentifikaciju, koji će provjeravati korisničke identitete i upravljati pristupom na nivou cijelog sistema.

Što se tiče autorizacije, ona se može implementirati na način koji omogućava mikroservisima da komuniciraju sa centralnim servisom za upravljanje korisničkim pravima i ulogama. Na primjer, iako se pravila za autorizaciju mogu primjenjivati na nivou mikroservisa, važno je da centralni servis upravlja definicijom uloga i prava pristupa, kako bi se osigurala dosljednost i pravilno tretiranje zahtjeva kroz cijeli sistem.

Na ovaj način, mikroservisi mogu da se fokusiraju na specifične funkcionalnosti, dok centralni servis upravlja sigurnosnim aspektima kao što su autentifikacija i autorizacija, omogućavajući bolju kontrolu i konzistentnost u zaštiti cijelog sistema.

2.6. OAuth2

OAuth 2.0 predstavlja autorizacijski protokol koji će aplikacijama trećih strana omogućiti da ostvare pristup HTTP servisu koji će biti limitiran.[25] Sam pristup može da se ostvari u ime vlasnika resursa, tako što će se orkestrirati interakcija odobrenja između HTTP servisa i vlasnika resursa ili da se omogući da treća strana uz pomoć aplikacije ostvari pristup na svoje ime. Navedeni protokol jeste jedan od rješenja za implementiranje autorizacije. On je dosta rasprostranjen i upotrebljava se od velikog broja najpoznatijih tehnoloških kompanija: Microsoft, Google, Yahoo i slično. Na osnovu standarda je definisan tok događaja OAuth2 protokola:

- Klijent potražuje od vlasnika resursa da li može da obavi nad njim određenu akciju. Na primjer, aplikacija će tražiti zahtjev kako bi mogla da pristupi pojedinim podacima ili korisničkom profilu na društvenoj mreži, gdje je vlasnik resursa unosio podatke. Isti taj vlasnik je najčešće korisnik sistema.
- Ukoliko vlasnik resursa dodijeli prava klijentu, onda će klijent dobiti odgovor koji potražuje od samog vlasnika.
- Klijent će pročitati odgovor i upotrijebiće ga za slanje zahtjeva serveru za autorizaciju. U navedenom primjeru, u društvenoj mreži bi mogao da se nađe server za autorizaciju.
- Server za autorizaciju će vratiti pristupni token.

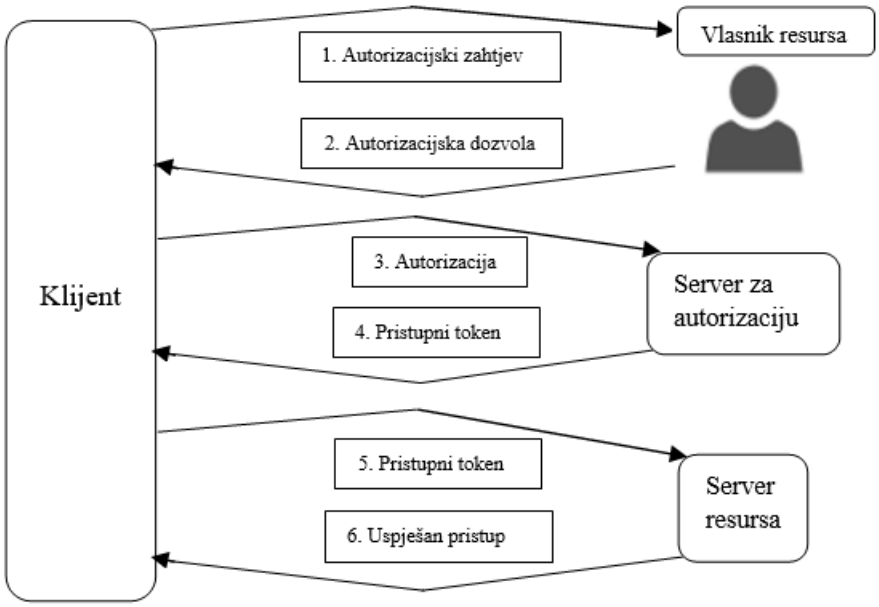
[25] IETF, *The OAuth 2.0 Authorization Framework*, 2012. https://tools.ietf.org/html/rfc6749. posjeta: 09.04.2024

- Klijent, uz pomoću tokena koji je dobijen od servera za autorizaciju, može da zatraži od servera resursa sve informacije koje ga zanimaju. Veoma često, token će se staviti u zaglavlje prilikom slanja zahtjeva.
- Server resursa će na zahtjev odgovoriti.

Na razne načine može da se odvija interakcija sa serverom za autorizaciju, a ti načini mogu biti:[26]

- Slučaj autorizacije uz pomoć lozinke: klijent će korisniku prikazati HTML formu u prvom koraku autorizacije. Korisnik je taj koji će unijeti lozinku i korisničko ime. Kod trećeg koraka, ovu informaciju korisnik će da koristi kako bi uspio da dohvati pristupni token sa servera za autorizacijum uz pomoć HTTP POST metode. Navedeni pristup je dosta nesiguran, iz razloga jer je prisutna mogućnost da klijent može da bude nesigurno ili neispravno implementiran, pa na taj način mogu da se ugroze i sami podaci.
- Implicitno odobravanje – posle preusmjeravanja na server za autorizaciju, klijent će direktno pristupiti pristupnom tokenu uz pomoć HTTP preusmjeravanja. Ovo će mobilnim aplikacijama i internet pretraživačima omogućiti da odmah pročitaju pristupni token. Međutim, sam token nije potpuno zaštićen od zlonamjernih napada iz razloga jer server za autorizaciju neće klijentu poslati direktno pristupni token, već će to uraditi uz pomoć HTTP preusmjeravanja. Navedeni način treba da se upotrebljava kada pristupnim tokenom može da se manipuliše uz pomoć JavaScript programskog jezika na strani mobilnih aplikacija ili klijenta.
- Autentikacija klijenta – u prvom koraku se autentifikuje klijent kako bi mogao da se dohvati pristupni token sa servera za autorizaciju. Na navedeni način, klijent može da pristupi podacima bez informacija koje se dodaju od strane vlasnika resursa. Na primjer, na navedeni način, statistički softver može da analizira i iščita podatke klijenata.

[26] Nadareishvili, Irakli; Ronnie, Mitra; McLarty, Matt; Amundsen, Mike: *Microservice Architecture – Aligning principles, practices and culture*, O'Reilly, Boston, 2016, p. 54.



Slika 6: *OAuth2 slijed događaja*[27]

Uz pomoć pristupnog tokena, klijent će uspjeti da ostvari prava pristupa zaštićenim resursima. Upravo iz ovog razloga, treba da bude zaštićen pristupni token; kada bi neautorizovana lica ostvarila prava pristupa nad resursima, onda bi ta lica mogla da se pretvaraju da su vlasnici ovih resursa i da izvršavaju akcije nad njima, a to potencijalno može da bude izuzetno opasno. Pristupni token može da sadrži pojedine dodatne enkodirane informacije. Na primjer, mogu da budu prisutne informacije kao što je puno ime vlasnika resursa, kao i nivo prava koji ima vlasnik resursa. Nivo prava nad resursima će odrediti koje akcije nad kojim resursima mogu da se izvršavaju i mogu da pridodaju korisniku specifično ili cijeloj grupi korisnika.

2.7. JSON Web token

JSON Web token (*JSON Web Token* – JWT) predstavlja siguran i kompaktan način prenosa informacija između dvije strane.[28] Informacije u JSON Web tokenu enkodirane su kao JSON objekt koji se upotrebljava kao čitljiv tekst JSON Web enkripcije (*JSON Web Encryption*) ili kao sadržaj JSON Web potpisa (*JSON Web Signature*). Integritet tereta može da se osigura uz pomoć implementacije MAC–a (*Message Authentication Code*) i digitalnog potpisa.

[27] IETF, *The OAuth 2.0 Authorization Framework*, 2012. https://tools.ietf.org/html/rfc6749. posjeta: 09.04.2024
[28] IETF, *JSON Web Token (JWT).* 2015. https://tools.ietf.org/html/rfc7519. posjeta: 09.04.2024

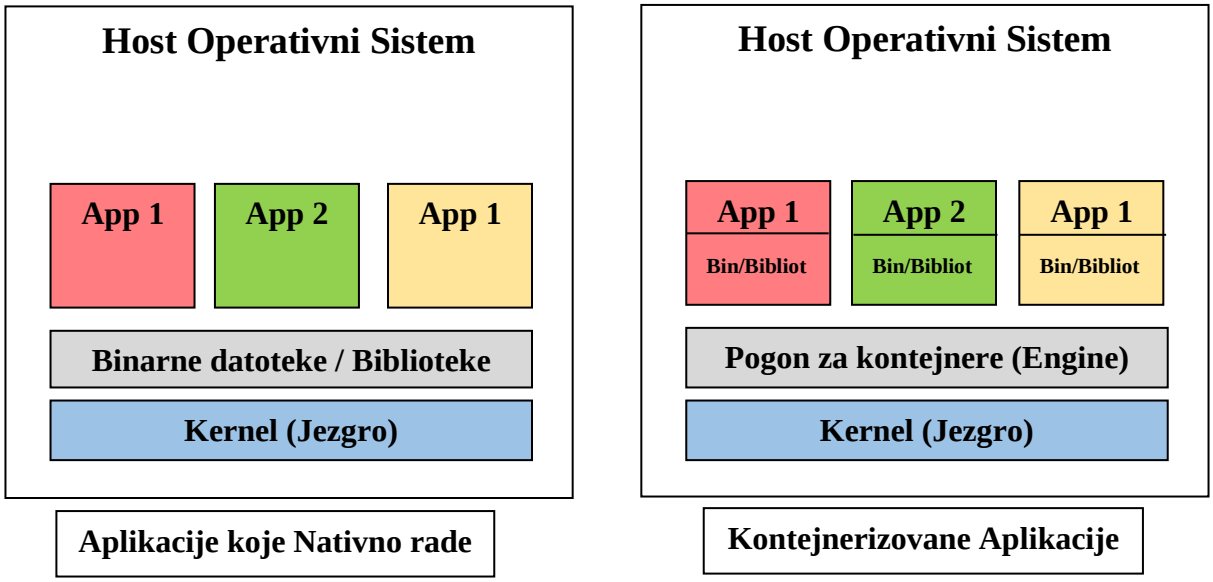
Kod mikroservisne arhitekture, korisnik može da se autentifikuje uz pomoć jednog od OAuth2 pristupa. Posle ovoga, korisnik će imati pristup korisničkom interfejsu nekog mikroservisa ili će moći da pošalje REST zahtjev. Svaki zahtjev korisnika može da prođe kroz određeni broj mikroservisa koji će, upotrebljavajući pristupni token korisnika, moći da odredi da li korisnik ima dovoljno prava pristupa.

Kod JSON Web tokena, token prvo treba da se dešifruje, a potom i da se provjeri da li je ispravan digitalni potpis servera za autorizaciju. Na kraju, odluka da li korisnik ne može ili može da koristi usluge mikroservisa na način na koji želi, u velikoj mjeri će zavisiti od informacija koje pristupni token nosi. Bitno je da se napomene da pristupni token na smije da sadrži informacije o tome kojem mikroservisu smije da pristupi sam korisnik, a kojem ne smije. Prema OAuth 2 standardnom protokolu, pristupni token će se izdati od strane servera za autorizaciju. Kako bi prava pristupa na serveru za autorizaciju mogla da se odrede, onda bi bila prisutna opasnost od napada na centralno mjesto koje je zaduženo za određivanje prava pristupa nad cijelim sistemom, pa bi onda došlo do poništenja ili umanjenja prirodno urođene sigurnosti distribuiranih sistema kao što su mikroservisi. Upravo iz tog razloga, server za autorizaciju jedino treba da upravlja korisničkim grupama, dok individualni mikroservisi su ti koji će odlučiti pravo pristupa i to na temelju informacija koje su dostupne iz pristupnog tokena.

3. KONTEJNERIZACIJA

3.1. Pojam kontejnerizacija

Kontejneri predstavljaju izvršne jedinice softvera u kojima će se naći kod aplikacije, a cijela jedinica je izolovana od ostatka sistema. Svaki kontejner ima neophodne programske biblioteke (*Library*) i ostatak konfiguracije koje su neophodne za pokretanje na bilo kojem drugom uređaju ili sistemu. Kako bi moglo da se postigne sve navedeno, kontejneri koriste mogućnosti operativnog sistema na kojem se nalaze, kako bi procesima mogla da se omogući neophodna procesorska izolacija, snaga, prostor i memorija. Sve navedeno se postiže kroz pojedine tehnologije koje se implementiraju u operativnim sistemima. Najlakše razumijevanje kontejnera može da se postigne ako se napravi poređenje sa virtuelnim mašinama. Svaka virtuelna mašina posjeduje svoj virtuelizovani hardver, operativni sistem, aplikacije sa neophodnim programskim bibliotekama, ali i konfiguracijom koja je potrebna za njihovo uspješno pokretanje. Od svih komponenti koje su navedene, kontejneri imaju samo neophodne konfiguracijske datoteke i programske biblioteke za pokretanje aplikacija koje se nalaze u njima, a izostanak potrebe za operativnim sistemom predstavlja glavni razlog zašto su kontejneri prenosivi, efikasni i brzi.



Slika 7: *Standardan sistem (lijevo) i sistem koji sadrži kontejnere (desno)*[29]

[29] Koutoupis, Petros: *Everything You Need to Know about Linux Containers, Part II: Working with Linux Containers (LXC)*, 2018. https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-ii-working-linux-containers-lx. posjeta: 15.04.2024

Uprkos zavisnosti kontejnera od kernela operativnog sistema, postoji veliki broj tehnologija koje omogućavaju izvršavanje kontejnera koji se temelje na Linux operativnom sistemu na MacOS i Windows uređajima, kao što su Hypervisor, WSL i HyperV virtuelizacijska tehnologija. Zbog ovih alata može da se napravi kontejner na Linux operativnom sistemu i da se pokrene na bilo kojoj platformi.[30]

Pošto virtuelne mašine doprinose boljoj iskorišćenosti resursa sistema, na isti način i kontejneri upotrebljavaju memoriju i procesorsku snagu fizičke mašine. Ali, u pojedinim djelovima kontejneri daleko bolje izvršavaju posao. Ovdje se radi o većim aplikacijama koje mogu da se podijele na manje djelove, a ovi djelovi mogu da se podijele u nekoliko kontejnera, time će se povećati iskorišćenost sistema. Isto tako, kontejneri imaju i pojedine mane koje moraju da se znaju prije nego što se donese odluka o njihovom korišćenju. Za razliku od virtuelnih mašina, kontejneri upotrebljavaju resurse daleko opreznije, ali zbog većeg broja slojeva koji su uključeni za vrijeme njegovog korišćenja, ni oni ne mogu da pruže savršene performanse.

Pošto različite kompanije pružaju razne tehnologije za upravljanje kontejnerima, normalno je da se očekuju pojedine nekompatibilnosti ako bi došlo do pokušaja da se kombinuje nekoliko tehnologija. Isto tako, za vrijeme upotrebe kontejnerskih tehnologija u Linux operativnom sistemu, podrazumijeva se da je korisnik veoma dobro upoznat sa komandnom linijom. Ako bi se pojavila potreba za grafičkim interfejsom, postoje pojedine tehnike kojima bi se moglo postići tako nešto. Ovakva potreba u velikoj mjeri sve komplikuje, pa nedostatak grafičkog interfejsa može da se navede kao potencijalni nedostatak.

U poslednjih nekoliko godina, upotreba kontejnera se povećala do te mjere da pojedine kompanije, u budućnosti, imaju plan da kontejnere upotrebljavaju kao zamjenu za virtuelne mašine koje su se do sada koristile. Ako je riječ o kombinovanju mikroservisne arhitekture i kontejnera kao platforme, onda tako nešto predstavlja dobar temelj za timove koji upotrebljavaju DevOps (*Development and Operations*) način rada. Isto tako, kontejneri se upotrebljavaju jer je neophodno da se „modernizuju“ aplikacije, a proces kada se prebacuje obična aplikacija u kontejner poznat je pod nazivom kontejnerizacija.

Početkom 1979. godine predstavljena je croot (*change root*) naredba, a po samom imenu može da se zaključi da navedena naredba omogućava mijenjanje trenutnog direktorijuma prokrenutog procesa u root direktorijumu, ali i svih njegovih hijerarhijskih poddirektorijuma. Na ovaj način je stvorena mogućnost izolovanja procesa u svoj poseban konfiguracijski sistem koji ne

[30] Kane, Sean; Matthiass, Karl: *Docker: Up & Running*, O'Reilly Media Inc., Boston, 2023, p. 1 – 9.

bi uticao na ostatak sistema. Tri godine kasnije, tačnije 1982. godine, chroot naredba postaje dio Unix operativnog sistema.

Već tokom 1990. godine, Bil Čuzvik, stručnjak za sigurnost i mreže, provodio je istraživanje kako bi bolje razumio ponašanje hakera kada bi stekao pristup sistemu. Čuzvik je želio da istraži kako bi napadači mogli da iskoriste sigurnosne propuste i koje bi metode mogli koristiti za eskalaciju privilegija ili prikrivanje svojih aktivnosti. Da bi ovo istraživanje bilo moguće, stvoreno je simulirano okruženje, tzv. testna laboratorija, koje je omogućavalo praćenje i analiziranje postupaka hakera u kontrolisanim uslovima. Ovo okruženje je imalo za cilj da replicira stvarne uslove u kojima hakeri djeluju, pružajući uvid u njihove taktike, tehnike i procedure. Tokom 2000. godine, FreeBSD uvodi *jail* naredbu u svoj sistem. I pored toga što je veoma slična chroot naredbi, navedena naredba uključuje pojedinu izolaciju procesa u vidu mreža, sistema, korisnika. Naredba *jail* je omogućila sistemu posebno podešavanje, dopuštajući instalacije u svakoj "ćeliji", no aplikacije unutar tih "ćelija" ograničene su u samoj funkcionalnosti.

Priča se dalje nastavlja 2006. godine, kada su inženjeri Google–a najavili da objavljuju njihovu novu tehnologiju, proces kontejnera, koji je napravljen sa svrhom da izoluje i ograniči resurse koje pojedini procesi upotrebljavaju.[31] Naredne godine, tehnologija je dobila drugi naziv, kontrolne grupe (*Control Groups*). Tokom 2008. godine, kontrolne grupe su postale dio Linux kernela, pa je na taj način stvoren LinuX Container (LXC) tehnologija.

Ova nova tehnologija predstavlja prvu potpunu implementaciju Linux kontejner menadžera. Od tehnologija su korišćeni imenski prostor i kontrolne grupe. Control Groups jesu svojstvo Linux kernela koje procesima omogućava da se organizuju u hijerarhijski oblikovane grupe. Grupisanje se odvijalo u jezgru kernela kod kontrolne grupa, a samo ograničavanje, praćenje i kontrolisanje se obavljalo u podsistemima (Subsystems), od kojih je svaki zadužen za pojedinu vrstu resursa (memorija, procesor i slično).

3.2. Prednosti kontejnerizacije

Arhitekture koje se temelje na kontejnerizaciji omogućavaju veliki broj prednosti iz perspektive skalabilnosti i sigurnosti, a koje su poželjne tokom razvijanja kompleksnijih programskih rješenja. Izolovanje određenih djelova aplikacije onemogućava dijeljenje koda kroz cijelu aplikaciju, pa se na taj način problem izoluje na jedan kontejner koji se jednostavno mijenja pomoću ažurirane verzije. Kontejnerizacija omogućava jednostavno ažuriranje koda bez uticaja na

[31] Kane, Sean; Matthiass, Karl: *Docker: Up & Running*, O'Reilly Media Inc., Boston, 2023, p. 1 – 9.

korisničko iskustvo, pa se na taj način aktiviraju isti kontejneri prethodne verzije aplikacije zajedno sa ažuriranom verzijom kontejnera.

Tokom navedenog načina aktiviranja novog kontejnera, dio korisnika počinje da koristi staru verziju programskog rješenja koji će se postepeno mijenjati novom verzijom. Navedeni način puštanja aplikacije u rad poznat je pod imenom *Canary deploy.*[32] Pored poboljšanja iz perspektive sigurnosti i jednostavnog ažuriranja, kontejnerizacija omogućava i poboljšanje pouzdanosti programskog rješenja. U slučaju nepravilnog rada neke instance kontejnera, on veoma brzo može automatski ili manualno da se zamijeni novom instancom, a uz redukovanje instance baze podataka, korisnički podaci će uvijek ostati sačuvani.

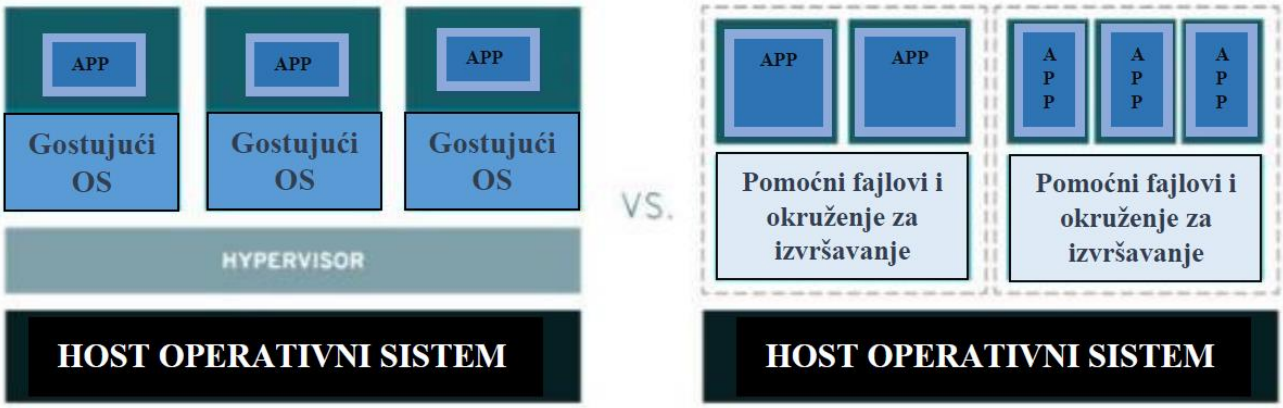
S obzirom da manuelna zamjena instanci kontejnera, sa svrhom održavanja rada sistema ili poboljšanja performansi sistema, zahtijeva veću količinu vremena i podložna je ljudskoj grešci, zamjena se veoma često radi tako što se koriste posebni alati koji su namijenjeni orkestraciji rada većeg broja kontejnera. Alati za automatsku orkestraciju veoma često se upotrebljavaju u sistemima koje upotrebljava veći broj korisnika, pa je iz tog razloga neophodno da se kontinuirano raspoređuju resursi grupama kontejnera, u zavisnosti od njihovog broja. Ovakvi alati omogućavaju praćenje zdravlja instanci kontejnera, jednostavno proširivanje i ažuriranje programskih rješenja, automatsko skaliranje, ali i zamjenu kontejnera kada je tako nešto neophodno.[33] Trenutno, na tržištu je prisutno nekoliko alata za automatsku orkestraciju kontejnerskih aplikacija, a posebno mogu da se izdvoje Docker Swarm, RedHat Openshift, Kubernetes. Ovi alati su veoma dobro testirani i dokumentovani u praksi, pa iz tog razloga predstavljaju odlično rješenje za automatsku orkestraciju.

3.3. Razlike u odnosu na virtuelnu mašinu

Na Slici 8, sa lijeve strane je prikazan sistem sa hipervizorom tipa 2 i nekoliko virtuelnih mašina, a sa desne strane je prikazan sistem koji sadrži kontejnere. Prva razlika koja može da se uoči jeste nedostatak operativnih sistema u sistemu sa kontejnerima, drugim riječima upotrebljava se samo jedan sistem. Sa druge strane, svaka virtuelna mašina ima poseban operativni sistem. Baš iz tog razloga, količina sistemskih resursa potrebnih za pokretanje virtuelne mašine predstavlja jednu od ključnih razlika između ovih tehnologija. Veličina kontejnera može da se mjeri u megabajtima,

[32] Servile, Valentina: *Canary Development*, O'Reilly Media Inc., Boston, 2023, p. 11 – 13.
[33] Kebbani, Nassim; Tylenda, Piotr; McKendrick, Russ: *The Kubernetes Bible*, Packt Publishing, Birmingham, 2022, p. 4 – 8.

dok veličina virtuelnih mašina može da dostigne i nekoliko gigabajta. Zbog navedenih činjenica, na sistemu može da se nalazi daleko više kontejnera nego virtuelnih mašina.



Slika 8: *Sistemi sa virtuelnim mašinama (lijevo) i sistemi sa kontejnerima (desno)*[34]

Pošto virtuelne mašine upotrebljavaju operativni sistem sa svim propratnim postavkama i programima, smatra se da je korisnik u prednosti zbog mogućnosti bolje pokrivenosti i boljeg upravljanja resursima. Ali, potpuni operativni sistem, zbog svoje veličine, može da potroši i do nekoliko minuta prije nego što je spreman za upotrebu. Kontejneri u velikoj mjeri prednjače u navedenom aspektu pošto postaju spremni za upotrebu u dosta kraćem vremenskom intervalu. Ako kontejner nije na dobar način konfigurisan, svaki korisnik, koji ima puno pravo pristupa, mogao bi da pristupi operativnom sistemu i da prikupi informacije o drugim kontejnerima. Sa druge strane, virtuelne mašine se smatraju sigurnijom opcijom, pošto sve što je potrebno može da se nađe u jednoj virtuelnoj mašini, drugim riječima ne postoji potreba za komunikacijom u količini koja je prisutna kod upotrebe kontejnera.

Kontejneri su izuzetno korisni ako se javlja potreba da se pokrene puno manjih programa na jednom sistemu, ako je neophodno da se isti program pokrene veći broj puta, ako se javlja potreba za sistemom koji neće zahtjevati puno resursa i moguće ga je brže pokrenuti, ako korisnik neće da vodi brigu o konfiguraciji i konfigurisanju programa kako bi funkcionisali na različitim sistemima. Virtuelne mašine su korisne u slučajevima potrebe za nekoliko različitih operativnih sistema, ako se javlja potreba za upravljanjem nekoliko većih programa na jednom sistemu, ako se pokreće program koji zahtijeva sve resurse i funkcionalnosti operativnog sistema, ako je neophodna potpuna sigurnost i izolacija. Isto tako, na jedan slikovit način može da se opiše razlika između navedene dvije tehnologije, tako što će se dati jedno zanimljivo poređenje. Pojmovi koji se koriste jesu stanovi

[34] Red Hat Containers vs VMs, 2023. https://www.redhat.com/en/topics/containers/containers-vsvms. posjeta: 15.04.2024

(kontejneri) i kuća (virtuelna mašina). Kuća, sa svojim instalacijama, ali i nekom dozom privatnosti od neželjenih posjetilaca izuzetno je velika i nezavisna. Sa druge strane, stanovi imaju identične instalacije kao i kuća, ali ove instalacije se dijele i sa drugim stanovima. Stanovi dolaze u raznim veličinama, a stanari će iznajmiti stan one veličine koja im je neophodna. Ovakva poređenja mogu da se prenesu i na tehnologije, kao što su virtuelne mašine i kontejneri.[35]

3.4. Kontejnerizacija u svrhu uspostavljanja servisa

Tokom donošenja odluke o kontejnerizaciji, bitno je da se donese odluka da li vrijedi da se započne cijeli proces. Postaviće se pitanja da li je trenutni proizvod speman da se u kontejneru pokrene, da li će poslije uspješne pripreme sve dobro da funkcioniše i da li će sve proći bez većih grešaka. Kontejneri predstavljaju dobru opciju za proizvode koji mogu da se podijele u nekoliko nezavisnih djelova. Ali, ako je riječ o zastarjelom proizvodu koji ne može lako da se dijeli, neophodno je da se u obzir uzme koliko je vremena neophodno za pripremu za izmjene aplikacije. Kontejneri, po prirodi, mogu da se okarakterišu kao lako prenosivi, a to znači da mogu ponovo lako da se iskoriste. Ako se pokuša napraviti neki zahtjevniji proces, veoma lako može da se stvori „mala virtuelna mašina“. U ovakvoj situaciji, ako se radi o nekom većem programu, javiće se i veća potreba za kontrolom. Dijeljenje programa na manje funkcionalne djelove, da bi se lakše rukovalo sa njima, izuzetno je koristan proces, ali sa sobom povlači i obaveze pravilnog upravljanja sa nekoliko novonastalih cjelina.

Pored pravilnog upravljanja, neophodno je da se vodi računa o svakom detalju u kontejneru, pošto ih oni sami po sebi neće dovoljno pružiti, a to predstavlja potencijalni problem ukoliko je prisutan veći broj kontejnera. Ukoliko je proizvod, koji treba da se preseli u kontejnere, proizvođač većeg broja izlaznih ili ulaznih operacija koje moraju da se prate, u obzir treba da se uzmu i jedinice za skladištenje. Pošto se kontejneri lako mijenjaju sa novim kontejnerima, potrebno je da se osigura mogućnost upotrebe postojećih podataka. U obzir mora da se uzme i način na koji će prebacivanje u kontejnere da utiče na stanja programa koji se spremaju, pošto će od sada na njih da utiče veliki broj manjih jedinica. Zbog načina na koji kontejneri pristupaju sistemu na kojem se nalaze, sigurnost predstavlja jednu od tema koje se često spominju kada je riječ o kontejnerizaciji. Pored brige o sigurnosti sistema na kojima se nalaze sami kontejneri, neophodno je da se vodi računa i o njihovoj izolaciji. Jedan od načina na koji ovako nešto može da se postigne jeste da im se dodijele najniže

[35] Chelladhurai, Jeeva; Singh, Vinod; Raj, Pethuru: *Learning Docker, Second Edition*, Packt Publishing, Birmingham – Mumbai, 2017, p. 54.

privilegije u vidu prava pristupa. Drugim riječima, radi se o pokretanju programa unutar kontejnera sa korisnikom koji ima osnovna prava pristupa. Postoji mogućnost i da se ograniči upotreba resursa po kontejneru, a na ovaj način će se isključiti napadi koji imaju za cilj da iscrpe resurse. Tokom kreiranja novog *image*-a, potrebno je da se vodi računa o osnovnom *image*-u, o broju njegovih preuzimanja, kao i da li dolazi iz sigurnih izvora. Praćenje i nadgledanje stanja predstavlja još jedan dio koji je izuzetno bitan. Ako se izostavi ovaj aspekt, utvrđivanje uzroka grešaka bilo bi znatno otežano, što bi dodatno usporilo odgovarajuće djelovanje. Ako se pođe od činjenice da proizvod koji želi da se kontejnerizuje, već ima neka rješenja za praćenje i nadgledanje stanja, u tom slučaju ne bi ni smjeli da se jave veći problemi. Na primjer, Docker pruža veliki broj ovakvih mogućnosti i omogućava izvoz povratnih informacija aplikacije u razne formate.

3.5. Docker tehnologija

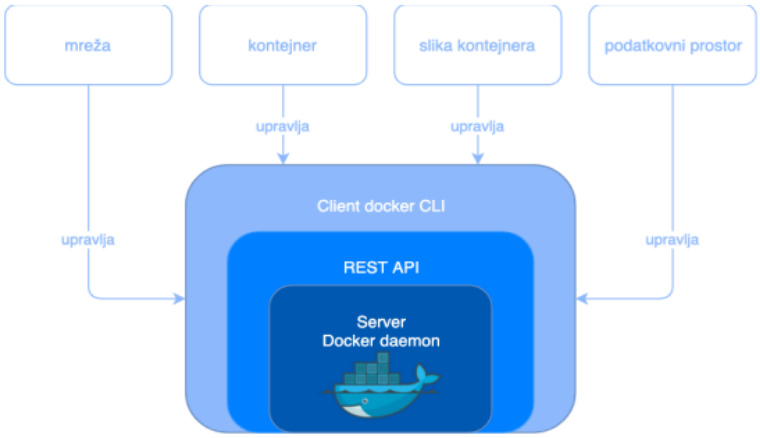
Tokom 2008. godine, u Parizu, od strane Sebastiana Pala, Kamel Founadia i Solomon Hejksa osnovana je kompanija dotCloud. Ova kompanija predstavlja preteču kompanije Docker Inc, koja je tokom 2010. godine, osnovana u Sjedinjenim Američkim Državama od strane istih ljudi. Kompanija Docker Inc. razvija program Docker, koji predstavlja prvu stabilnu verziju, a zvanično je objavljena 2013. godine. Docker predstavlja mehanizam otvorenog koda koji automatizuje implementaciju aplikacija u kontejnere.[36] Dizajniran je da omogući brzo i lagano pokretanje koda, kao i za efikasan tok rada kod prenošenja koda sa računara na testno okruženje.

Izuzetno je jednostavan, za njegovo pokretanje dovoljno je da računar ima Docker binarnu datoteku i Linux jezgro.[37] Docker ima misiju da pruži sledeće:

- Logična podjela dužnosti – Programeri imaju zadatak da brinu o samoj aplikaciji koja se izvodi unutar kontejnera, dok je Docker taj koji brine o upravljanju kontejnerima. Docker je napravljen tako da poboljša dosljednost, time što osigurava da okruženje u kojem programer piše kod, može da odgovori okruženjima u kojima su raspoređene aplikacije.
- Lagan i jednostavan način modeliranja stvarnosti – Docker je izuzetno brz, pa na taj način aplikacija može da se za svega nekoliko minuta kontejnerizuje. Oslanja se na model *Copy–on–write*, pa je svaki unos promjena u aplikaciju isto veoma brz.

36 Turnbull, James: *The Docker Book*, Turnbull Press, New York, 2014, p. 7.
37 Schenker, Gabriel: *Learn Docker – Fundamentals of Docker 19.x*, Packt Publishing, Brimingham, 2020, p. 12.

- Podstiče arhitekturu usmjerenu na usluge – Docker podstiče mikroservisne i servisno orijentisane arhitekture. Preporučuje se da svaki kontejner pokrene jedan proces ili aplikaciju. Na taj način se podstiče distribucija aplikacije, gdje se servis ili aplikacija predstavlja nizom međusobno povezanih kontejnera. Ovo će olakšati skaliranje, distribuciju i elimisanje grešaka kod aplikacija.
- Efikasan i brz životni ciklus razvoja – Dockerov cilj jeste da smanji vrijeme ciklusa između koda koji se testira, piše, koristi i implementira, tj. cilj je da aplikacije budu lake za izradu.



Slika 9: *Šema Docker pokretača*[38]

Server preko upravljačke linije upotrebljava REST API programski servero kako bi mogao da upravlja Docker pozadinskim procesima pomoću skripti ili direktnim unošenjem naredbi putem servera preko upravljačke linije. U slučaju spoljašnjih aplikacija koje izvršavaju naredbe u Dockeru, one to obavljaju na način da se direktno spoje na aplikacijsko programski server. Docker se sastoji od četiri glavne komponente, a to su:

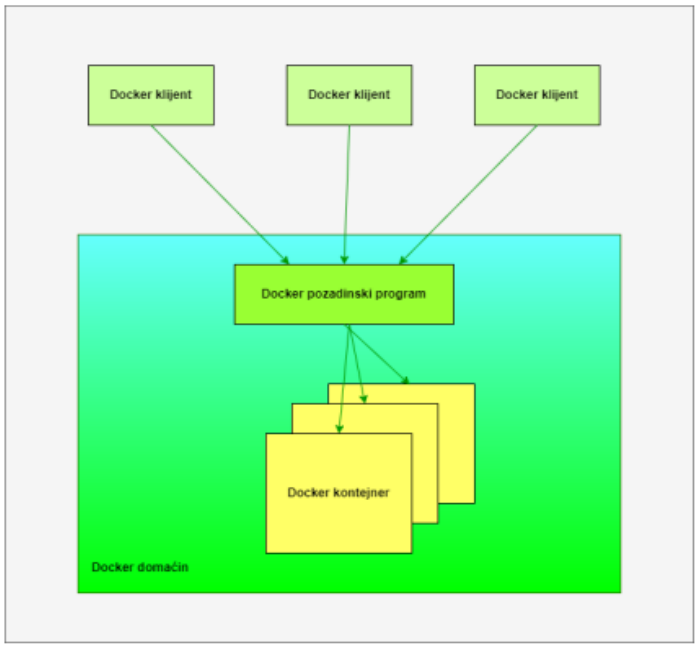
- Docker klijent i server;
- Docker *image*;
- Registri;
- Docker kontejneri.

3.5.1. Docker klijent i server

Docker je aplikacija zasnovana na klijent-server arhitekturi. Docker klijent komunicira sa Docker pozadinskim programom (*Daemon-om*) ili serverom, koji potom obavlja sav posao. Klijent i

[38] Docker. 2019. *Docker Overview*. https://docs.docker.com/get-started/overview/. posjeta: 16.04.2024

pozadinski program mogu biti pokrenuti na istom *host-u*, ali je takođe moguće da se lokalni Docker klijent poveže sa pozadinskim programom na udaljenom *host-u*..



Slika 6: *Arhitektura Dockera*[39]

3.5.2. Docker slike (*Image*)

Docker *image* predstavljaju glavni „građevinski“ blok kod Dockera. Iz Docker *image* -a se pokreću kontejneri. Sami *image* predstavlja dio životnog ciklusa Dockera koji je poznat pod imenom *Build*. Predstavljaju slojeviti format, koji upotrebljava Uninon konfigiuracioni sistem, a koji je napravljen korak po korak tako što se koristi niz upustava. Zbog održavanja konfigiuracionih sistema, ponekad je logično da se pojedine datoteke drže na odvojenim lokacijama, ali samim korisnicima moraju da se predstave kao jedna cjelina. Upravo ovako nešto omogućava Union konfigiuracioni sistem koji dozvoljava da datoteke budu fizički na raznim lokacijama, ali su logički spojene u jednu cjelinu. Logički skup spojenih foldera poznat je pod imenom unija, a svaki fizički direktorijum se još zove i grana, pa iz tog razloga Docker ima dvije prednosti. Prva prednost jeste da ne postoji potreba za dupliranjem cijelog seta datoteka svaki put kada se pokreće i kreira novi Docker kontejner. Upravo zbog ovog kreiranje Docker kontejnera jeste brz postupak i ne zauzima velike resurse. Druga prednost jeste ta što su slojevi odvojeni, a ako je neophodno da se naprave izmjene na nekom sloju i kasnije da se ažuriraju na pojedinim Docker kontejnerima, ažuriraće se onaj sloj koji je promijenjen, a tako nešto će u velikoj mjeri olakšati i ubrzati navedeni proces.

[39] Turnbull, James: *The Docker Book*, Turnbull Press, New York, 2014, p. 10.

Docker *image-i* kontejnera mogu da se posmatraju kao klase u objektno orijentisanom programiranju. Klase se koriste kao nacrti za instanciranje velikog broja objekata, a samim tim imaju i mogućnost nasleđivanja drugih klasa kako bi se na temelju već postojećih klasa proširivanjem istih dobile prilagođene klase.[40] Docker *image*-i mogu da se smatraju izvornim kodom za kontejnere. Prenosivi su i mogu da se ažuriraju, dijele i skladište. Mogu da se primijene četiri naredbe: *expose, from, entrypoint* i *add*. *From* predstavlja osnovnu *image*-a, *expose* predstavlja vrata (*Port*), *entrypoint* daje argumente i naredbe za izvršni kontejner, a *add* kopira direktorijume i datoteke u kontejner.

3.5.3. Registri

Docker skladišti sve *image-e* koje se izgrade u registre. Postoje dvije vrste registara: privatni i javni. Docker Inx. Upravlja javnim registrom za *image* čije je ime Docker Hub.

Docker Hub ima više desetina hiljada *image*-a koji su različiti programeri podijelili i napravili. Isto tako sadrži i *image-e* kao što su MySQL baza podataka ili Nginx web server. Kod privatnog registra mogu da se skladište vlastiti *image*-i koji mogu da sadrže izvorni kod ili neke druge vlasničke informacije koje žele da se zaštite ili podijele samo sa nekim osobama.

3.5.4. Docker kontejner

Docker pomaže da se izrade i implementiraju kontejneri u okviru kojih servisi i aplikacije mogu da se smiještaju. Kao što je prethodno navedeno, kontejneri će se pokrenuti iz *image*-a, a mogu da sadrže jedan ili nekoliko pokrenutih procesa. Docker kontejner jeste:

- Izvršno okruženje;
- Format *image-a*;
- Skup standardnih operacija.

Docker, kao primjer, pozajmljuje koncept standardnog brodskog kontejnera, koji se upotrebljava za globalni transport robe, kao model za sve svoje kontejnere. Međutim, umjesto dostavljanja robe, Docker kontejneri šalju softver. Svi Docker kontejneri sadrže *image* programa, kod, pa se nad njim omogućava izvođenje raznih operacija. Može da se stvori, pokrene, zaustavi, ponovo pokrene i na samom kraju uništi. Kao i kod brodskih kontejnera, Dockeru nije važan sadržaj kada obavlja ove operacije, tj. Dockeru nije važno da li je kontejner baza podataka ili web server, on svaki kontejner tretira na identičan način. Isto tako, Dockeru nije važno ni gdje se kontejneri šalju:

[40] Black, Andrew: Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation, vol. 231,* 2013, p. 3 – 20.

može da se gradi na ličnom računaru, da se preuzme na virtuelni i fizički server, da se učita u registar i slično. Baš kao i kod transportnih kontejnera, Dockerov kontejner je prenosiv, zamjenljiv i kompleksan. Postoje i određeni primjeri gdje i za šta Docker može da se primijeni:[41]

- Pokretanje samostalnih aplikacija i servisa kroz veći broj okruženja, koncept koji je izuzetno koristan u arhitekturama koje se orijentišu na servise, ali i implementacije koje se oslanjaju na mikroservise;
- Docker pomaže da izgradnja toka rada i lokalni razvoj aplikacije budu lakši, brži i efikasniji. Lokalno, programeri mogu da dijele, grade i pokreću Docker kontejnere. Kontejneri mogu da se izgrade u razvoju, a zatim da se promovišu u okruženju za testiranje, a na kraju i na produkcionom okruženju.
- Testiranje i izrada složenih arhitektura i aplikacija na lokalnom računaru prije nego što se postavi u produkcionom okruženju;
- Izgradnja višekorisničke infrastrukture Paas (*Platform as a Service*).

3.5.5. Poređenje Docker i Podman tehnologije

Podman se, kao i Docker, bavi kontejnerima, otvorenog je koda, a za razliku od Dockera nema pozadinski program (*Deamonless*). Upotrebljava Linuxov izvorni alat koji olakšava izgradnju, pronalaženje, implementaciju i dijeljenje aplikacija uz pomoć *Open Containers Initiative* (OCI) kontejnera i *image*-a kontejnera. Kontejneri koji se nalaze pod kontrolom Podmana mogu da se pokrenu od strane nekorijenskih ili korijenskih (*Root*) korisnika. Podman upravlja cijelim sistemom kontejnera koji sadrži podove (*Pods*), *image-e* kontejnera i same kontejnere. Isto tako, ima modularan dizajn koji mu omogućava da upotrebljava pojedinačne komponente sistema, samo kada mu tako nešto zatreba.

Tabela 2: Poređenje Dockera i Podmana[42]

	Docker	**Podman**
Pokreće se na	Windows, Linux, macOS	Windows (sa WSL –om), Linux, macOS
Docker – compose	Podržava	Podržava

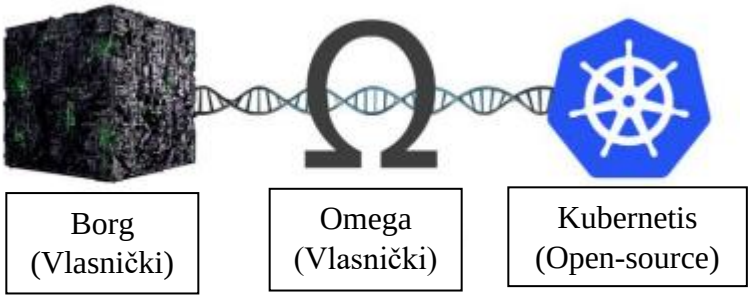
[41] Turnbull, James: *The Docker Book*, Turnbull Press, New York, 2014, p. 12.
[42] Aleksić, Marko: *Podman vs Docker: Everything You Need to Know*, 2022. https://phoenixnap.com/kb/podman-vs-docker. posjeta: 16.04.2024

Docker – swarm	Podržava	Ne podržava
Monolitna platforma	Da	Ne
Slike (*image*)	Može da izgradi *image* za kontejner	Koristi Buildah za izgradnju *image*-a
Root	Kontejnere mogu da pokrenu samo Root korisnici	Kontejnere mogu da pokrenu Root ili ne Root korisnici
Pozadinski program	Koristi pozadinski program	Posjeduje arhitekturu bez pozadinskog programa

Na osnovu prethodne tabele, može da se zaključi da razlika između Podmana i Dockera jeste ta što Podman ne upotrebljava pozadinski program, a kod Podmana pored Root korisnika, kontejnere mogu da pokrenu i korisnici koji nisu Root. Sa druge strane, Docker sam može da izgradi *image* za kontejner, a Podman mora da upotrebi Buildah da bi izgradio *image*. Docker jeste monolitna platforma koja podržava Docker – swarm, alat za raspoređivanje i grupisanje kontejnera. Ovim platformama je zajedničko to da obije podržavaju Docker – compose, alat koji je napravljen za pomoć prilikom dijeljenja i definisanja aplikacija sa nekoliko kontejnera, a obije platforme mogu da se pokrenu na Windows–u, Linux–u i macOS–u.

3.6. Kubernetes tehnologija

Kubernetes je razvijen od strane Google-a i trenutno ga održava CNCF (*Cloud Native Computing Foundation*). Temelji Kubernetes-a potiču iz iskustava stečenih radom sa sistemima kao što su „Borg“ i „Omega“, koje je Google koristio prije nego što je 2014. godine predstavio Kubernetes kao otvoreni izvorni projekat. Kubernetes je napisan na programskom jeziku Go i koristi se za efikasno skaliranje, implementaciju i upravljanje kontejnerskim aplikacijama u data centrima u cloudu. Kao otvoreni izvor, Kubernetes može biti prilagođen i korišten kao osnova za razvoj drugih platformi. Iako se često koristi za orkestraciju kontejnera, može se shvatiti i kao alat za orkestraciju aplikacija, naročito kontejnerizovanih mikroservisnih aplikacija u Cloud okruženju.



Slika 10: *Razvoj sistema Kubernetes*[43]

Orkestrator jeste sistem koji postavlja i upravlja aplikacijama. Može da implementira same aplikacije i na dinamički način da reaguje na promjene. Na primjer, Kubernetes može:[44]

- Dinamički da smanji i poveća aplikaciju po potrebi;
- Da postavi (*Deploy*) aplikaciju;
- Da izvede ažuriranja i da vrati funkcionalno stanje bez prekida rada;
- Da samoizliječi (*Self – healing*) kada dođe do određenog kvara.

Ono što je najinteresantnije jeste to da Kubernetes sve ovo može da izvršava bez nadzora i bez da se neka osoba uključi u njegov rad. Na početku je neophodno da se podese željene konfiguracije, a potom će ostatak Kubernetes sam odraditi. Na najvišem nivou, Kubernete može da bude orkestrator za izvorne mikroservisne aplikacije u Cloud-u ili klaster za pokretanje aplikacija. Kao klaster, on predstavlja grupu čvorova (*Nodes*) i kontrolni *plane* (*Control Plane*). *Nodes* predstavljaju mjesta gdje se pokreću aplikacijski servisi. Sa druge strane, *Control Plane* se smatra „mozgom“ jer implementira sve bitne faktore, kao što su ažuriranja bez prekida rada i automatsko ažuriranje. Kao orkestrator, Kubernetes predstavlja sistem koji se brine o upravljanju i postavljanju aplikacija. Kako bi aplikacija mogla da se pokrene uz pomoć Kubernetesa, neophodno je prvo da se napravljena aplikacija kontejnerizuje, a potom da se ovaj kontejner preda Kubernetes klasteru. Sam klaster se sastoji od jednog ili nekoliko glavnih čvorova (*Head Nodes*) i većeg broja običnih *Nodes*-a. *Head Nodes* kontroliše cijeli klaster, a to znači da snosi odgovornost za izvršavanje nadzora, planiranje odluka, odgovaranje na događaje i implementiranje promjena.

Iz tog razloga često se dešava da se *Head Nodes* poistovjećuje sa *Control Plane*-om. Obični *Nodes* predstavljaju mjesta gdje se aplikacije pokreću, pa su zato još poznate pod imenom *Dana Plane*. Svaki od njih je povezan sa *Head Nodes*, pa ga samim tim izvještavaju o svom stanju i

[43] Hamori, Ferenc: *The History of Kubernetes on Timeline*, 2023. https://blog.risingstack.com/the-history-of-kubernetes/. posjeta: 16.04.2024
[44] Poulton, Nigel; Joglekar, Pushkar: *The Kubernetes Book*, Leanpub, Victoria, 2020, p. 27.

konstantno čekaju nove zadatke. Da bi aplikacija mogla da se pokrene na Kubernetesu, neophodno je sledeće:

- Aplikacija treba da se zapiše kao mali nezavisni mikroservis u nekom programskom jeziku;
- Svaki mikroservis treba da se zapakuje u kontejner;
- Svaki kontejner treba da se zapakuje u svoj *Pod*;
- Svi *Pods* treba da se implementiraju u klastere pomoću kontrolora višeg nivoa, kao što su, na primjer, *StatefulSets, Deployment, DaeomnSets* i slično.

Head Nodes predstavlja kolekciju sistemskih servisa koji čine *Control Plane* klastera. Sistemski servisi mogu biti: *Cluster store*, API server, kontroler i planer. API server jeste centralna, glavna stanica Kubernetesa. Komunikacija između svih komponenti, bilo da je riječ o unutrašnjim ili spoljašnjim komponentama, mora da prođe kroz API server. Sam server izlaže RESTful API preko HTTPS–a, a on se definiše u YAML konfiguracijskoj datoteci. YAML datotete posjeduju željeno stanje aplikacije, a pod ovim stanjem se smatraju *image*-i kontejnera koji će da se koriste, zatim koji mrežni priključci treba da se izlože, kao i koliko *Pod* replika treba da se pokrene. Svi zahtjevi prema API serveru podliježu provjerama, uz pomoć autorizacije i provjere autentičnosti, a nakon što je to izvršeno, provjerava se konfiguracija u YAML datoteci, zadržava u skladištu klastera, a na samom kraju se postavlja na klaster. *Control Plane* trajno skladišti i prati stanje cijele konfiguracije, ali i stanje samog klastera. Iz tog razloga je bitan dio klastera, pošto bez skladištenja neće biti ni klastera.

Skladištenje klastera se trenutno temelji na *etcd*, popularnoj distribuiranoj bazi podataka. Kada je riječ o dostupnosti, *etcd* preferira dosljednost u odnosu na dostupnost. Drugim riječima, to znači da će se zaustaviti ažuriranje klastera po potrebi kako bi mogla da se održi dosljednost. Ali, ukoliko *etcd* postane nedostupan, aplikacije koje se na klasteru izvode morale bi da nastave da rade, samo ništa neće moći da se ažurira.

Kao i kod svih distribuiranih baza podataka, veoma je važna dosljednost upisivanja u bazu podataka. *Controller manager* implementira sve pozadinske kontrolne petlje koje odgovaraju na događaje i nadziru klaster. Kontrolne petlje uključuju: kontroler grupe replika, kontroler čvora i kontroler krajnjih tačaka. Svi oni rade kao pozadinska petlja za posmatranje koja konstantno prati API server zbog promjena.[45] Glavni cilj jeste da se osigura da trenutno stanje klastera u potpunosti odgovara željenom stanju.

Logika koju implementira svaka kontrolna petlja može biti sledeća:

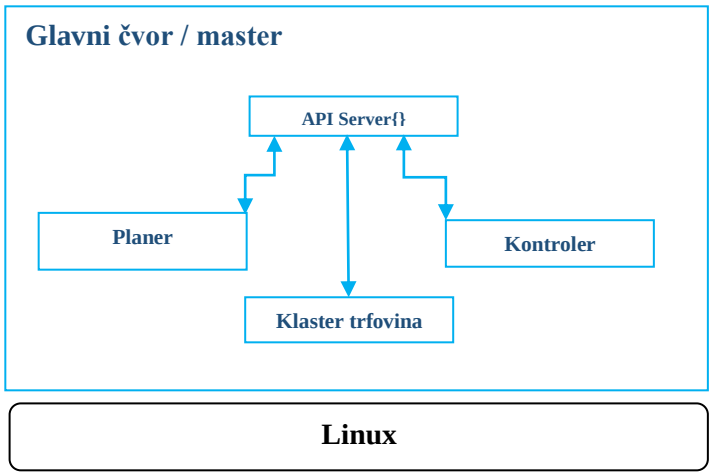
- Postiže se željeno stanje;

[45] Poulton, Nigel; Joglekar, Pushkar: *The Kubernetes Book*, Leanpub, Victoria, 2020, p. 29.

- Posmatra se trenutno stanje;
- Određuju se razlike.

Svaka kontrolna petlja je zainteresovana i specijalizovana samo za svoj zadatak Kubernetes klastera i ne vodi računa o drugim kontrolnim petljama. Ovako nešto je bitno za distribuirani dizajn Kubernetesa i pridržava se Unix–ove filozofije izgradnje složenih sistema od manjih specijalizovanih djelova. Planer posmatra API server tako što traži nove radne zadatke, a potom se ti zadaci dodjeljuju određenim zdravim čvorovima.

U pozadini se implementira složena logika, koja filtrira same čvorove koji ne mogu da izvrše zadatak, a potom rangira čvorove koji su sposobni. Tokom identifikacije čvorova koji mogu da pokrenu zadatak, planer će izvesti razne provjere. Zanemaruju se svi čvorovi koji ne mogu da pokrenu zadatak, dok će svi drugi da se rangiraju na sledeći način: koliko čvor ima slobodnih resursa, da li čvor ima neophodan *image*, koliko zadataka se već izvodi od strane svakog čvora. Ukoliko *plane* ne može da pronađe odgovarajući čvor, zadatak ne može da se rasporedi i označiće se kao da je na čekanju. Planer ne snosi nikakvu odgovornost za pokretanje zadataka, nego se koristi za izbor čvorova na kojima će da se izvodi zadatak.



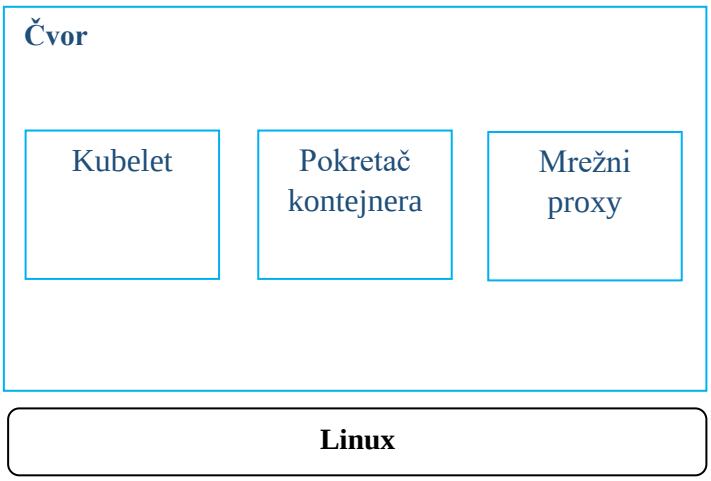
Slika 11: *Prikaz Head Node (glavnog čvora) klastera*[46]

Drugim riječima, čvorovi mogu da se definišu kao radnici u Kubernetes klasteru, pa sami tim njihovi zadaci su sledeći:

- Daju izvještaj o nivou kontrole (pomoću API servera);
- Prate API server za nove radne zadatke;
- Izvršavaju nove radne zadatke.

[46] Poulton, Nigel; Joglekar, Pushkar: *The Kubernetes Book*, Leanpub, Victoria, 2020, p. 128.

Na Slici 12. je prikazano da je čvor (*Node*) klastera u suštini jednostavnije građe od *Head Nodes*. On se sastoji od tri elementa: mrežni proxy, kubelet i okruženje za izvršavanje kontejnera (*Container runtime* – CRI).



Slika 12: *Prikaz čvora (Node) klastera*[47]

Kubelet predstavlja osnovni Kubernetes agent i radi na svakom čvoru koji se u klasteru nalazi. Kada se klasteru pridruži neki novi čvor, od strane procesa se instalira kubelet na čvor. Kubelet snosi odgovornost za registrovanje čvora na klasteru. Registracija na efikasan način spaja memoriju, procesor i skladištenje čvora u jedan širi skup klastera. Jedan od najvažnijih zadataka kubeleta jeste posmatranje API servera za nove zadatke. Ukoliko kubelet ne može da pokrene neki zadatak, javiće glavnom uređaju, a potom će dopustiti kontrolnom nivou da odluči koje radnje trebaju da se preduzmu. Vrijeme izvođenja kontejnera jeste komponenta čvora koja ima zaduženja za zadatke koji su povezani sa kontejnerima, a tu se prvenstveno misli na: zaustavljanje i pokretanje kontejnera, povlačenje *image*. C*ontainer runtime* maskira unutrašnju mašineriju Kubernetisa i izlaže čisto dokumentovan interfejs na koji može da se priključi *Runtime* (vrijeme izvođenja) kontejnera treće strane.

Kada je riječ o mrežnom proxy–ju, on radi na svakom čvoru u klasteru i ima odgovornost za lokalno umrežavanje klastera. Mrežni proxy osigurava da svaki čvor treba da dobije ličnu jedinstvenu IP adresu, a potom implementira lokalna IPVS ili IPTABLES pravila za obrađivanje usmjeravanja (*Routing*), kao i balansiranje opterećenja prometa na Pod mreži.

[47] Poulton, Nigel; Joglekar, Pushkar: *The Kubernetes Book*, Leanpub, Victoria, 2020, p. 128.

3.6.1. Poređenje Kubernetes i drugih tehnologija

Tehnologije koje će biti upoređene sa Kubernetes–om su: Docker Swarm, Apache Mesos i Nomad. Kada je riječ o Docker Swarm, on predstavlja platformu za orkestraciju kontejnera otvorenog koda, koju je napravio i koju održava Docker. Docker Swarm pretvara nekoliko Docker instanci u jedan virtuelni računar. Docker Swarm klaster ima tri stavke:

- Balanser opterećenja;
- Čvorove;
- Zadatke i službe.

Čvorovi predstavljaju individualne instance Docker mehanizma koji kontrolišu klaster, a samim tim i upravljaju kontejnerima koji se upotrebljavaju kako bi se pokrenuli zadaci i servis. Takođe, Docker Swarm klasteri uključuju balansiranje opterećenja za usmjeravanje zahtjeva preko čvorova. Sa druge strane, Apache Mesos predstavlja projekat otvorenog koda za upravljanje računarskim klasterima razvijen je na Kalifornijskom Univerzitetu koji se zove Berkeley. Napravljen je upotrebom identičnih principa kao Linux jezgro, samo na različitom nivou apstrakcije. Mesos kernel radi na svim računarima i pruža aplikacije sa API–jima za upravljanje resursima, kao i raspoređivanje u cijelom data centru i okruženjima Cloud–a.

Nomad jeste program koji se koristi kao upravitelj klastera i planera koji su dizajnirani za mikroservise i grupna radna opterećenja koje je razvila kompanija HashiCorp, čije je sjedište u San Francisku (SAD). Nomad je visoko dostupan, distribuiran i skaliran na hiljade čvorova koji obuhvataju nekoliko regija i data centara.[48]

Tabela 3: Poređenje alata za orkestraciju kontejnera[49]

Funkcija	Nomad	Apache Mesos	Kubernetes	Docker Swarm
Podržava kontejnere	LXC, Rkt, Docker	Rkt, Docker	Rkt, Docker	Docker
Otkrivanje servisa	Program treće strane (*Third party-Consul)*	Da	Da	Da

[48] Ozmen, Huseyin: *Design and implementation of an Iot-based home automation system utilizing fog and cloud computing paradigms*, Bogazici Univeristy, Istanbul, 2018, p. 49.
[49] Ozmen, Huseyin: *Design and implementation of an Iot-based home automation system utilizing fog and cloud computing paradigms*, Bogazici Univeristy, Istanbul, 2018, p. 49.

Repliciranje glavnog čvora	Da	Program treće strane (*Zookeeper*)	Da	Da
Balansiranje opterećenja	Program treće strane (*Consul)*	Program treće strane (*Marathon*)	Podržava spoljašnje balansiranje	Podržava spoljašnje balansiranje
Automatsko skaliranje	Ne	Da	Da	Ne
Praćenje kontejnera	Da	Program treće strane	Program treće strane	Program treće strane
Bilježenje (*Logging*) kontejnera	Da	Da	Program treće strane (ELK)	Program treće strane (ELK)
Upravljanje tajnama (*Secrets*)	Program treće strane (*Vault)*	Ne	Da	Da

Rkt (ili *Rocket*) kontejneri predstavljaju alternativu Dockeru, koju je razvila CoreOS kompanija. Rkt kontejneri fokusiraju se na jednostavnost i sigurnost, a za razliku od Dockera, ne zavise od centralizovanog demona (*Docker Engine*), već su dizajnirani da rade bez pozadinskog procesa, što omogućava veću fleksibilnost i sigurnost u određenim okruženjima.

LXC (*Linux Containers*) su kontejneri koji koriste Linux jezgro i njegove mogućnosti za izolaciju procesa i resursa, kao što su mreža, memorija i CPU. LXC omogućava kreiranje laganih, ali potpuno funkcionalnih Linux okruženja, slično virtuelnim mašinama, ali sa manjim troškovima resursa.

Na osnovu tabele može da se zaključi da su svi alati veoma slični. Docker Swarm podržava samo Docker kontejnere, dok svi ostali, pored Docker kontejnera, podržavaju i Rkt kontejnere, a Nomad uz sve njih podržava i LXC kontejnere. Svi alati mogu da otkriju svoje servise, dok Nomad za tako nešto upotrebljava treće strane. Kada je riječ o repliciranju glavnog čvora, svi alati mogu da ga repliciraju, međutim Apache Mesos za ovako nešto upotrebljava program pod imenom *Zookeeper*. Kubernetes i Docker Swar podržavaju spoljašnje balansiranje, Nomad upotrebljava program pod imenom *Consul,* a Apache Mesos upotrebljava program *Marathon.*

Kada je riječ o automatskom skaliranju, njega imaju samo Apache Mesos i Kubernetes, dok Nomad posjeduje samo praćenje kontejnera, pri čemu svi drugi upotrebljavaju za to program treće strane. Svi alati mogu da bilježe zapise kontejnera, dok Kubernetes i Docker Swarm u ovom slučaju upotrebljavaju program ELK. Nomad za upravljanje tajnama upotrebljava program *Vault*, dok Apache Mesos jedini ne može da upravlja tajnama (*Secrets*).

3.7. Razlozi za izbor alata za kontejnerizaciju

Pri izboru alata za orkestraciju kontejnera, nekoliko faktora treba uzeti u obzir:

- Podrška za specifične tipove kontejnera: Neki alati podržavaju samo određene kontejnere (npr. *Docker Swarm* podržava samo Docker kontejnere), dok drugi pružaju širu podršku (kao što su *Nomad* i *Apache Mesos* koji podržavaju *Rkt* i *LXC* kontejnere). Izbor alata može zavisiti od toga koje vrste kontejnera su već u upotrebi u sistemu.
- Automatsko skaliranje i upravljanje resursima: Alati kao što su *Kubernetes* i *Apache Mesos* pružaju napredne mogućnosti automatskog skaliranja, što može biti presudno za sisteme sa velikim varijacijama u opterećenju.
- Integracija sa alatima trećih strana: Mogućnost povezivanja sa alatima za balansiranje opterećenja, praćenje kontejnera i upravljanje tajnama može uticati na izbor. Na primjer, Nomad koristi *Consul* za balansiranje opterećenja i *Vault* za upravljanje tajnama.
- Jednostavnost upotrebe i administracije: *Docker Swarm* je često preferiran zbog jednostavne upotrebe i lake integracije sa Docker okruženjem, čineći ga dobrim izborom za manje i srednje projekte. *Kubernetes*, s druge strane, pruža veće mogućnosti, ali dolazi sa složenijom administracijom.

U ovom radu je odabran Docker zbog njegove široke prihvaćenosti i dugogodišnjeg liderstva u oblasti kontejnerizacije. Docker se ističe jednostavnošću u instalaciji i upotrebi, što ga čini pristupačnim kako početnicima, tako i iskusnim korisnicima. Pored toga, Docker ima veliku i aktivnu zajednicu korisnika, što olakšava pronalaženje dokumentacije, tutorijala i tehničke podrške, što ga čini idealnim za orkestraciju kontejnera uz alate za skaliranje aplikacija od kojih je najpoznatiji *Docker Swarm*. Njegova integracija sa *Docker Swarm*-om omogućava lako skaliranje aplikacija, što je ključno za dinamične sisteme sa mikroservisnom arhitekturom. *Docker Swarm* je, uz to, fleksibilan i dovoljno robustan za orkestraciju kontejnera u različitim okruženjima, pružajući optimalnu ravnotežu između performansi i jednostavnosti administracije, što ga čini idealnim za potrebe ovog rada.

Tabela 4: Tržišni udio za vodeće tehnologije kontejnerizacije

Rang	Technologija	Procijenjeni tržišni udio (%)
1	Docker	83.03%
2	Kubernetes Engine (GKE)	12.17%
3	LXC (Linux Containers)	1.15%

Docker dominira tržištem kontejnerizacije sa ogromnim udjelom od 83.03%. Slijedi Kubernetes Engine sa 12.17%, dok su ostale tehnologije poput LXC daleko manje zastupljene, sa 1.15% udjela na tržištu. Ovi podaci pokazuju da je Docker najrasprostranjenija tehnologija, dok Kubernetes ima značajan udio kao orkestrator kontejnera.

4. ARHITEKTURA APLIKACIJE

4.1. Detaljan opis aplikacije i arhitektura mikroservisa

Aplikacija je osmišljena kao platforma za plasman domaćih poljoprivrednih proizvoda, koristeći modernu arhitekturu baziranu na mikroservisima i kontejnerizaciji. Sastoji se od tri ključna dijela, od kojih svaki predstavlja zaseban mikroservis:

1. Korisnički portal (UI) – Implementiran pomoću Vue.js, ovaj dio omogućava krajnjim korisnicima pristup aplikaciji, pregled proizvoda, naručivanje i upravljanje narudžbinama. Krajnji korisnici mogu pretraživati proizvode po kategorijama, dodavati ih u korpu i kreirati narudžbine koje se šalju prodavcima.
2. Sistem za upravljanje sadržajem (CMS) – Izrađen u Laravelu, CMS pruža administrativni interfejs za upravljanje sadržajem aplikacije. Administrator može kreirati i upravljati kategorijama proizvoda, korisnicima i oglasima. Korisnici, koji su proizvođači, koriste CMS za kreiranje i upravljanje oglasima te pregled narudžbina.
3. Sistem za naručivanje – Ovaj mikroservis takođe koristi Laravel i služi za upravljanje cijelim procesom naručivanja. Omogućava korisnicima da kreiraju narudžbine, a prodavcima da te narudžbine prihvataju ili odbijaju. Povezan je sa CMS-om i korisničkim interfejsom kako bi omogućio efikasnu interakciju između različitih komponenti aplikacije.

Aplikacija koristi Docker za kontejnerizaciju svih mikroservisa, čime se obezbjeđuje izolovano i konzistentno okruženje za svaki servis. Svaki mikroservis je smješten u zaseban Docker kontejner, omogućavajući lako upravljanje, skaliranje i održavanje aplikacije.

Projektni zadatak

Cilj projekta je razviti modernu, skalabilnu aplikaciju koja omogućava korisnicima (kupcima) da pretražuju, pregledaju i kupuju domaće poljoprivredne proizvode, dok proizvođači imaju mogućnost kreiranja i upravljanja oglasima putem centralnog sistema. Aplikacija treba da omogući:

- Jednostavan korisnički portal: Korisnici mogu lako pretraživati proizvode, dodavati ih u korpu i izvršavati narudžbine.

- Efikasno upravljanje sadržajem: Proizvođači mogu kreirati oglase, a administratori mogu upravljati korisnicima i kategorijama proizvoda.
- Automatizovan proces naručivanja: Kupci mogu kreirati narudžbine, a proizvođači primati obavještenja o narudžbinama putem email-a i potvrđivati ih ili odbijati.
- Mikroservisna arhitektura: Sistem mora biti modularan, kako bi se osigurala fleksibilnost, održivost i mogućnost skaliranja.
- Kontejnerizacija sa Dockerom: Svaki dio aplikacije treba biti kontejnerizovan radi izolacije i lakšeg upravljanja.

Zavisnosti servisa

Zavisnosti servisa se odnose na međusobnu povezanost i komunikaciju između mikroservisa i drugih djelova sistema. Mikroservisne aplikacije zavise od:

1. Korisnički portal (UI) – Zavisnost ovog servisa je REST API koji dolazi iz CMS-a i sistema za naručivanje. Vue.js kao frontend framework komunicira sa backendom putem HTTP zahtjeva, što omogućava slanje i primanje podataka o oglasima i narudžbinama.
2. CMS (Laravel) – CMS je ključan za upravljanje podacima u aplikaciji i zavisi od baze podataka (MySQL). Takođe, CMS komunicira sa UI mikroservisom, šaljući podatke o korisnicima, kategorijama i oglasima putem REST API-ja.
3. Sistem za naručivanje (Laravel) – Ovaj servis zavisi od baze podataka za čuvanje informacija o narudžbinama. On komunicira sa UI i CMS mikroservisom radi sinhronizacije podataka o narudžbinama i statusu kupovine.

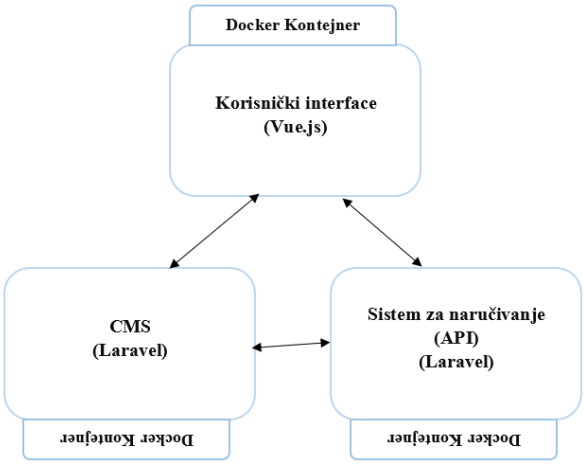
Šematski prikaz servisa i kontejnera

Na Slici 13 predstavljen je šematski prikaz servisa i Docker kontejnera koji čine aplikaciju. Sastoji se od tri ključne komponente, svaka smještena unutar svog Docker kontejnera:

- Korisnički portal: Ova komponenta predstavlja frontend aplikacije, koji korisnici koriste za interakciju. Povezana je sa dva backend servisa, CMS-om i sistemom za naručivanje, za komunikaciju i prikaz podataka.

- CMS : Ovaj servis služi za upravljanje sadržajem i povezan je sa korisničkim interfejsom. CMS obrađuje podatke i funkcije koje su potrebne za prikaz i rad sa oglasima, kategorijama, i drugim aspektima sadržaja aplikacije.
- Sistem za naručivanje : Ova komponenta pruža funkcionalnosti za upravljanje narudžbinama putem API-ja. Takođe, povezana je sa CMS-om za sinhronizaciju određenih podataka i funkcija potrebnih za sistem naručivanja.

Svaka od ovih komponenti je smještena unutar zasebnog Docker kontejnera, što omogućava lako skaliranje, nezavisnu konfiguraciju i održavanje svake komponente aplikacije. Docker omogućava da ovi servisi međusobno komuniciraju kroz mrežu, dok je aplikacija modularna i efikasna za razvoj i produkciju.



Slika 13: *Šematski prikaz servisa i kontejnera*

Označavanje kontejnera i mikroservisa

Svaki mikroservis ima svoj nezavisan Docker kontejner, kako bi se olakšalo upravljanje aplikacijom i skaliranje njenih djelova.

- Kontejneri:
  agro-front – Ovaj kontejner sadrži Vue.js aplikaciju koja predstavlja korisnički portal.
  agro-trade-cms – Kontejner u kojem se nalazi Laravel CMS, odgovoran za upravljanje sadržajem aplikacije.
  rural-shop – Kontejner za sistem naručivanja, takođe izrađen u Laravelu.

- Mikroservisi:

UI mikroservis – Odgovoran za prikaz aplikacije i korisničku interakciju. Koristi REST API za komunikaciju sa CMS-om i sistemom za naručivanje.

CMS mikroservis – Implementira funkcije za upravljanje korisnicima, kategorijama i oglasima. Komunicira sa UI-jem i sistemom za naručivanje.

Sistem za naručivanje – Upravljanje narudžbinama, slanje notifikacija putem email-a, te interakcija sa CMS-om i korisničkim interfejsom.

Razvoj aplikacije podrazumijevao je kontejnerizaciju svakog mikroservisa putem Dockerfile-a, a zatim njihovo povezivanje putem Docker Compose alata. Detaljan opis razvoja svakog mikroservisa dat je u nastavku:

1. UI mikroservis (Vue.js):

   - Vue.js je korišćen za kreiranje jednostavnog i responzivnog korisničkog interfejsa.
   - UI je kontejnerizovan pomoću Dockerfile-a, koji definiše sve potrebne zavisnosti (Node.js, NPM,) i omogućava lako pokretanje u izolovanom kontejneru.
   - UI komunicira sa backend mikroservisima putem REST API-ja.

2. CMS mikroservis (Laravel):

   - Laravel je odabran zbog svoje jednostavne sintakse, modularnosti i podrške za rad sa bazama podataka.
   - Za CMS mikroservis kreiran je Dockerfile, u kojem su definisane zavisnosti (PHP 8.2, Composer, MySQL). Ovaj Dockerfile osigurava izolovano okruženje za Laravel aplikaciju.
   - CMS komunicira sa MySQL bazom podataka, gdje se čuvaju svi podaci o korisnicima, oglasima i kategorijama.
   - Docker Compose koristi se za orkestraciju CMS-a i povezivanje sa drugim mikroservisima.

3. Sistem za naručivanje (Laravel):

   - Ovaj mikroservis upravlja narudžbinama. Kreiran je pomoću Laravel frameworka, a njegovo okruženje definisano je u Dockerfile-u.

- Laravel omogućava lak pristup bazi podataka i implementaciju REST API-ja za komunikaciju sa UI i CMS mikroservisima.
- Docker Compose koristi se za automatizovano upravljanje kontejnerima, kao i za povezivanje ovog mikroservisa sa ostalim djelovima aplikacije.

Implementacija i razvoj kroz Docker

Docker je ključni alat u kontejnerizaciji aplikacije. Za svaki mikroservis kreiran je poseban Dockerfile, koji definiše sve potrebne korake za izgradnju kontejnera.

- Dockerfile omogućava standardizovano okruženje za svaki mikroservis:
    1. Instaliraju se sve zavisnosti (npr. za PHP, Node.js);
    2. Kopira se aplikacijski kod u kontejnere.

Izbor tehnologija korišćenih za izradu aplikacije bio je ključan za uspješnu implementaciju svih njenih komponenti.

Zašto Laravel?

Laravel je odabran kao ključna tehnologija za razvoj ove aplikacije zbog svojih brojnih prednosti i mogućnosti koje pruža:

1. Jednostavnost i čitljivost koda: Laravel koristi sintaksu koja je jednostavna za razumijevanje, što omogućava brži razvoj i lakše održavanje koda.
2. Bogata dokumentacija: Laravel nudi obimnu i detaljnu dokumentaciju koja olakšava učenje i upotrebu ovog frejmvorka, čak i za one koji tek počinju raditi sa njim.
3. Snažna zajednica i podrška: Velika zajednica programera koja koristi Laravel pruža podršku i razvija brojne ekstenzije, što dodatno olakšava razvoj aplikacija.
4. Integrisani alat za migracije: Laravel ima ugrađene alate za migraciju baza podataka, što omogućava jednostavno kreiranje i održavanje baza podataka kroz verzije.
5. Napredne funkcionalnosti: Laravel dolazi sa ugrađenim funkcionalnostima kao što su autentifikacija, autorizacija, sistem za upravljanje sesijama, redovima, događajima i notifikacijama, što ubrzava razvoj složenih aplikacija.

6. Modularnost i fleksibilnost: Omogućava modularni pristup razvoju, gdje se različite funkcionalnosti mogu lako dodavati ili mijenjati bez narušavanja cjelokupne strukture aplikacije.
7. Optimizacija performansi: Laravel je optimizovan za performanse, sa alatima za keširanje i brzo izvršavanje upita ka bazi podataka.

Izbor Laravel-a za razvoj ove aplikacije osigurao je stabilnu i lako održivu platformu, koja omogućava brzu i efikasnu implementaciju potrebnih funkcionalnosti.

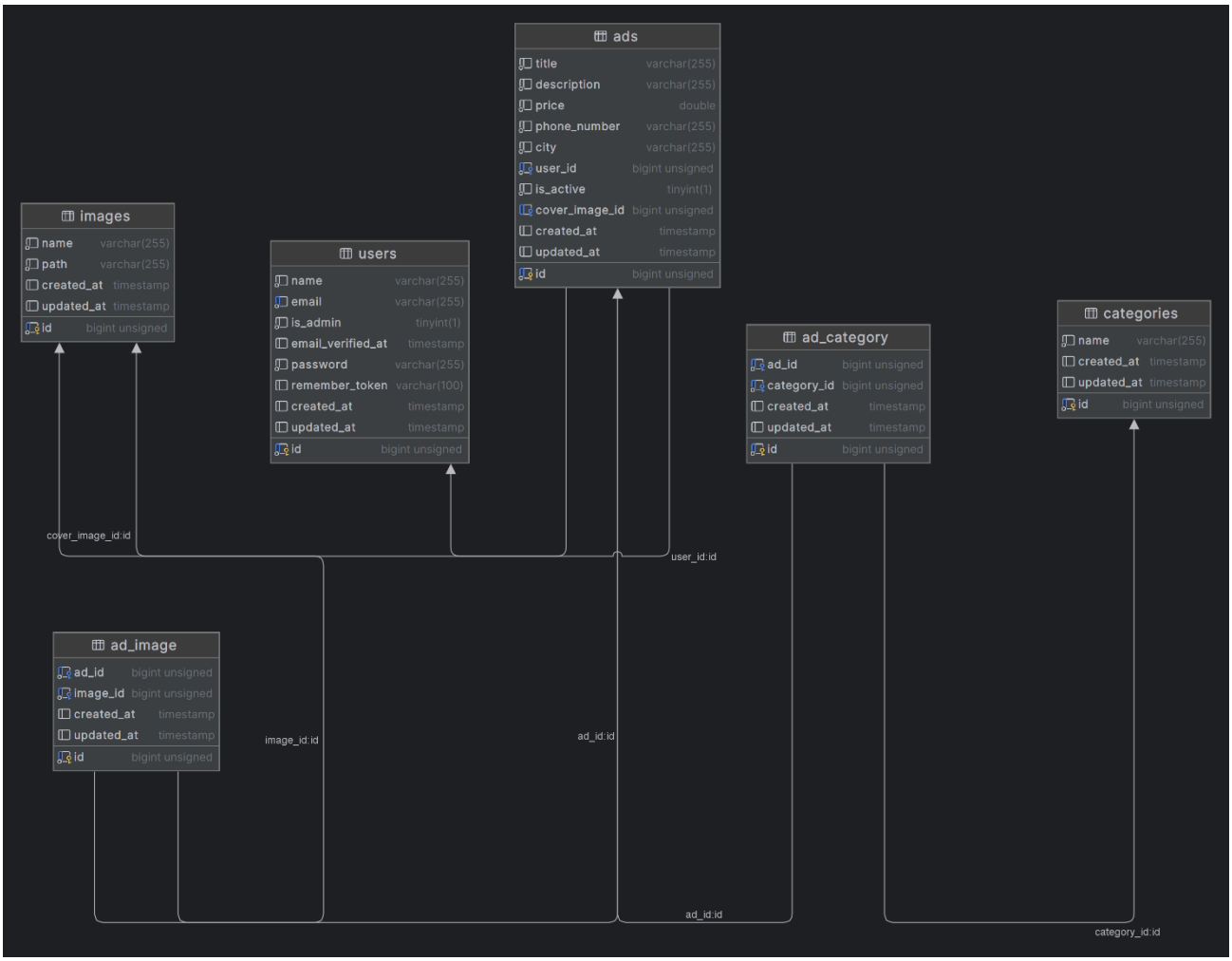
Zašto Vue.js?

Vue.js je odabran za razvoj korisničkog interfejsa zbog sledećih prednosti:

1. Jednostavnost i lakoća učenja: Vue.js je poznat po svojoj jednostavnosti i intuitivnoj sintaksi, što olakšava početak rada i ubrzava proces učenja.
2. Laka integracija sa mikroservisima: Vue.js se lako integriše sa različitim mikroservisima putem REST API-ja ili GraphQL-a, što omogućava fleksibilnu arhitekturu aplikacije.

Za kontejnerizaciju svih mikroservisa, koristio se Docker. Za svaki mikroservis napravljen je poseban, nezavisan kontejner, ukupno njih tri, čiji su *image*-i kreirani putem *Dockerfile*-a. Svaki *Dockerfile* definiše specifične zavisnosti i konfiguracije potrebne za odgovarajući mikroservis. Ovakav pristup omogućava jednostavno upravljanje i skaliranje aplikacije, kao i konzistentno okruženje za razvoj i produkciju. Korišćenjem Docker-a osigurala se izolovanost svakog mikroservisa, što doprinosi većoj stabilnosti i lakšem održavanju cjelokupnog sistema.

4.2. Struktura baze podataka



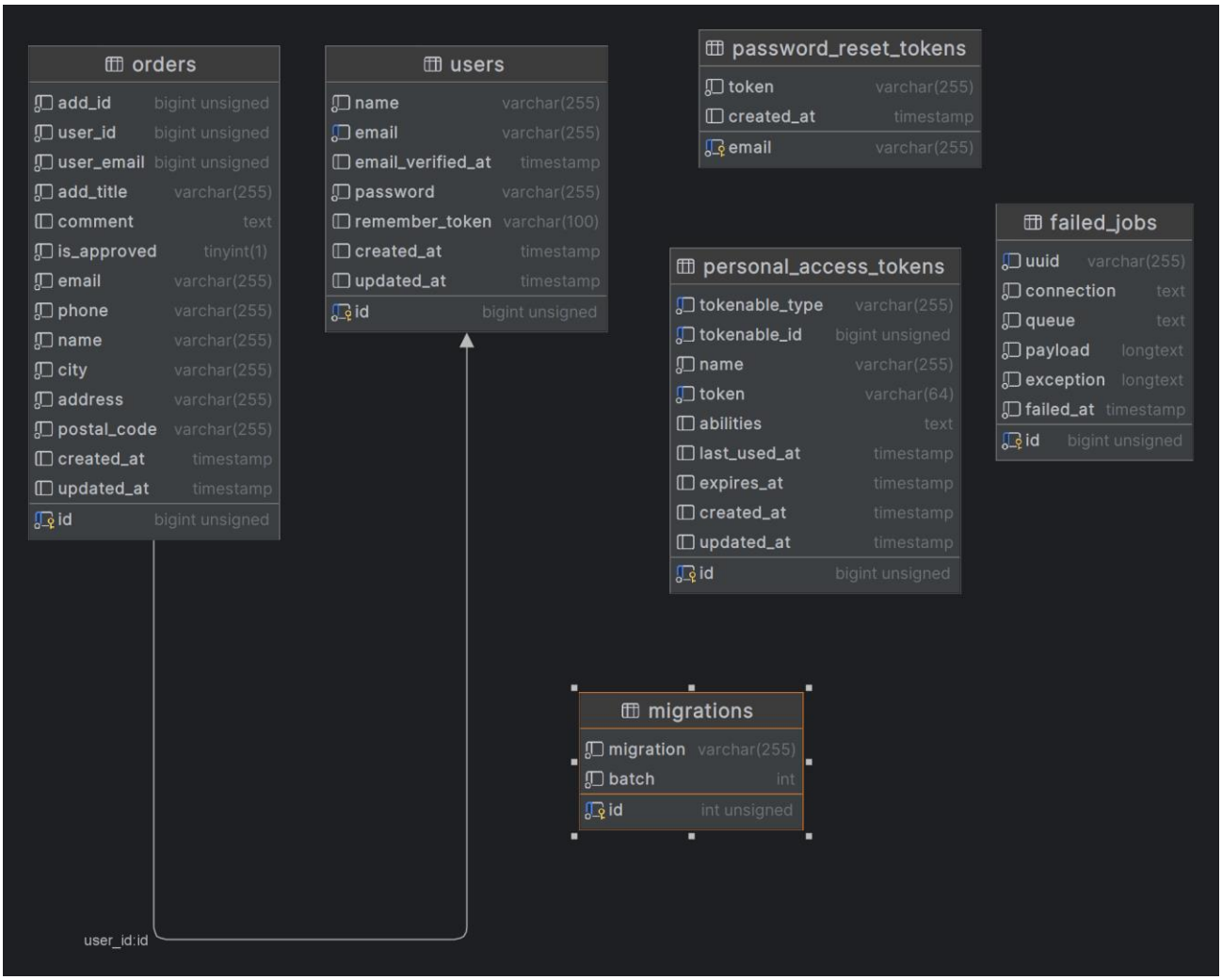
Slika 14: *Struktura baze podataka Sistema za upravljanje sadržajem (CMS)*

Baza podataka Sistema za upravljanje sadržajem (CMS)(Slika 14) koristi nekoliko tabela za upravljanje korisnicima, oglasima i povezanim slikama i kategorijama. Ključne tabele i njihove relacije su:

- Users i Ads: Relacija između korisnika i oglasa je jedan-na-više, što znači da jedan korisnik može kreirati više oglasa, dok svaki oglas pripada samo jednom korisniku. Ovo se postiže putem user_id u tabeli oglasa.
- Ads i Images: Svaki oglas može imati jednu naslovnu sliku, povezanu preko cover_image_id. Takođe, postoji veza između oglasa i više slika preko tabele ad_image, koja omogućava više-na-više relacije između oglasa i slika.

- Ads i Ad_Image: Oglasi mogu imati više slika, a svaka slika može biti povezana sa više oglasa. Ova veza se realizuje preko tabele ad_image, koja povezuje ad_id i image_id.
- Ads i Categories: Oglasi mogu biti povezani sa više kategorija, dok svaka kategorija može sadržati više oglasa. Ova relacija se realizuje preko tabele ad_category, koja povezuje ad_id i category_id.

Ova struktura omogućava korisnicima da kreiraju i upravljaju oglasima, dodaju više slika i dodjeljuju ih različitim kategorijama. Korisnici mogu organizovati oglase na fleksibilan način koristeći ove relacije.



Slika 15: *Struktura baze podataka Sistema za naručivanje*

Baza podataka Sistema za naručivanje (Slika 15) je dizajnirana za upravljanje korisnicima, narudžbinama, tokenima za pristup i praćenje grešaka. Ključne tabele i njihove relacije su:

- Users i Orders: Relacija između korisnika i narudžbina je jedan-na-više, što znači da jedan korisnik može imati više narudžbina, dok svaka narudžbina pripada samo jednom korisniku. Ovo se postiže putem user_id u tabeli narudžbina.
- Users i Password_Reset_Tokens: Svaki korisnik može imati više tokena za resetovanje lozinke, što omogućava više zahtjeva za resetovanje lozinke putem povezanosti sa email adresom korisnika.
- Users i Personal_Access_Tokens: Svaki korisnik može imati više tokena za pristup (API tokena ili sesija), povezani su preko tokenable_id sa korisnikom. Ovi tokeni omogućavaju pristup različitim funkcionalnostima sistema.
- Failed_Jobs: Ova tabela bilježi informacije o neuspešnim poslovima, uključujući UUID, konekciju, grešku i sadržaj (*payload*). Iako nije direktno povezana sa korisnicima, ova tabela pomaže u praćenju i rješavanju grešaka u procesima.

Struktura ove baze omogućava efikasno upravljanje korisnicima, narudžbinama, sigurnosnim tokenima, i praćenje grešaka, kao i vođenje evidencije o migracijama baze podataka.

4.3. Primjena Dockerfile-a

Dockerfile je esencijalni alat u razvoju i implementaciji kontejnerizovanih aplikacija. Ovaj tekstualni dokument omogućava programerima da efikasno definišu korake potrebne za izgradnju Docker kontejnera, pružajući im potpunu kontrolu nad okruženjem, zavisnostima i servisima koji se izvršavaju unutar kontejnera. Iskustvo stečeno tokom rada na razvoju aplikacije govori u prilog tome da je korišćenje *Dockerfile*-a značajno olakšalo proces razvoja softvera, omogućavajući brže konfigurisanje i pokretanje aplikacija u kontejnerizovanom okruženju. Osim što olakšava automatizaciju procesa izgradnje kontejnera, *Dockerfile* omogućava i standardizaciju razvojnog okruženja, čime se olakšava timski rad i upravljanje aplikacijama. Kroz definisanje niza instrukcija, *Dockerfile* omogućava programerima da jasno i precizno dokumentuju svaki korak u procesu izgradnje kontejnera, uključujući instaliranje potrebnih paketa, konfigurisanje okruženja, kopiranje aplikacijskog koda i pokretanje servisa.

U nastavku će biti objašnjen *Dockerfile* za jedan od mikroservisa unutar aplikacije (mikroservis za kreiranje oglasa - CMS, implementiran u Laravelu). Kroz naredne segmente, detaljno će se razmotriti i objasniti svaki dio ovog *Dockerfile-a*. On sadrži konfiguraciju potrebnu za pokretanje ovog mikroservisa, kao i sve zavisnosti i servise koji su mu potrebni za uspješno izvršavanje. Prikazaće se kako je moguće efikasno definisati okruženje, instalirati neophodne pakete, podesiti servise i pokrenuti mikroservis unutar Docker kontejnera.

4.4. Implementacija aplikacije putem Dockerfile-a

Dockerfile je ključni element za automatizovanu izgradnju Docker *image*-a, koji zatim omogućava kontejnerizaciju aplikacije. Proces počinje pisanjem Dockerfile-a, gdje se definišu instrukcije poput izbora osnovnog *image*-a, instalacije zavisnosti, konfiguracije okruženja i pokretanja aplikacije. Nakon kreiranja Dockerfile-a, koristi se komanda *docker build* kako bi se izgradio Docker *image* koji sadrži sve potrebne komponente. Zatim se kontejner pokreće pomoću *docker run* komande, što omogućava aplikaciji da radi unutar kontejnera. U slučaju kada aplikacija sadrži više servisa, Docker Compose se koristi za upravljanje višestrukim kontejnerima, omogućavajući pokretanje svih servisa jednom komandom *docker-compose up*.

U dijelu koda na Slici 16. se prikazuje prvi dio konfiguracije u *Dockerfile-u* mikroservisa koji je sistem za upravljanje sadržajem (CMS), dok ostali *Dockerfile*-ovi postoje na serveru ali neće biti prikazani u samom radu.

U *Dockerfile*-u CMS-a u kojem se postavlja ENV DEBIAN_FRONTEND na *noninteractive*, što osigurava da instalacija paketa neće pokretati interaktivne dijaloge tokom procesa, što je posebno korisno prilikom automatizovane izgradnje Docker kontejnera.

Zatim se definiše verzija PHP 8.2 koja je korišćena u aplikaciji. Dodaju se neophodni koraci za instalaciju PHP 8.2 i njegovih pripadajućih paketa. Prvo se ažurira lista dostupnih paketa (*apt update*), zatim se instalira *software-properties-common*, što će omogućiti upravljanje dodatnim softverskim repozitorijumima. Nakon toga, dodaje se PPA repozitorijum *ondrej/php* kako bi se instalirala PHP 8.2 verzija. Lista dostupnih paketa će biti ponovo ažurirana da bi novi repozitorijum bio uključen, i na kraju se instaliraju svi potrebni paketi za PHP 8.2 verziju, uključujući PHP CLI, FPM, MySQL, ZIP, GD, i ostali.

```
FROM ubuntu:latest AS base

ENV DEBIAN_FRONTEND noninteractive

# Install dependencies
RUN apt update
RUN apt install -y software-properties-common
RUN add-apt-repository -y ppa:ondrej/php
RUN apt update
RUN apt install -y php8.2\
    php8.2-cli\
    php8.2-common\
    php8.2-fpm\
    php8.2-mysql\
    php8.2-zip\
    php8.2-gd\
    php8.2-mbstring\
    php8.2-curl\
    php8.2-xml\
    php8.2-bcmath\
    php8.2-pdo
```

Slika 16: *Docker konfiguracioni fajl (Instalacija verzija i paketa PHP-a)*

Nakon što su PHP paketi instalirani, dodaje se i PHP-FPM (*PHP FastCGI Process Manager*), što će biti ključno za obradu PHP zahtjeva unutar ovog kontejnera. Dalje će biti instaliran Composer, popularni alat za upravljanje zavisnostima u PHP projektima. Zatim će biti instaliran *curl* koji će omogućiti preuzimanje Composera sa njegovog zvaničnog sajta, a dalje će biti preuzeta Composer skripta i instalirana u sistemski direktorijum kako bi se moglo lako pristupiti Composer-u iz bilo kojeg dijela ovog sistema. (Slika 17)

```
# Install php-fpm
RUN apt install -y php8.2-fpm php8.2-cli

# Install composer
RUN apt install -y curl
RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer
```

Slika 17: *Docker konfiguracioni fajl (Instalacija PHP-fpm-a i Composer-a)*

Dio koda na Slici 18. dodaje podršku za Node.js u Docker kontejneru. Prvo se instaliraju neophodni alati za upravljanje paketima. Zatim se preuzima i dodaje ključ Node.js repozitorijuma, omogućavajući verifikaciju preuzetih paketa. Nakon toga se dodaje Node.js repozitorijum u listu izvora paketa, a zatim se ažurira lista dostupnih paketa. Na kraju, instalira se Node.js u Docker kontejneru.

```
# Install nodejs
RUN apt install -y ca-certificates gnupg
RUN mkdir -p /etc/apt/keyrings
RUN curl -fsSL https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key
        | gpg --dearmor -o /etc/apt/keyrings/nodesource.gpg
ENV NODE_MAJOR 18
RUN echo
        "deb [signed-by=/etc/apt/keyrings/nodesource.gpg]
        https://deb.nodesource.com/node_$NODE_MAJOR.x nodistro main"
        | tee /etc/apt/sources.list.d/nodesource.list
RUN apt update
RUN apt install -y nodejs
```

Slika 18: *Docker konfiguracioni fajl (Instalacija NodeJS-a)*

```
#Install nginx
RUN apt install -y nginx
RUN echo "\
    server {\n\
        listen 8000;\n\
        listen [::]:8000;\n\
        root /var/www/html/public;\n\
        add_header X-Frame-Options \"SAMEORIGIN\";\n\


        add_header X-Content-Type-Options \"nosniff\";\n\
        add_header 'Access-Control-Allow-Origin' '*';\n\
        add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';\n\
         add_header 'Access-Control-Expose-Headers' 'Content-Length,Content-Range';\n\
        add_header X-Content-Type-Options \"nosniff\";\n\
        index index.php;\n\
        charset utf-8;\n\
        location / {\n\
            try_files \$uri \$uri/ /index.php?\$query_string;\n\
        }\n\
        location = /favicon.ico { access_log off; log_not_found off; }\n\
        location = /robots.txt  { access_log off; log_not_found off; }\n\
        error_page 404 /index.php;\n\
        location ~ \.php$ {\n\
            fastcgi_pass unix:/run/php/php8.2-fpm.sock;\n\
            fastcgi_param SCRIPT_FILENAME \$realpath_root\$fastcgi_script_name;\n\
            include fastcgi_params;\n\
        }\n\
        location ~ /\.(?!well-known).* {\n\
            deny all;\n\
        }\n\
    }\n" > /etc/nginx/sites-available/default

RUN echo "\
    #!/bin/sh\n\
    echo \"Starting services...\"\n\
    service php8.2-fpm start\n\
    nginx -g \"daemon off;\" &\n\
    echo \"Ready.\"\n\
    tail -s 1 /var/log/nginx/*.log -f\n\
    " > /start.sh
```

Slika 19: *Docker konfiguracioni fajl (Konfiguracija NGINX-a)*

U dijelu koda prikazanom na Slici 19. instalira se NGINX web server unutar Docker kontejnera. Nakon instalacije, vrši se konfiguracija NGINX servera da bi bio spreman za obradu zahtjeva. Konfiguracija uključuje definisanje osnovnih podešavanja servera kao što su 'slušači' portova i početni direktorijum, kao i postavljanje bezbjednosnih zaglavlja i pravila rute. Takođe, konfiguriše se NGINX da prosleđuje PHP zahtjeve PHP-FPM servisu. Postavlja se i skripta za

pokretanje servisa unutar kontejnera, koja prvo pokreće PHP-FPM, a zatim NGINX server, čime se omogućava da kontejner bude spreman za obradu zahtjeva.

Na Slici 20, u dijelu koda kopiraju se sve datoteke i direktorijumi iz trenutnog radnog direktorijuma (gdje se nalazi *Dockerfile*) u direktorijum */var/www/html* unutar Docker kontejnera. Nakon toga, postavlja se radni direktorijum kontejnera na */var/www/html*, gdje će se izvršavati sve dalje komande. Takođe, vrše se podešavanja dozvola nad ovim direktorijumom kako bi vlasnik bio *www-data* korisnik, tj. da bi direktorijum imao sve dozvole.

```
COPY . /var/www/html
WORKDIR /var/www/html


RUN chown -R www-data:www-data /var/www/html
```

Slika 20: *Docker konfiguracioni fajl (Kopiranje i dodjela permisija)*

Nakon kopiranja aplikacionog koda u Docker kontejneru, izvršavaju se neophodne komande (Slika 21) za instalaciju PHP i JavaScript zavisnosti (*composer install, npm install*) i izgradnju - *build* JavaScript resursa (*npm run build*). Zatim se pokreću Artisan komande za konfiguraciju Laravel aplikacije, uključujući povezivanje skladišta, generisanje aplikacionog ključa i brisanje optimizovanih verzija fajlova (*php artisan storage:link, php artisan key:generate, php artisan optimize:clear*)

```
RUN composer install
RUN npm install
RUN npm run build
RUN php artisan storage:link
RUN php artisan key:generate
RUN php artisan optimize:clear
```

Slika 21: *Docker konfiguracioni fajl (Komande za pokretanje projekta)*

Ključni dio Dockerfile-a je na Slici 22, jer se putem njega definiše komanda za pokretanje servisa unutar kontejnera. Ova komanda pokreće skriptu */start.sh*, koja inicijalizuje potrebne servise unutar kontejnera, uključujući PHP-FPM i NGINX server, omogućavajući da Laravel aplikacija bude dostupna na portu 8000.

Slika 22: *Docker konfiguracioni fajl (Definisanje porta i pokretanje skripte)*

Za svaki mikroservis aplikacije kreiran je poseban Dockerfile, koji definiše zavisnosti i okruženje u kojem taj mikroservis funkcioniše. Iako je u ovom dijelu rada prikazan samo Dockerfile za jedan od mikroservisa, isti postupak je primijenjen na preostala dva mikroservisa, koja su takođe kontejnerizovana koristeći Docker.

5. IMPLEMENTACIJA

U ovom poglavlju detaljno će se opisati postupak pokretanja aplikacija, kreiranje Docker *image*-a i pokretanja kontejnera pomoću alata kao što su Docker i Docker Compose. Dok se u poglavlju „Arhitektura aplikacije“ detaljno prikazao sami sadržaj *DockerFile-a*.

Na početku, izvršeno je kloniranje Git repozitorijuma. Git je distribuirani sistem za kontrolu verzija koji omogućava efikasno praćenje promjena u izvornom kodu i olakšava timski rad. Korištenjem Git-a, svaki član tima može raditi na svojoj kopiji koda, a promjene se kasnije mogu integrirati u zajednički repozitorijum. Git također omogućava vraćanje na prethodne verzije koda, što doprinosi stabilnosti i pouzdanosti tokom razvoja. Ovaj postupak kloniranja obezbjeđuje lokalnu kopiju izvornog koda mikroservisa, čime se omogućava njegovo prilagođavanje i implementacija na različitim serverima. Prvobitno se izvršava komanda **git clone *https://github.com/*****/*****.git***. Ovaj postupak obezbjeđuje lokalnu kopiju izvornog koda koji se koristi za razvoj mikroservisa. Nakon kloniranja, mikroservisi su implementirani na Oracle Cloud serveru koji je odabran za implementaciju ovog projekta. Ovi mikroservisi smješteni su u direktorijumu */var/websites/agro-trade-cms*, što omogućava njihovu dostupnost i upravljanje na serveru.

5.1. Kreiranje i pokretanje kontejnera

Nakon kloniranja projekta na serveru, slijedi kreiranje Dockerfile-a čiji sadržaj je objašnjen u poglavlju **4.4.** a koji služi za kreiranje *image*-a. Takođe se definiše i *docker-compose.yml* koji nakon samog *build*-a kreira novi *image* (implementacija Docker Compose-a je objašnjena u poglavlju 5.1.2.).

Pokretanje *build*-a na samom projektu, i pokretanje komponenti koje se nalaze u Dockerfile-u se vrši pomoću komande: **docker build -t** *agro-trade-cms:latest* **.**

5.1.1. Pristup i provjera kontejnera

Komandom **docker ps** (Slika 23), prikazuju se svi aktivni Docker kontejneri na serveru. Kada je potrebno pristupiti određenom kontejneru, koristi se Container ID samog kontejnera i implementira se u sledeću komandu: **docker exec -it** *ff14da970620* **bash** (kao što je prikazano u primjeru kontejnera: *ff14da970620*). Na taj način dobija se pristup konkretnom Docker kontejneru putem Shell terminala (Slika 24).

```
ubuntu@                    ubuntu-test:~$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED         STATUS
                                                                                    NAMES
ff14da970620   agro-front:latest                    "sh /start.sh"           4 days ago      Up 4 days
                                                                                    agro-full_agro-front_1
acde44262cea   agro-trade-cms:latest                "sh /start.sh"           4 days ago      Up 4 days
                                                                                    agro-full_agro-trade_1
8f1125956a2f   rural-shop:latest                    "sh /start.sh"           4 days ago      Up 4 days
```

Slika 23: *docker ps komanda*

```
ubuntu@                    -ubuntu-test:~$ docker exec -it ff14da970620 bash
root@                    -ubuntu-test:/var/www/html# ll
total 516
drwxr-xr-x 1 www-data www-data   4096 Jun 24 05:30 ./
drwxr-xr-x 1 root     root       4096 Jun 18 07:11 ../
-rw-r--r-- 1 www-data www-data    258 Jun  4 06:29 .editorconfig
-rw-r--r-- 1 www-data www-data   1323 Jun 24 05:30 .env
-rw-r--r-- 1 www-data www-data   1176 Jun  4 06:40 .env.example
drwxr-xr-x 1 www-data www-data   4096 Jun 24 05:29 .git/
-rw-r--r-- 1 www-data www-data    186 Jun  4 06:29 .gitattributes
-rw-r--r-- 1 www-data www-data    243 Jun  4 06:29 .gitignore
-rw-r--r-- 1 www-data www-data   3008 Jun 17 06:10 Dockerfile
-rw-r--r-- 1 www-data www-data   4158 Jun  4 06:29 README.md
drwxr-xr-x 1 www-data www-data   4096 Jun 10 06:32 app/
-rwxr-xr-x 1 www-data www-data   1686 Jun  4 06:29 artisan*
drwxr-xr-x 1 www-data www-data   4096 Jun  4 06:29 bootstrap/
-rw-r--r-- 1 www-data www-data   1969 Jun  4 06:29 composer.json
-rw-r--r-- 1 www-data www-data 293874 Jun  4 06:29 composer.lock
drwxr-xr-x 1 www-data www-data   4096 Jun  4 08:05 config/
drwxr-xr-x 1 www-data www-data   4096 Jun  4 06:29 database/
-rw-r--r-- 1 www-data www-data      0 Jun  4 08:48 docker-build.sh
-rw-r--r-- 1 www-data www-data    615 Jun 18 07:11 index.nginx-debian.html
-rw-r--r-- 1 www-data www-data    169 Jun  4 06:29 jsconfig.json
drwxr-xr-x 1 www-data www-data   4096 Jun 24 05:30 node_modules/
-rw-r--r-- 1 www-data www-data  87673 Jun 24 05:30 package-lock.json
-rw-r--r-- 1 www-data www-data    687 Jun  4 08:04 package.json
-rw-r--r-- 1 www-data www-data   1142 Jun  4 06:29 phpunit.xml
-rw-r--r-- 1 www-data www-data     95 Jun  4 06:29 postcss.config.js
drwxr-xr-x 1 www-data www-data   4096 Jun 24 05:30 public/
drwxr-xr-x 1 www-data www-data   4096 Jun  4 06:29 resources/
drwxr-xr-x 1 www-data www-data   4096 Jun 10 06:32 routes/
drwxr-xr-x 1 www-data www-data   4096 Jun  4 06:29 storage/
-rw-r--r-- 1 www-data www-data    567 Jun  4 06:29 tailwind.config.js
drwxr-xr-x 1 www-data www-data   4096 Jun  4 06:29 tests/
drwxr-xr-x 1 www-data www-data   4096 Jun 24 05:30 vendor/
-rw-r--r-- 1 www-data www-data    472 Jun  4 06:29 vite.config.js
```

Slika 24: *Shell terminal docker kontejnera*

5.1.2. Docker Compose

Na Slici 25 prikazano je kreiranje Docker Compose file-a koji se kreira u root repozitorijumu projekta i koji služi za definisanje Docker kontejnera, servisa, mrežnih postavki i drugih konfiguracija za pokretanje aplikacije, a u ovom slučaju je sa sadržajem prikazanom na Slici 25.

Slika 25: *Docker Compose file*

U ovom Docker Compose fajlu *docker-compose.yml*, verzija je definisana kao '3.8', što označava upotrebu odgovarajuće verzije specifikacije Docker Compose formata. U okviru sekcije *services*, definisan je servis nazvan *app*. Ovaj servis koristi Docker *image* ***agro-trade-cms:latest***, koji je ranije kreiran i konfigurisan da koristi *host* mrežni mod, omogućavajući aplikaciji da dijeli mrežne resurse sa *Host-om*.

U okviru istog Docker Compose fajla, definiše se okruženje za aplikaciju, gdje se postavljaju promjenljive za bazu podataka. Konkretno, definišu se promjenljive *DB_DATABASE*, *DB_USERNAME*, i *DB_PASSWORD*, koje aplikacija koristi za pristup bazi podataka. Ove promjenljive sadrže odgovarajuće vrijednosti za naziv baze podataka, korisničko ime i lozinku. Ovim postavkama obezbjeđuje se ispravna konfiguracija za rad aplikacije sa bazom podataka unutar Docker kontejnera.

Kada se koriste komande za rad sa Docker Compose fajlom, kao što su *docker-compose up -d* za pokretanje servisa i *docker-compose down* za njihovo zaustavljanje, važno je znati da ova komanda ne samo da zaustavlja sve aktivne kontejnere, već takođe uklanja mreže, zapise i sve druge resurse koji su kreirani i povezani sa servisima definisanim u *docker-compose.yml* fajlu. Odmah nakon *build*-a novih promjena, i kreiranja novog *image*-a, koristi se komanda *docker-compose down* za zaustavljanje i uklanjanje svih aktivnih kontejnera, mreža i drugih resursa koji su definisani u *docker-compose.yml* fajlu.

Ako je potrebno ponovo pokrenuti *image*-a, koristi se *docker-compose up -d* za pokretanje svih servisa definisanih u Docker Compose fajlu u pozadinskom režimu (*detached mode*). Kada se izvrši ova komanda, Docker Compose će podići sve definisane kontejnere i pokrenuti ih kao servise u pozadini, što znači da će oni nastaviti sa radom iako se izlaz iz terminala zatvori.

Kako bi se pojednostavilo pokretanje servisa, s obzirom na to da aplikacija sadrži tri odvojena dijela, i kako bi se izbjeglo pokretanje svih servisa pojedinačno, sve komande se stavljaju u jedan Docker Compose fajl (Slika 26). Sa svim servisima definisanim u jednom fajlu, napravljeno je jedno centralno mjesto za upravljanje svim komponentama sistema. To olakšava pregled i modifikaciju konfiguracija. Pokretanje svih servisa postaje jednostavno pomoću komande (*docker-compose up -d*). Isto važi i za zaustavljanje (*docker-compose down*). Ovo smanjuje potrebu za višestrukim skriptama ili komandama za pokretanje različitih djelova sistema.

```yaml
version: '3.8'

//interface aplikacije

services:
  agro-front:
    image: agro-front:latest
    network_mode: host


//cms aplikacije
  agro-trade:
    image: agro-trade-cms:latest
    network_mode: host
    volumes:
      - laravel-storage:/var/www/html/storage
    environment:
      DB_DATABASE: agro_trade
      DB_USERNAME: agro_trade_usr
      DB_PASSWORD: Admin2022!


//aplikacija za naručivanje
  rural-shop:
    image: rural-shop:latest
    network_mode: host
    environment:
      DB_DATABASE: rural_shop
      DB_USERNAME: rural_shop_usr
      DB_PASSWORD: Admin2022!

volumes:
  laravel-storage:/var/www/html/storage
```

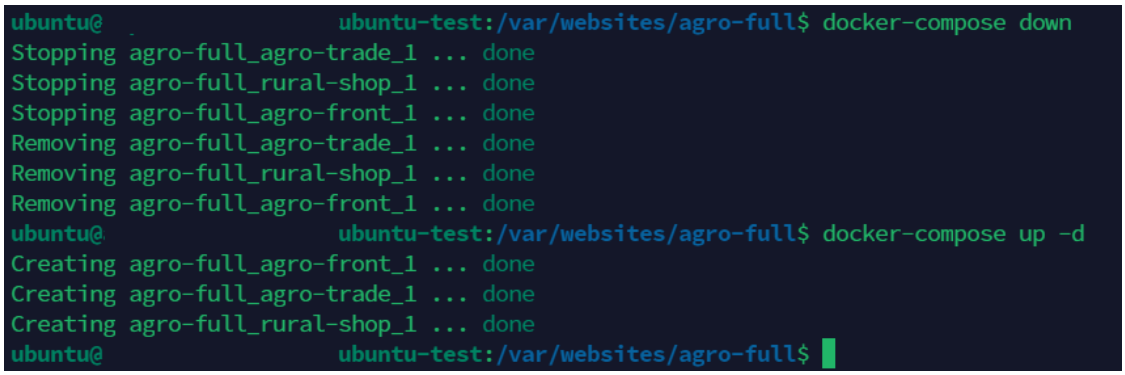
Slika 26: *docker-compose.yml fajl*

Servis nazvan **agro-front** (Interface aplikacije) koristi Docker *image agro-front:latest*. Ovaj servis koristi network_mode: host, što omogućava aplikaciji da dijeli mrežne resurse sa host mašinom.

Servis **agro-trade** (Sistem za upravljanje sadržajem) koristi Docker *image agro-trade-cms:latest.* Takođe koristi network_mode: host. Definisane su i promjenljive okruženja za konfiguraciju baze podataka koje aplikacija koristi za pristup.

Servis **rural-shop** (Sistem za naručivanje) koristi Docker *image rural-shop:latest* i takođe koristi network_mode: host. Kao i za agro-trade, definisane su promjenljive okruženja za konfiguraciju baze podataka.

Na Slici 27 prikazano je kako funkcionišu Docker Compose komande. Komanda *docker-compose down* zaustavlja i uklanja trenutno pokrenute kontejnere, dok komanda *docker-compose up -d* pokreće nove kontejnere u pozadini koristeći konfiguracije definisane u *docker-compose.yml* fajlu.



Slika 27: *docker-compose down I docker-compose up -d komande*

Nakon ovog procesa, poslednje izmjene su preuzete na serveru, aplikacija je aktivna i nalazi se na serveru. Svi kontejneri su pokrenuti i funkcionalni, omogućavajući pristup aplikaciji. Ovaj proces osigurava da su svi servisi pravilno pokrenuti i međusobno povezani.

6. APLIKACIJA

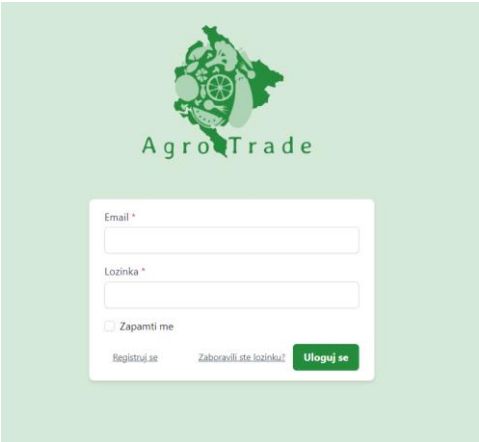
U samom master radu se analizira primjena mikroservisa i kontejnerizacije kroz razvoj aplikacije namijenjene plasmanu poljoprivrednih proizvoda. Aplikacija je strukturirana oko tri dijela aplikacije, svaki smješten u poseban kontejner radi demonstracije funkcionalnosti ovih tehnologija.

- Prvi mikroservis je CMS (Sistem za upravljanje sadržajem), gdje su definirane dvije uloge: Administrator i Korisnik. Administrator ima privilegije za upravljanje korisnicima i kategorijama, dok Korisnik kreira oglase i potvrđuje porudžbine unutar aplikacije.
- Drugi mikroservis je sistem za naručivanje, koji omogućava korisnicima kreiranje porudžbina putem korisničkog interfejsa.
- Treći dio je Korisnički portal, koji pruža pregled svih funkcionalnosti podržanih prethodna dva mikroservisa korisnicima aplikacije.

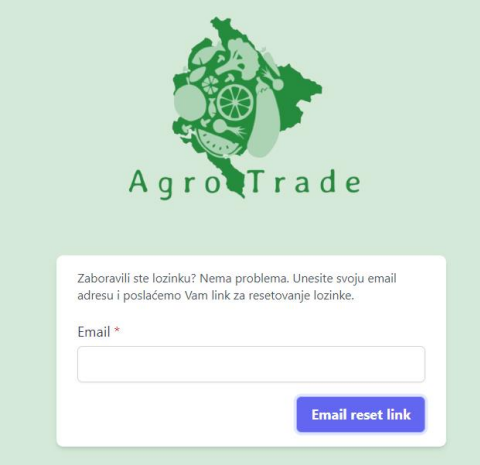
Ovakva arhitektura omogućava integraciju i interakciju različitih komponenti aplikacije kroz definisane mikroservise.

6.1. Sistem za upravljanje sadržajem (CMS)

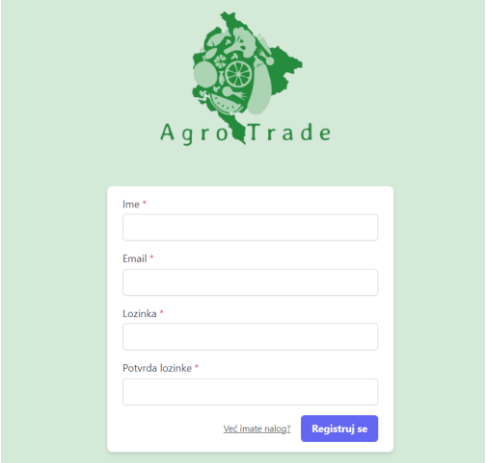
Na početnoj stranici CMS-a prikazuje se forma za prijavu (Slika 28). Korisnicima koji već imaju nalog na aplikaciji dostupna su polja za unos e-mail adrese i lozinke. Takođe, moguće je odabrati opciju "Zapamti me" koja čuva sesiju nakon prijavljivanja. Uz to, tu je i dugme "Zaboravili ste lozinku?" koje, nakon odabira, otvara novu formu (Slika 29) za unos e-mail adrese na koju će biti proslijeđen e-mail sa opcijom resetovanja lozinke. Za korisnike koji nemaju nalog na CMS-u, dostupno je dugme "Registruj se" koje otvara novu formu (Slika 30) za registraciju korisnika.



Slika 28: *Log in stanica na CMS-u*



Slika 29: *Forma za reset lozinke*



Slika 30: *Forma za registraciju*

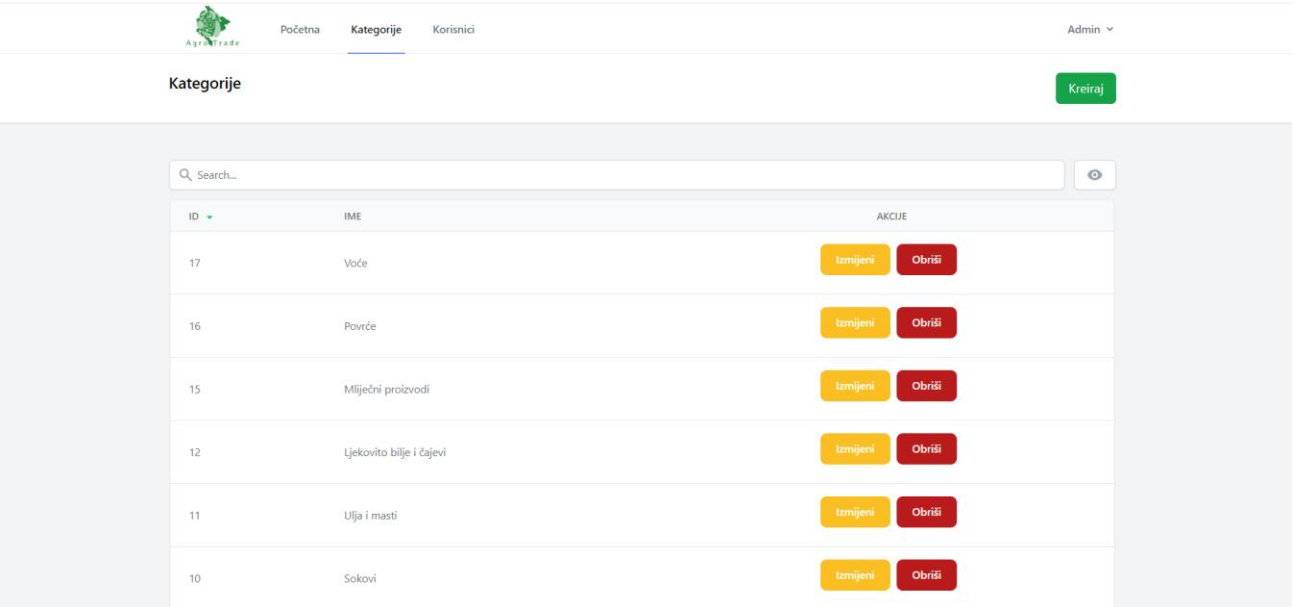
Na CMS-u postoje dvije uloge: Administratorska i Korisnička.

Administratorski sistem (Admin uloga)

U Admin dijelu upravlja se svim prikazima koji se nalaze na korisničkom interfejsu, uključujući:

- Prikaz i kreiranje kategorija proizvoda
- Prikaz i kreiranja korisnika

Kategorije koje se kreiraju na Admin panelu (Slika 31) biće prikazane na korisničkom interfejsu. Prikaz je vrlo jednostavan, sa nekoliko funkcionalnosti.

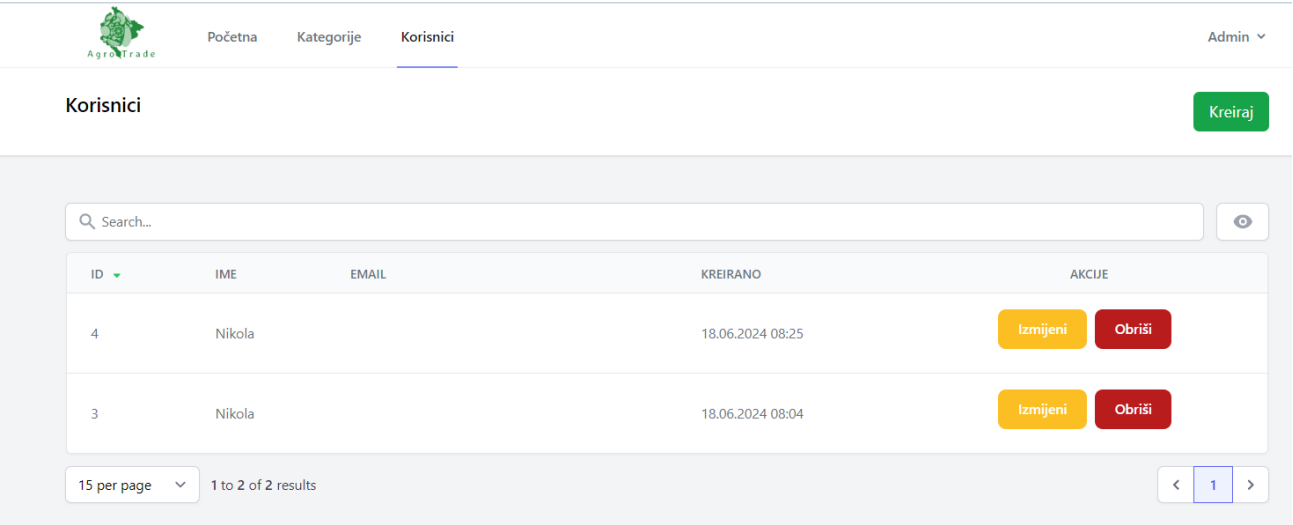


Slika 31: *Admin panel-Prikaz kategorija*

Klikom na dugme "Kreiraj" otvara se mini forma (Slika 32) sa poljima za unos naziva kategorije i slike koja će biti prikazana na početnom prozoru kategorije. Ovaj interfejs omogućava administriranje kategorija proizvoda na efikasan i intuitivan način.

Slika 32: *Admin panel-Kreiranje kategorija*

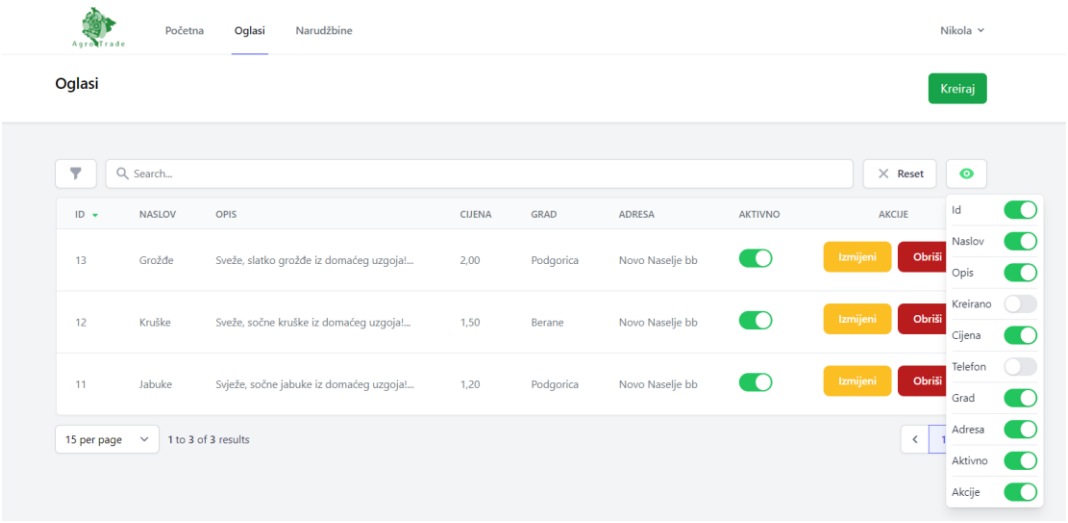
Odabirom dijela panela za korisnike, otvara se panel sa svim korisnicima koji su registrovani na aplikaciji (Slika 33). Kroz ovaj panel omogućeno je upravljanje korisnicima, uključujući izmjenu njihovih podataka, resetovanje lozinke, brisanje korisnika, sortiranje i ostale administrativne funkcije. Takođe, na panelu se nalazi dugme "Kreiraj" koje ima iste funkcionalnosti kao i dugme za registraciju na početnoj stranici CMS-a (vidi Sliku 30).



Slika 33: *Admin panel-Prikaz korisnika*

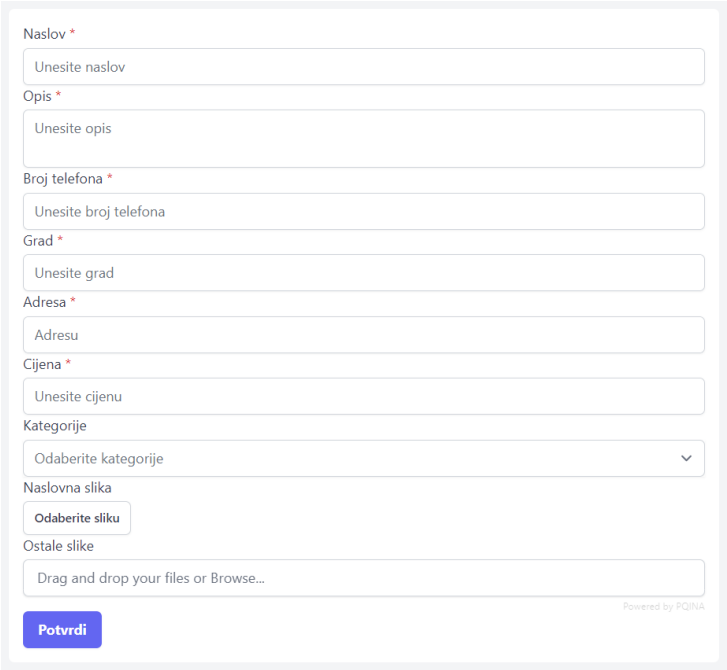
Korisnički servis (User uloga)

Na user dijelu, nakon kreiranja naloga, otvara se korisnički panel (Slika 34) koji je u ovoj aplikaciji namijenjen proizvođačima koji žele kreirati oglas koji će se prikazivati na korisničkom interfejsu. U listi oglasa nalaze se različite funkcionalnosti, uključujući dugme "Aktivno" koje može sakriti oglas sa korisničkog interfejsa, dugme za izmjenu i brisanje oglasa, kao i dugme za upravljanje prikaza željenih kolona.



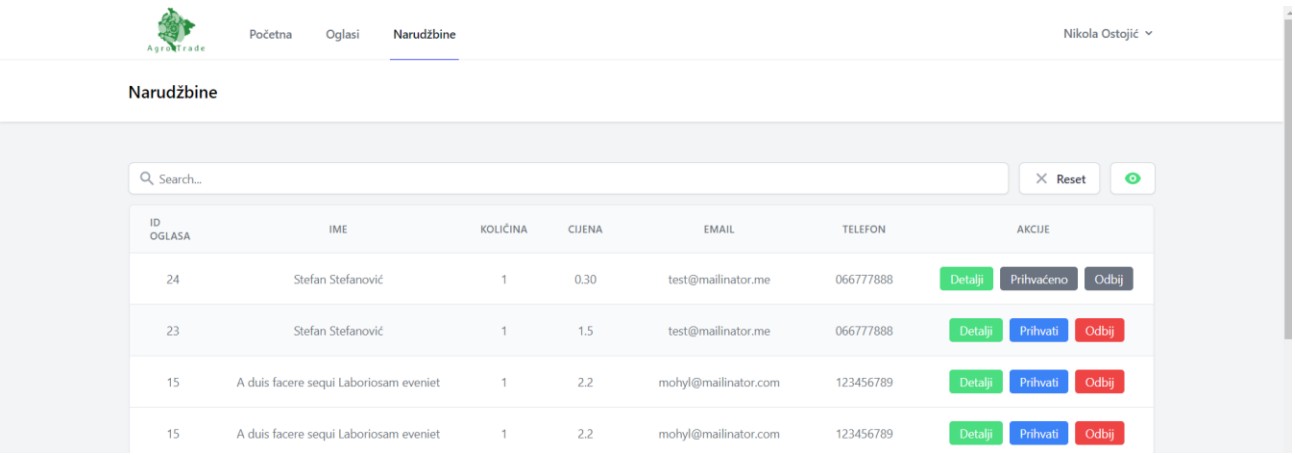
Slika 34: *User panel-Prikaz oglasa*

Takođe, u panelu gdje se nalaze oglasi postoji dugme "Kreiraj" koje služi za kreiranje novih oglasa. Klikom na to dugme, otvara se forma za kreiranje oglasa (Slika 35). Na toj formi nalaze se sva potrebna polja za kreiranje oglasa: Naslov, Opis, Broj telefona, Grad, Adresa, Cijena, Kategorija (sve koje su kreirane na Admin panelu), Naslovna slika i Ostale slike.

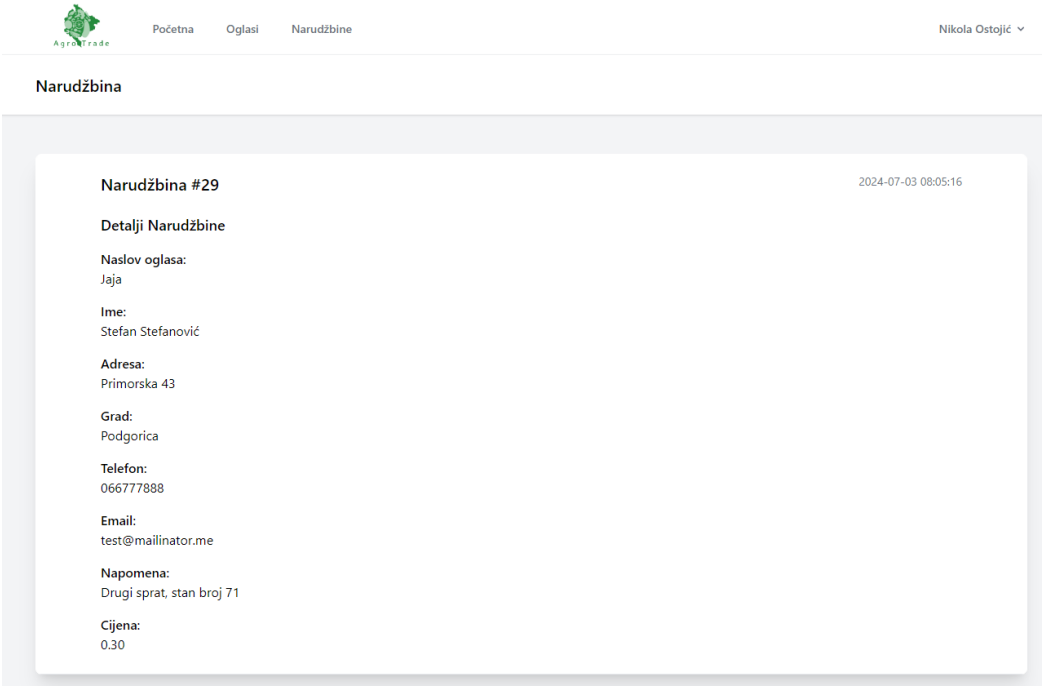


Slika 35: *User panel-Kreiranje oglasa*

Na korisničkom panelu postoji i odjeljak za narudžbine (Slika 36), koji je povezan s mikroservisom za naručivanje. Ovaj odjeljak funkcioniše po principu kreiranja narudžbe preko korisničkog interfejsa putem posebne forme (Slika 45). Kada se narudžbina kreira, aplikacija dostavlja tu narudžbinu korisniku CMS-a koji je kreirao oglas na uvid i potvrdu narudžbine. Dio za narudžbine sadrži nekoliko funkcionalnosti kao što su: pretraživanje oglasa, opcija za kontrolu vidljivosti kolona u tabeli, dugme za reset opcija i još dodatnih funkcionalnosti. Za svaki oglas postoji dugme za prihvatanje i odbijanje narudžbine, kao i mogućnost pregleda svih detalja narudžbine (Slika 37).



Slika 36: *User panel-Prikaz narudžbina*



Slika 37: *User panel-Prikaz detalja narudžbine*

Nakon prihvatanja narudžbine, na e-mail adresu kurirske službe koja je ranije definisana, stiže potvrda da je narudžbina prihvaćena, zajedno sa svim informacijama o narudžbini i detaljima o kreatoru oglasa od kojeg je potrebno preuzeti narudžbinu. Istovremeno, nakon prihvatanja, korisniku koji je kreirao narudžbinu stiže e-mail sa potvrdom narudžbine. Dok nakon odbijanja, narudžbina se briše, a na e-mail korisnika stiže obavještenje o odbijanju narudžbine. Sve ovo se radi u cilju obavještavanja svih uključenih strana i osiguravanja efikasne komunikacije.

6.2. Sistem za naručivanje

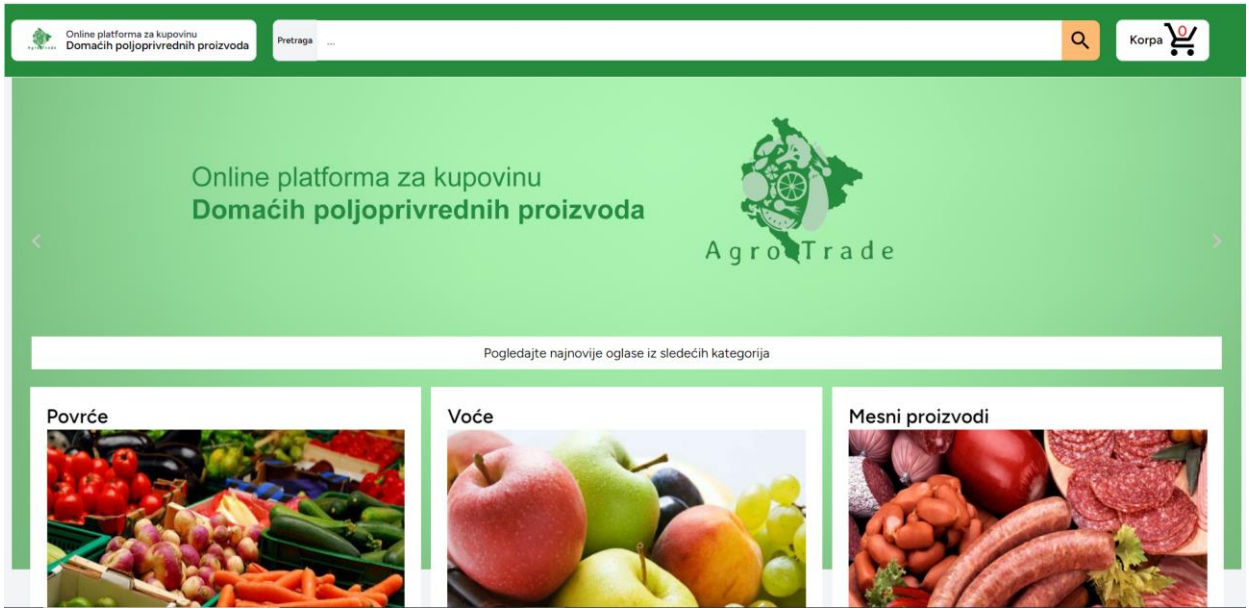
Sistem za naručivanje je implementiran preko API-ja (*application programming interface*) koji je povezan sa CMS-om i korisničkim interfejsom. On sadrži svoje specifične funkcionalnosti, koji omogućava komunikaciju između različitih djelova aplikacije. Ovaj mikroservis obezbjeđuje da se sve operacije vezane za naručivanje, u pozadini, odvijaju glatko i efikasno.

Ovaj sistem predstavlja odvojeni mikroservis koji omogućava kreiranje narudžbine i prenos iste do korisnika koji je kreirao oglas. Nakon što je narudžbina kreirana, prvo se šalje e-mail obavještenje sa informacijama o narudžbini. Ovo inicijalno obavještenje sadrži sve relevantne podatke koje je naručilac unio, pružajući kreatoru oglasa uvid u detalje narudžbine.

Narudžbina se, takođe, šalje putem e-maila osobi koja je kreirala narudžbinu, ali tek nakon potvrde od strane kreatora oglasa. Kreator oglasa može potvrditi ili odbiti zahtjev za narudžbinu, u zavisnosti od specifičnih zahtjeva koje je kreirao korisnik. Ako kreator oglasa odbije narudžbinu, korisniku stiže e-mail obavještenje sa razlogom odbijanja. Ako prihvati narudžbinu, e-mail sa informacijama o narudžbini i vremenu dostave se šalje korisniku koji je kreirao narudžbinu, kao i kurirskoj službi koja je zadužena za preuzimanje narudžbine od kreatora oglasa i dostavljanje korisniku. Na taj način, sistem osigurava efikasan proces naručivanja i dostave, uz pravovremeno informisanje svih uključenih strana.

6.3. Korisnički portal

Korisnički portal (Slika 38), predstavlja glavni dio aplikacije koji koriste krajnji korisnici. Sastoji se od početnog dijela koji komunicira sa dva mikroservisa u pozadini. Na ovom dijelu aplikacije prikazuju se kategorije koje su prethodno definisane u Admin panelu CMS-a. Kada korisnik odabere određenu kategoriju, prikazuju se oglasi koji su kreirani u Korisničkom panelu CMS-a za tu kategoriju. Ovaj dizajn omogućava jednostavnu i intuitivnu navigaciju kroz dostupne oglase, pružajući korisnicima lakoću pristupa potrebnim informacijama.



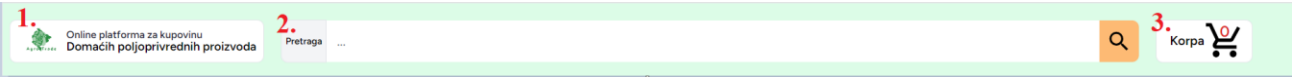
Slika 38: *Prikaz korisničkog interfejsa*

Sama početna strana sačinjena je od kategorija proizvoda i dodatnih funkcionalnosti. Na dnu početne strane nalaze se nasumično odabrani "Preporučeni oglasi" (Slika 39) koji su kreirani u različitim kategorijama. Tu je i dugme "Vrati na vrh" koje omogućava korisnicima da brzo skroluju na sami vrh stranice. Ovaj dizajn pruža korisnicima jednostavan pristup raznovrsnim oglasima i olakšava navigaciju kroz sadržaj aplikacije.

Slika 39: *Prikaz preporučenih oglasa*

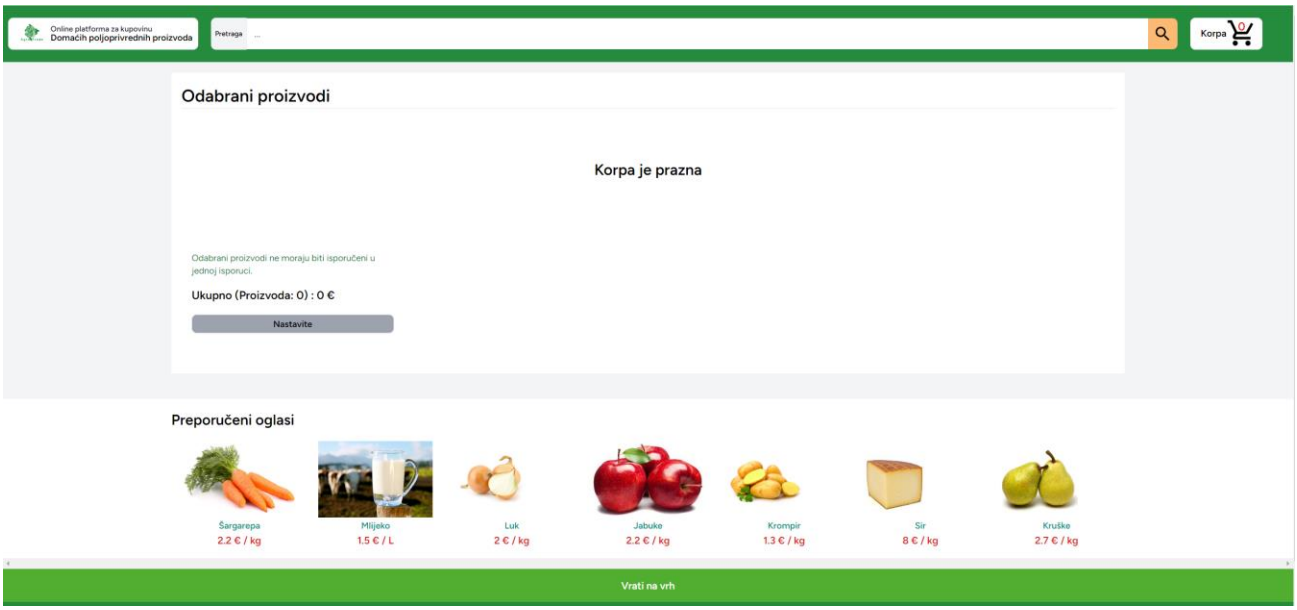
Pored preporučenih oglasa, na svakoj stranici nalazi se navigacioni heder (Slika 40) sa nekoliko ključnih funkcionalnosti:

1. Dugme sa logotipom koje vraća korisnika na početnu stranicu;
2. Polje za pretragu oglasa;
3. Korpa koja prikazuje broj odabranih namirnica.



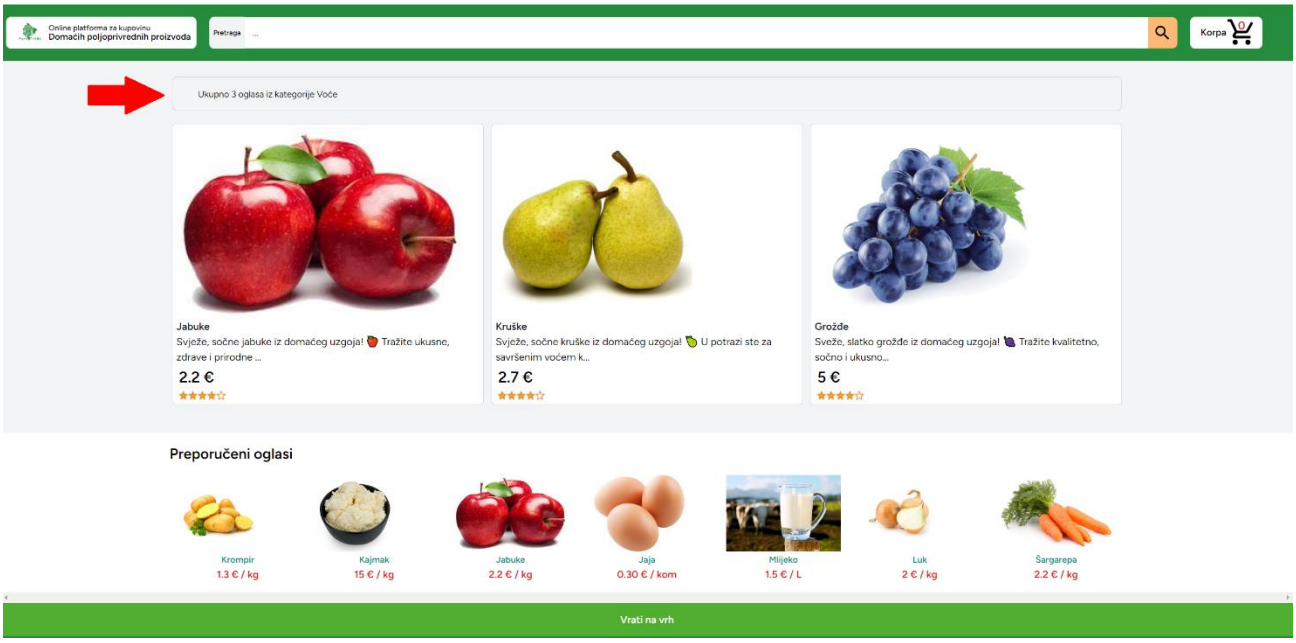
Slika 40: *Prikaz hedera*

Polje za pretragu oglasa izlistaće svaki oglas koji sadrži pretraženi pojam u naslovu ili opisu oglasa. Klikom na dugme "Korpa" u gornjem desnom uglu aplikacije otvara se nova stranica koja prikazuje sve proizvode odabrane ranije. U slučaju prikazanom na Slici 41, kada nijedna namirnica nije odabrana, ikonica korpe prikazuje broj "0", a klikom na ikonicu, korpa prikazuje da je prazna.



Slika 41: *Prikaz prazne korpe*

Povratkom na glavnu stranicu i odabirom kategorije "Povrće," za koju su ranije kroz CMS unešena tri testna oglasa radi boljeg prikaza funkcionalnosti, prikazuju se relevantni oglasi. Broj oglasa za tu kategoriju je jasno prikazan (označeno crvenom bojom na Slici 42), zajedno sa podacima unesenim tokom kreiranja oglasa, kao što su naslov, opis i cijena. Ovaj prikaz omogućava korisnicima brz i jednostavan uvid u dostupne proizvode u odabranoj kategoriji. (Slika 42).

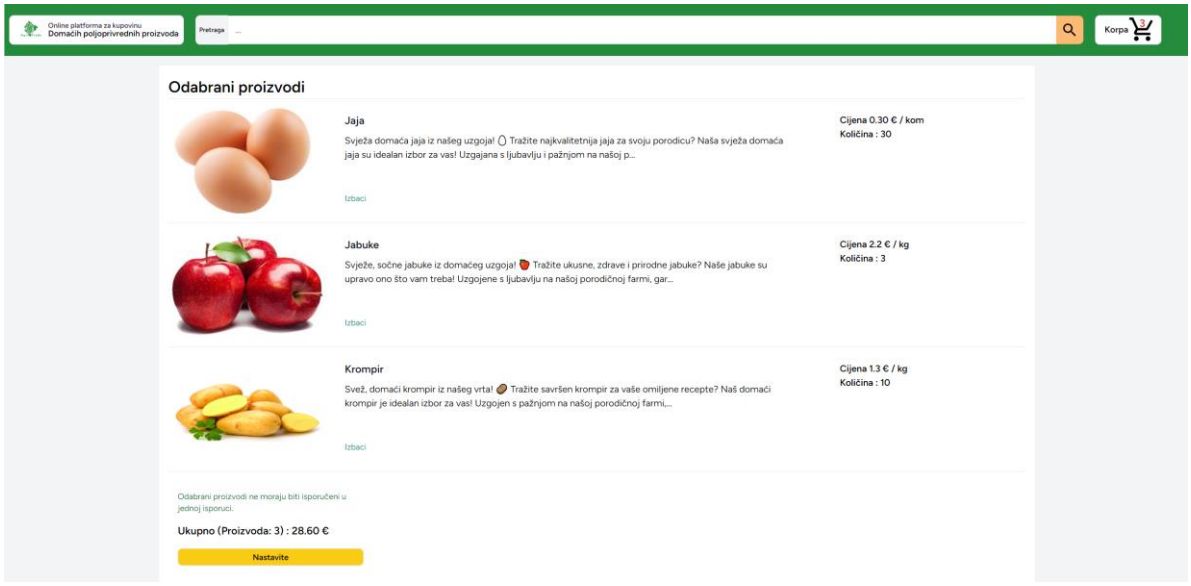


Slika 42: *Prikaz kategorije „Voće“*

Odabirom određenog oglasa, kreiranog od strane korisnika na CMS-u, otvara se prikaz cjelokupnog oglasa sa svim unesenim podacima. U desnom dijelu prozora prikazana je cijena proizvoda i polje za unos količine koju korisnik želi da poruči. Klikom na dugme "Dodaj u korpu" porudžbina se dodaje u korpu, uz automatsku potvrdu koja obavještava korisnika da je porudžbina uspješno dodata. Broj artikala u korpi se ažurira i prikazuje na ikonici korpe kao potvrda da je porudžbina evidentirana (Slika 43).

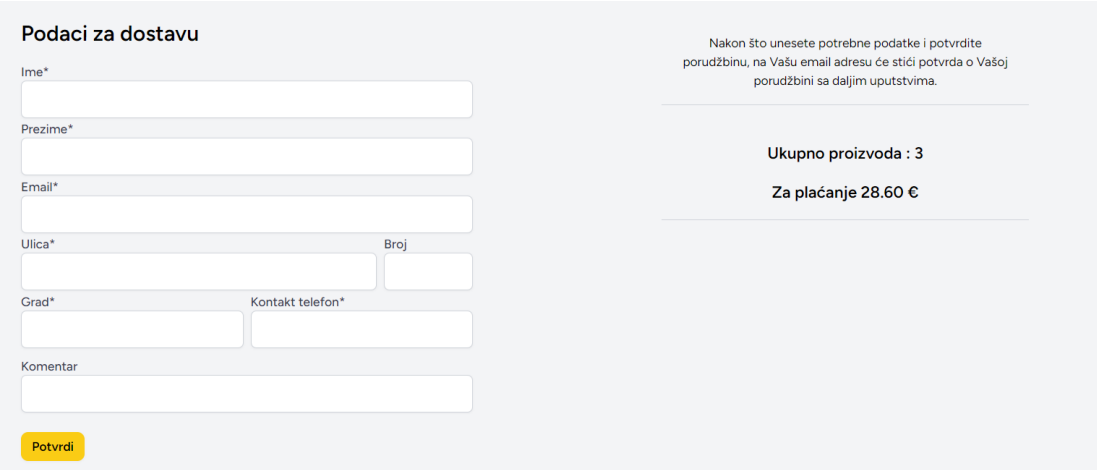
Slika 43: *Prikaz oglasa*

Kada korisnik odabere proizvode koje želi da poruči, oni se automatski nalaze u korpi. U korpi se prikazuje račun za svaki pojedinačni proizvod, ukupna cijena, količina i broj odabranih proizvoda. Ako dođe do greške prilikom odabira i korisnik želi da ukloni neki proizvod iz korpe, dostupno je dugme "Izbaci" koje odmah uklanja taj proizvod. (Slika 44)



Slika 44: *Prikaz korpe sa odabranim proizvodima*

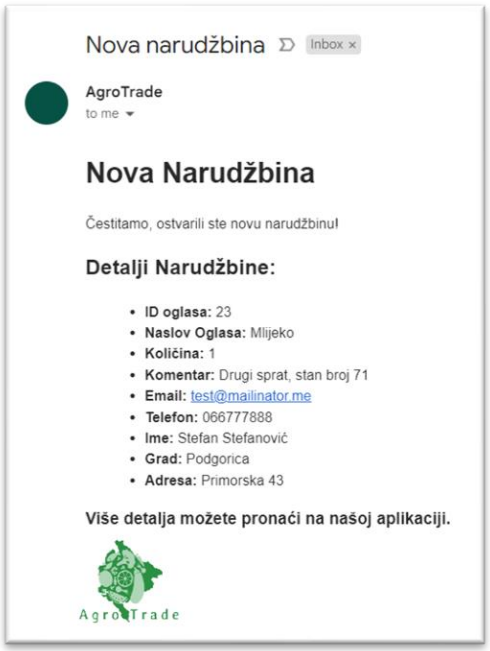
Klikom na dugme "Nastavi", aplikacija vodi korisnika na stranicu za unos podataka potrebnih za dostavu odabranih proizvoda (Slika 45). Ovaj proces omogućava korisnicima jednostavno upravljanje porudžbinama i unos potrebnih informacija za isporuku.



Slika 45: *Podaci koji su potrebni za dostavu*

Uz pažljivo popunjavanje forme sa ličnim podacima, korisnik može ostaviti dodatni komentar u vezi sa narudžbinom. Jedna narudžbina može obuhvatiti proizvode od više proizvođača i kreatora oglasa. Nakon unošenja potrebnih podataka za dostavu i potvrde narudžbine, pojaviće se *pop-up* prozor sa potvrdom o uspješnom kreiranju porudžbine (Slika 46). E-mail notifikacija će biti poslata svakom proizvođaču čiji je proizvod izabran u narudžbini sa svim informacijama koje je naručilac unio u formu za dostavu (Slika 47).

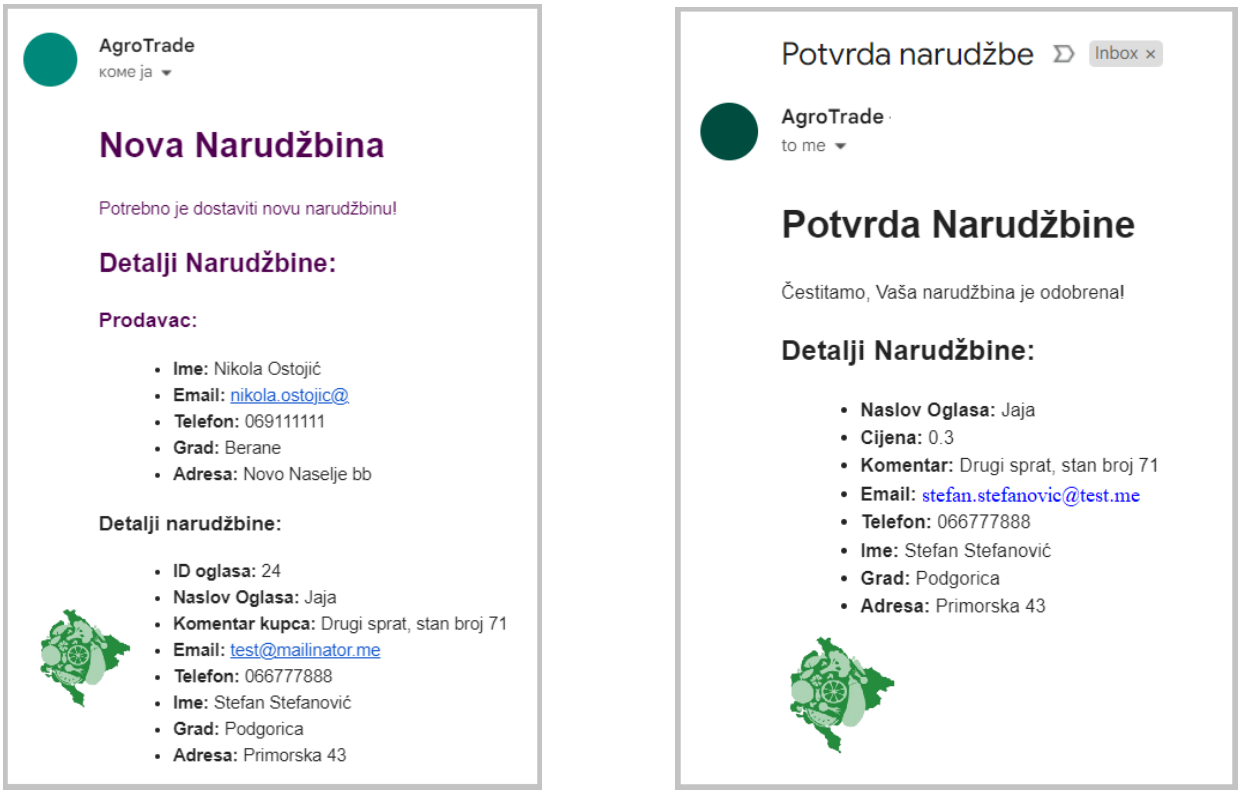
Slika 46: *Pop-up prozor za uspješno kreiranu porudžbinu*



Slika 47: *E-mail notifikacija kreatora oglasa o novoj narudžbini*

Kada stigne notifikacija na e-mail kreatoru oglasa sa informacijama o kreiranoj narudžbini, on ima obavezu da se uloguje na CMS i detaljnije pogleda narudžbinu. Nakon pregleda, kreator oglasa odlučuje da li će prihvatiti ili odbiti narudžbinu. U oba slučaja, korisniku koji je kreirao narudžbinu stiže e-mail obavještenje o statusu porudžbine (Slika 49), bilo da je prihvaćena ili odbijena.

Ako kreator oglasa prihvati porudžbinu, automatski se šalje e-mail i kurirskoj službi (Slika 48). Ovaj e-mail sadrži sve informacije koje je korisnik unio prilikom kreiranja narudžbine (ime, prezime, broj telefona, adresa za isporuku) kao i informacije o kreatoru oglasa od kojeg treba da preuzme narudžbinu. Na taj način se osigurava da svi relevantni detalji budu dostupni kurirskoj službi za efikasno preuzimanje i isporuku narudžbine.



Slike 48-49: *E-mail notifikacija kurirske službe (lijevo) i korisnika o potvrdi narudžbine(desno)*

7. ZAKLJUČAK

Realizacijom ovog projekta prikazano je da se mikroservisna arhitektura, u kombinaciji sa Docker kontejnerizacijom, može koristiti za značajno poboljšanje performansi i brzine izrade aplikacija. Kreiranjem nezavisnih kontejnera za svaki mikroservis, postignuta je efikasnija distribuciju resursa i lakše održavanje sistema. Svaki mikroservis mogao je biti razvijen, testiran i implementiran odvojeno, što je omogućilo bržu isporuku novih funkcionalnosti. Aplikacija za plasman domaćih poljoprivrednih proizvoda, koja je razvijena koristeći ove tehnologije, demonstrira kako se mikroservisi mogu uspješno implementirati i orkestrirati koristeći Docker. Ovakav pristup omogućava fleksibilnost u korišćenju različitih tehnologija i alata, čime se dodatno unapređuje razvojni proces i sigurnost aplikacije.

Online platforma za kupovinu domaćih poljoprivrednih proizvoda predstavlja praktičan primjer primjene ovih tehnologija. Korišćenjem mikroservisne arhitekture, svaki dio ove aplikacije - kao što su korisnički portal, sistem za upravljanje sadržajem i sistem za naručivanje - integrisan je kao zaseban mikroservis u Docker kontejnere. Ova arhitektura omogućava efikasnije upravljanje resursima, bolju skalabilnost tokom povećanog broja posjetilaca, kao i lakše održavanje sistema. Korisnici mogu brzo i jednostavno pretraživati, pregledati i kupiti raznovrsne domaće proizvode direktno od proizvođača.

Ovaj projekat ima perspektivu za dalje unapređenje. Postoji potencijal za uvođenje funkcionalnosti kao što je integracija sistema za naplatu putem platnih kartica, što bi značajno poboljšalo korisničko iskustvo i efikasnost procesa kupovine, kao i novih funkcionalnosti koje platforma može da podrži. Takođe, planira se povećanje resursa i optimizacija performansi kako bi se aplikacija bolje skalirala i lakše nosila sa rastućim zahtjevima tržišta. Kontinuirano praćenje novih tehnologija i alata omogućiće aplikaciji da ostane konkurentna i da konstantno unapređuje svoje funkcionalnosti.

LITERATURA

1. Aleksić, Marko: *Podman vs Docker: Everything You Need to Know*, 2022. https://phoenixnap.com/kb/podman-vs-docker. posjeta: 16.04.2024
2. Black, Andrew: Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation, vol. 231*, 2013, p. 3 – 20.
3. Chelladhurai, Jeeva; Singh, Vinod; Raj, Pethuru: *Learning Docker, Second Edition*, Packt Publishing, Birmingham – Mumbai, 2017.
4. Conway, Melvin: How do committees invent, *Datamation*, vol. 14, no. 4, 1968, pp. 28 – 31.
5. Docker. 2019. *Docker Overview*. https://docs.docker.com/get-started/overview/. posjeta: 16.04.2024
6. Dragoni, Nicola; Giallorenzo, Saverio; Lluch – Lafuente, Alberto; Mazzara, Manuel: *Microservices – Yesterday, today and tomorrow*, Springer, New York, 2017.
7. Garrigos, Irene; Wimmer, Manuel: Current Trends in Web Engineering, Springer International Publishing, Rome, 2018.
8. Gross, Hans; Melideo, Matteo; Sillitti, Alberto: Self certification and trust in component procurement, *Journal of Science of Computer Programming*, 2005.
9. Hamori, Ferenc: *The History of Kubernetes on Timeline*, 2023. https://blog.risingstack.com/the-history-of-kubernetes/. posjeta: 16.04.2024
10. IETF, *The OAuth 2.0 Authorization Framework*, 2012. https://tools.ietf.org/html/rfc6749. posjeta: 09.04.2024
11. IETF, *JSON Web Token (JWT).* 2015. https://tools.ietf.org/html/rfc7519. posjeta: 09.04.2024
12. Kane, Sean; Matthiass, Karl: *Docker: Up & Running*, O'Reilly Media Inc., Boston, 2023.
13. Kebbani, Nassim; Tylenda, Piotr; McKendrick, Russ: *The Kubernetes Bible*, Packt Publishing, Birmingham, 2022.
14. Koutoupis, Petros: *Everything You Need to Know about Linux Containers, Part II: Working with Linux Containers (LXC)*, 2018. https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-ii-working-linux-containers-lx. posjeta: 16.04.2024
15. Mazzara, Manuel;.Govoni, Sergio; *A Case Study of Web Services Orchestration*, Springer, Berlin, 2005.
16. Mazzara, Manuel; Khanda, Kevin; Mustafin, Ruslan; Rivera, Victor; Safina, Larisa; Sillitti, Alberto: *Microservices Science and Engineering*, Innopolis University, Innopolis, 2017.

17. Nadareishvili, Irakli; Ronnie, Mitra; McLarty, Matt; Amundsen, Mike: *Microservice Architecture – Aligning principles, practices and culture*, O'Reilly, Boston, 2016.
18. Namiot, Dmitri; Sneps – Sneppe, Manfred: On Micro – services Architecture, *International Journal of Open Information Technologies*, vol. 2, no. 9, 2014, pp. 24 – 27.
19. Newman, Sam: *Izgradnja mikrosistema – Dizajn sitno granulisanih sistema*, O'Reilly Media, Boston, 2015.
20. Ozmen, Huseyin: *Design and implementation of an Iot-based home automation system utilizing fog and cloud computing paradigms*, Bogazici Univeristy, Istanbul, 2018.
21. Pardon, Guy; Buijze Allard; Dustdar, Schahram: *Practical Microservices Architectural Patterns*, Apress, Kerala, 2019.
22. Poulton, Nigel; Joglekar, Pushkar: *The Kubernetes Book*, Leanpub, Victoria, 2020.
23. Red Hat Containers vs VMs, 2023. https://www.redhat.com/en/topics/containers/containers-vsvms. posjeta: 15.04.2024
24. Richards, Mark: *Microservices vs. service-oriented architecture*, O'Reilly Media, California, 2015.
25. Ronnie, Mitra; Nadareishvilli, Irakli: *Microservices: Up and Running: A Step-by-Step Guide to Building a Microservices Architecture*, O'Reilly Media, New York, 2020.
26. Safina, Larisa; Mazzara, Manuel; Montesi, Fabrizio; Rivera. Victor: *Data-driven workflows for microservices (genericity in jolie)*, AINA, Crans – Montana, 2016.
27. Sanchez, Carlos; Vilarino, Pablo: *PHP Microservices*, Packt Publishing, Birmingham.
28. Schenker, Gabriel: *Learn Docker – Fundamentals of Docker 19.x*, Packt Publishing, Brimingham, 2020.
29. Servile, Valentina: *Canary Development*, O'Reilly Media Inc., Boston, 2023.
30. Sillitti, Alberto; Vernazza, Tullio; Succi, Giancarlo: *Service oriented programming: a new paradigm of software reuse*, Springer, Berlin, 2002.
31. Thönes, Johannes: Microservices, *IEEE software*, vol. 32, no. 1, 2015, p. 116.
32. Turnbull, James: *The Docker Book*, Turnbull Press, New York, 2014.
33. Wolff, Eberhard: *Microservices: Flexible Software Architecture*, Addison-Wesley Professional, Boston, 2016.