

**SVEUČILIŠTE U ZAGREBU  
FAKULTET ORGANIZACIJE I INFORMATIKE  
VARAŽDIN**

**Josip Cesar**

# **APLIKACIJE TEMELJENE NA NoSQL I RELACIJSKIM BAZAMA PODATAKA**

**DIPLOMSKI RAD**

**Varaždin, 2018.**

**SVEUČILIŠTE U ZAGREBU**  
**FAKULTET ORGANIZACIJE I INFORMATIKE**  
**V A R A Ž D I N**

**Josip Cesar**

**Matični broj: 44435/15–R**

**Studij: Informacijsko i programsko inženjerstvo**

**APLIKACIJE TEMELJENE NA NoSQL I RELACIJSKIM BAZAMA**  
**PODATAKA**

**DIPLOMSKI RAD**

**Mentor:**

Doc. dr. sc. Mario Konecki

**Varaždin, rujan 2018.**

*Josip Cesar*

### **Izjava o izvornosti**

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

*Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi*

---

## Sažetak

Pojava NoSQL (Not only SQL) sustava za upravljanje podacima bila je rezultat nemogućnosti tradicionalnih relacijskih sustava za upravljanje podacima da omoguće modernim aplikacijama performanse i skalabilnost uslijed rada s velikim količinama raznovrsnih podataka. Svijet NoSQL baza podataka danas obuhvaća puno različitih rješenja koja su razvijena s ciljem da riješe te probleme, ali ne i s ciljem da zamijene relacijske baze podataka.

Ovaj rad bavi se aplikacijama koje se temelje na NoSQL i reakcijskim bazama podataka kao i samom idejom istovremenog korištenja različitih baza podataka unutar iste aplikacije. Rad započinje opisivanjem relacijskih i NoSQL baza podataka, nakon čega slijede opisi i usporedba nekih od najpoznatijih sustava koji se temelje na ovim tehnologijama. Nakon toga rad se posvećuje aplikacijama i aplikacijskim bazama podataka. Daje se njegova definicija, opisuju se vrste aplikacija i njihove arhitekture te se opisuju različiti pristupi razvoju aplikacijske baze podataka. Potom se pažnja prebacuje na samu ideju korištenja različitih tehnologija za pohranu podataka unutar iste aplikacije. U tom dijelu rada opisuju se različiti arhitekturni pristupi kojima je moguće ostvariti integraciju NoSQL i relacijskih baza podataka unutar iste aplikacije. Tom prilikom posebna pažnja se posvećuje arhitekturi mikroservisa. Nakon toga se u nastavku rada opisuje implementacija praktičnog primjera aplikacije temeljene na NoSQL i relacijskim bazama podataka čija arhitektura je inspirirana mikroservisima, a konačni zaključak rada je kako NoSQL i relacijske baze podataka mogu funkcionirati zajedno, ali i da razvoju aplikacije koja će se temeljiti na više različitih baza podataka treba pristupiti s oprezom.

**Ključne riječi:** aplikacije, usporedba; relacijske baze; NoSQL; polyglot persistence; mikroservisi

# Sadržaj

1. Uvod .....	1
2. Relacijske baze podataka .....	2
2.1. Relacijski model podataka.....	2
2.2. ACID transakcije .....	4
2.3. SQL jezik .....	5
2.4. Nedostaci i izazovi relacijskih baza podataka.....	6
3. NoSQL baze podataka.....	8
3.1. Značajke NoSQL baza podataka.....	8
3.1.1. Ne-relacijski model podataka .....	9
3.1.2. Nepostojanje sheme.....	10
3.1.3. Distribuiranost .....	11
3.1.4. CAP i BASE pristup.....	11
3.1.5. Otvoreni izvorni kôd.....	12
3.1.6. Podrška za velike podatke.....	13
3.1.7. Horizontalna skalabilnost .....	13
3.1.8. Raznolikost modela podataka .....	14
3.2. Vrste NoSQL baza podataka.....	14
3.2.1. Ključ/vrijednost baze podataka .....	15
3.2.2. Dokument-baze podataka .....	16
3.2.3. Stupčane baze podataka.....	18
3.2.4. Graf-baze podataka.....	20
3.3. Prednosti i nedostaci NoSQL baza podataka .....	21
4. Pregled i usporedba pojedinih implementacija relacijskih i NoSQL baza podataka .....	22
4.1. MySQL.....	22
4.2. Redis .....	24
4.3. MongoDB.....	24

4.4. Cassandra .....	26
4.5. Neo4j .....	26
4.6. Usporedba pregledanih rješenja .....	28
4.6.1. Razlike u modelima podataka .....	28
4.6.2. Razlike u modelima upita .....	28
4.6.3. Razlike u agregaciji podataka.....	29
4.6.4. Razlike u integraciji s aplikacijama .....	30
5. Aplikacije temeljene na NoSQL i relacijskim bazama podataka .....	32
5.1. Definicija i vrste aplikacija .....	32
5.1.1. Desktop aplikacije .....	33
5.1.2. Mobilne aplikacije.....	33
5.1.3. Web aplikacije.....	34
5.2. Arhitekture aplikacija.....	35
5.2.1. Monolitne arhitekture.....	35
5.2.2. Distribuirane arhitekture .....	37
5.3. Definicija i pristupi razvoju aplikacijske baze podataka.....	38
5.3.1. Relacijski pristup .....	38
5.3.2. NoSQL pristup.....	39
5.3.3. Hibridni pristup .....	39
5.4. Polyglot persistence.....	40
5.4.1. Pristupi integraciji NoSQL i relacijskih baza podataka .....	41
5.4.2. Prednosti i mane „ <i>Polyglot persistence</i> “ arhitekture.....	45
5.5. Aplikacije temeljene na mikroservisima .....	45
5.5.1. Karakteristike mikoservisne arhitekture .....	46
5.5.2. Upravljanje podacima.....	47
5.5.3. Prednosti i mane mikroservisne arhitekture.....	48
6. Primjer razvoja web aplikacije temeljene na NoSQL i relacijskim bazama podataka .....	49
6.1. Opis web aplikacije .....	49

6.2. Implementacija servisa.....	50
6.2.1. Servis za rad s korisnicima.....	51
6.2.2. Servis za upravljanje proizvodima .....	53
6.2.3. Servis za upravljanje košaricama proizvoda .....	55
6.2.4. Servis za upravljanje narudžbama.....	56
6.3. Implementacija i pregled klijentske aplikacije .....	59
7. Osvrt na aplikacije temeljene na NoSQL i relacijskim bazama podataka.....	65
8. Zaključak .....	66
Popis literature .....	67
Popis slika .....	70
Popis programskih kôdova .....	72
Prilog 1 – Izvorni kôd i upute za pokretanje praktičnog primjera .....	73

# 1. Uvod

Prije pojave NoSQL-a, relacijske baze podataka predstavljale su, više-manje, jedini izbor prilikom odabira aplikacijske baze podataka. NoSQL pokret je to promijenio te danas, uz relacijske sustave za upravljanje bazama podataka (SUBP) postoji i veliki izbor NoSQL sustava temeljenih na različitim tehnologijama. No, razvoj NoSQL baza podataka nikad nije bio usmjeren na to da se zamjene relacijske baze podataka, već je fokus njihovog razvoja primarno bio na rješavanju različitih problema za koje relacijske baze podataka nikad nisu bile dizajnirane. Iz tog razloga danas se NoSQL i relacijske baze podataka mogu koristiti zajedno, u sklopu iste aplikacije, s ciljem da se upravljanje podacima što bolje prilagodi karakteristikama i načinu njihova korištenja.

S time na umu, ovaj rad sastoji se od teorijskog i praktičnog dijela kroz koje se nastoji čitatelje upoznati s ovim tehnologijama i idejom istovremenog korištenja različitih vrsta baza podataka unutar iste aplikacije. Na početku samog rada nastoji se, kroz upoznavanje NoSQL i relacijskih baza podataka, te usporedbom nekih od danas najpopularnijih postojećih relacijskih i NoSQL sustava za upravljanje podacima, dati uvid u različite značajke ovih tehnologija, kao i uvid u njihove prednosti i nedostatke. Nakon toga rad se posvećuje aplikacijama i aplikacijskim bazama podataka. Naime, prilikom njihovog razvoja postoje situacije u kojima je, ovisno o potrebama aplikacije, upravljanje podacima na najbolji način moguće ostvariti istovremenim korištenjem različitih vrsta baza podataka. Ovaj pristup se u literaturi obično krije pod pojmom „*polyglot persistence*“ te će se čitatelji u ovom dijelu rada upoznati s njegovim prednostima i nedostacima kao i s različitim arhitekturnim pristupima koji omogućavaju integraciju više različitih vrsta baza podataka unutar iste aplikacije. U tom dijelu rada posebna pažnja se posvećuje arhitekturi mikroservisa koja omogućava implementaciju „*polyglot persistence*“ arhitekture na prirodan način.

Nakon toga, u praktičnom dijelu rada prikazuje se implementacija web aplikacije temeljene na NoSQL i relacijskim bazama podataka čija arhitektura je inspirirana upravo mikroservisima, a sam rad potom završava kratkim osvrtom na aplikacije temeljene na NoSQL i relacijskim bazama podataka te zaključkom.



## 2. Relacijske baze podataka

Darwen (kao što citira Šestak (2016, str. 4)) definira relacijske baze podataka kao organizirane kolekcije podataka u kojima su podatci organizirani u skup relacija.

Ove baze podataka predstavljaju najpoznatiju i najčešće korištenu vrstu baza podataka na kojima se temelje svi današnji relacijski sustavi za upravljanje bazama podataka (SUBP), od kojih su najpoznatiji: Access, Oracle, DB2, SQL Server, PostgreSQL itd.

Same relacijske baze podataka temelje se na relacijskom modelu čije su temeljne ideje prvi puta iznesene u slavlom članku pod naslovom „*A Relational Model of Data for Large Shared Data Banks*“ kojeg je objavio Edgar Frank Ted Codd (u nastavku Codd) 1970. godine. U tom članku Codd je kao osnovnu strukturu za pohranu i prikaz podataka predložio tablicu (relaciju) s time da bi se podacima moglo upravljati pomoću operacija kao što su selekcija, projekcija, spajanje i dr., a za povezivanje redova u tablicama koristili bi se redundantni podatci (Rabuzin, 2011, str. 3).

Ukratko, glavna karakteristika relacijskih baza podataka jest njihova reprezentacija podataka u obliku **relacija** koje se sastoje od više **n-torki** (engl. *tuples*) i **atributa**. Napomenimo kako je u praksi uobičajeno da se, zbog načina njihove vizualizacije, kao zamjena za ove pojmove obično koriste pojmovi **tablica**, **red** i **stupac** (respektivno).

U nastavku ovog poglavlja opisuju se tri osnovna „stupa“ (engl. *pillars*) za koje Harisson (2015) te Gaspar i Coric (2017) kažu da predstavljaju temelj zajedničke arhitekture relacijskih baza, a to su:

1. Relacijski model podataka
2. ACID transakcije
3. SQL jezik.

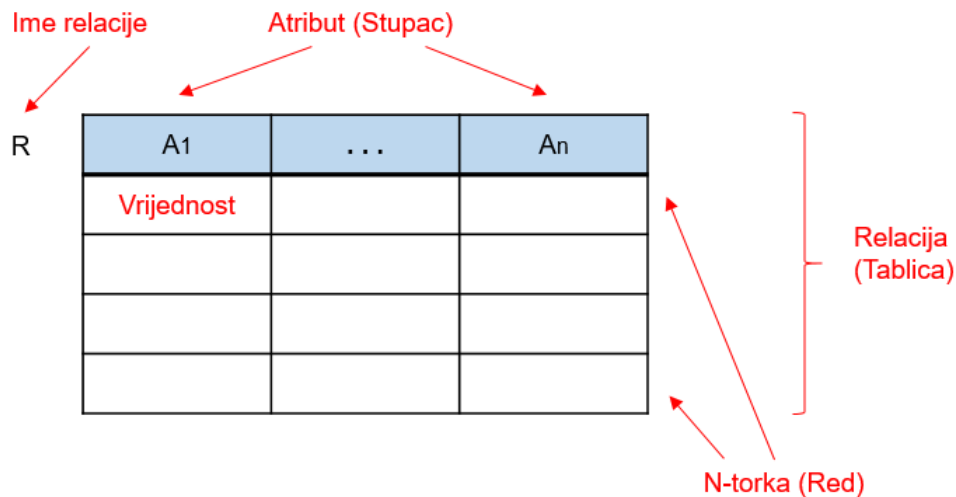
### 2.1. Relacijski model podataka

Coddov relacijski model podataka predstavlja matematičku osnovu za relacijske baze podataka. Ovaj model ne opisuje fizičku implementaciju podataka, tj. ne bavi se time kako podatci trebaju biti pohranjeni na disk ili u memoriji već opisuje kako bi podatci trebali biti prikazani korisnicima. S time na umu, Harrison (2015, str. 8) navodi kako su ključni koncepti relacijskog modela slijedeći:

- Relacija
- N-torke

- Ograničenja
- Operacije.

**Relacija**, odnosno tablica predstavlja centralni dio strukture za prikaz podataka u relacijskom modelu, a sastoji se od proizvoljnog broja **N-torki** (redova) i **atributa** (stupaca) (slika 1).



Slika 1. Strukturalni koncepti relacija u relacijskom modelu

Prema do sada viđenom moglo bi se zaključiti kako su relacije zapravo dvodimenzionalne tablice u koje se zapisuju različiti podatci, no to ne znači da je svaka dvodimenzionalna tablica ujedno i relacija u smislu relacijskog modela. Naime, A. Hoffer, Ramesh i Topi (kao što citiraju Gaspar i Coric (2017, str. 8)) navode kako tablice moraju imati slijedeća svojstva kako bi se mogle smatrati relacijama u smislu relacijskog modela podataka:

- Svaka relacija (tablica) u bazi treba imati **jedinstveno** ime.
- Vrijednost koja se nalazi na svakom križanju retka i stupca je **jednostruka** (engl. *single-valued*). To jest, svaki atribut u određenom retku tablice može biti povezan sa samo jednom vrijednošću; atributi s višestrukim vrijednostima nisu dozvoljeni u relacijama.
- Svaki redak u relaciji treba biti **jedinstven**, tj. u jednoj relaciji ne smiju biti prisutna dva identična retka.
- Svaki atribut, odnosno stupac unutar relacije treba imati **jedinstveno** ime.
- Redoslijed stupaca (s lijeva na desno) je **nevažan**. Promjena redoslijeda stupaca u relaciji neće promijeniti njeno značenje ili način njenog korištenja.
- Redoslijed redova (od vrha prema dnu) je također **nevažan**. Kao i kod stupaca, redoslijed redova u relaciji može se promijeniti ili spremati u bilo kojem slijedu.

Nadalje, **ograničenja** u relacijskom modelu podataka predstavljaju skup različitih pravila temeljem kojih se nastoji osigurati točnost i integritet podataka u bazi. Pomoću njih definiraju se različita ograničenja vezana uz **domenu** te različite vrste integriteta poput **entitetskog** i **referencijalnog integriteta** s ciljem da se ograniči koji podatci i akcije su prihvatljivi prilikom rada s bazom podataka.

Domena u ovom slučaju predstavlja skup svih vrijednosti koje pojedini atribut u nekoj tablici može poprimiti. Ovdje vrijedi napomenuti pravilo da više atributa u istoj tablici može imati istu domenu, ali svaki atribut u tablici može imati samo jednu domenu.

Entitetski integritet pak predstavlja ograničenje, odnosno uvjet integriteta prema kojem niti jedan od atributa primarnog ključa ne može poprimiti NULL vrijednost. Samo pravilo entitetskog integriteta temelji se na definiciji primarnog ključa koja nam govori kako primarni ključ predstavlja jedan ili više atributa iz relacije pomoću kojih je svaki red u toj relaciji jedinstveno identificiran.

Nadalje, referencijalni integritet predstavlja pravilo kojim se osigurava da vrijednost vanjskog ključa u određenoj relaciji mora odgovarati nekoj od vrijednosti primarnog ključa u relaciji na koju se vanjski ključ referencira, ili mora biti NULL.

Uz ograničenja, unutar relacijskog modela postoje različite **operacije** kao što su selekcija, unija, projekcija, presjek i dr., a čije izvršavanje nad jednom ili više relacija u praksi zapravo predstavlja izvršavanje različitih upita nad tablicama, a ti upiti potom korisnicima vraćaju rezultat u tabličnom obliku, tj. u obliku nove relacije.

## 2.2. ACID transakcije

Transakciju možemo definirati kao „niz naredbi koje imaju svojstvo da se izvršavaju sve (u kompletu) ili nijedna istim redom kojim su zapisane u samoj transakciji, što je poznato kao vremenska uređenost niza naredbi.“ (Rabuzin, 2011).

Transakcije se u relacijskim bazama podataka temelje na tzv. ACID principu kojim se opisuju slijedeća četiri svojstva transakcija u bazama podataka (Brkić i Mekterović, 2017):

- **Atomarnost** (engl. *Atomicity*): Transakcija se mora obaviti u cijelosti ili se uopće ne smije obaviti.
- **Konzistentnost** (engl. *Consistency*): Transakcijom baza podataka mora uvijek prolaziti iz jednog konzistentnog stanja u drugo konzistentno stanje.
- **Izolacija** (engl. *Isolation*): Kada se paralelno odvija više transakcija njihov učinak mora biti jednak kao da su se odvijale jedna iza druge.

- **Trajnost** (eng. *Durability*): Ako je transakcija obavila svoj posao, njezini efekti ne smiju biti izgubljeni uslijed kvara sustava, čak i u situaciji kada se kvar desi neposredno nakon završetka transakcije.

Temeljem ovih svojstava garantira se kako će svako neuspješno izvršavanje transakcije unutar relacijske baze podataka rezultirati njenim **poništanjem**.

## 2.3. SQL jezik

Upitni jezik **SQL** (skraćenica od engl. *Structured Query Language*) je općeprihvaćen i standardiziran upitni jezik koji se koristi za upravljanje podacima unutar relacijskih baza podataka. Sam SQL nastao je na temelju jezika SEQUEL (engl. *Structured English Query Language*) kojeg su zajedno razvili Donald D. Chamberlin i Raymond F. Boyce nakon što su njih dvojica u ranim sedamdesetim godinama prošlog stoljeća upoznali Coddov relacijski model podataka. Nakon toga, prve komercijalne implementacije SQL-a kao što su Oracle i DB2 počele su se pojavljivati tijekom kasnih sedamdesetih i ranih osamdesetih godina prošlog stoljeća. Nadalje, 1986. godine izašla i prva verzija SQL standarda kojeg su prihvatile organizacije ANSI i ISO. Nove verzije standarda potom su izlazile svakih nekoliko godina (1996., 1999., 2003., 2006. i 2008.). (Chamberlin, 2012).

Šestak (2016) u svom radu navodi kako je upravo standardizacija SQL jezika značajno olakšala njegovu primjenu u izgradnji različitih aplikacija različitih namjena.

Sam SQL jezik je **deklarativan** i **ne-proceduralan**. To znači da je fokus kod SQL jezika na tome da se njegovim korisnicima omogući da prilikom pisanja upita mogu definirati **što** žele dobiti kao rezultat upita, a sustav potom sam treba odredi **kako** će se željeni rezultat dobiti.

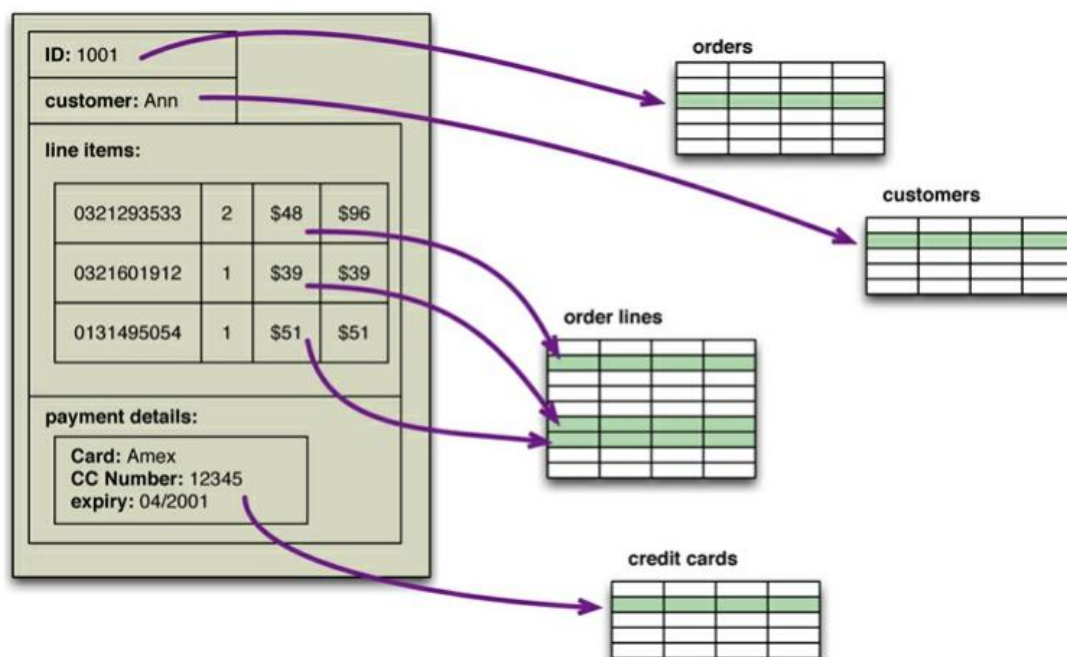
Nadalje, SQL jezik sastoji se od skupa naredbi koje se koriste za definiranje, manipuliranje i pretraživanje podataka. Temeljem toga one se tradicionalno klasificiraju u slijedeće grupe:

- 1) **Naredbe za kreiranje objekata** (engl. *Data Definition Language*, DDL); omogućavaju kreiranje, mijenjanje i brisanje postojećih objekata kao što su sheme, tablice, pogledi, indeksi itd. U ovu grupu pripadaju naredbe koje se temelje na CREATE, DROP i ALTER klauzulama.
- 2) **Naredbe za manipuliranje podacima** (engl. *Data Definition Language*, DML); omogućuju različite operacije kao što su upis, promjena i brisanje podataka. U ovu grupu pripadaju INSERT, UPDATE i DELETE naredbe.
- 3) **Naredba za postavljanje upita** (engl. *Query Language*, QL); osnovna naredba za zadavanje upita u SQL jeziku je naredba SELECT.

Iako SQL koriste svi proizvođači (engl. *vendors*) koji razvijaju relacijske sustave za upravljanje podacima on kao takav nije dovoljan za izradu kompletnih aplikacija već se u pravilu koristi zajedno s ostalim programskim jezicima (C, C#, Java, PHP i dr.) gdje ima ulogu svojevrsnog sučelja koje tim programskim jezicima omogućava pristup podacima koji su spremjeni u nekom od relacijskih sustava za upravljanje podacima.

## 2.4. Nedostaci i izazovi relacijskih baza podataka

Iako su relacijske baze podataka popularne i imaju niz prednosti, rad s njima prilikom razvoja aplikacija nije savršen. Jedan od glavnih problema s kojima se susrećemo prilikom razvoja aplikacija kada su u pitanju relacijske baze podataka jest **neusklađenost modela** (engl. *impedance mismatch*) koja postoji između strukture podataka koju želimo prikazati u samoj aplikaciji (ili memoriji računala) i relacijskog modela. Slika 4.2 prikazuje primjer neusklađenosti modela koja nastaje kada se podatci koji trebaju biti prikazani kao jedna cjelina uslijed normalizacije dijele i zapisuju u više različitih tablica u relacijskoj bazi podataka.



Slika 2. Primjer problema neusklađenosti modela (Izvor: Fowler i Sadalage, 2013)

S pojavom različitih alata za objektno-relacijsko mapiranje (engl. *Object-Relational Mapper*, ORM) kao što su Hibernate, iBATIS i sl. ovaj problem neusklađenosti modela postao je puno manje izražen, ali je još uvijek prisutan. Osim toga, pretjerano korištenje ORM alata

također može uzrokovati probleme, posebno kada su u pitanju performanse vezane uz izvršavanje kompleksnih upita.

Nadalje, Vanroose i Thillo (kao što citiraju Gaspar i Coric (2017, str. 23)) kao najveće probleme i ograničenja relacijskih baza podataka navode slijedeće:

- Zahtijevaju konverziju informacija iz njihove prirodne reprezentacije u tablice.
- Te iste informacije kasnije treba rekonstruirati iz njihove tablične reprezentacije.
- Podatke je potrebno modelirati prije nego li mogu biti pohranjeni.
- U stupce tablica mogu se pohranjivati samo slični podatci (jer shema je fiksna).
- Relacijski sustavi u većini slučajeva ne skaliraju dobro.
- Povezivanje podataka između različitih sustava je teško (jer isti podatci mogu imati različite identifikatore).
- Kompleksnu poslovnu logiku i pravila nije lagano izraziti u SQL-u.
- Performanse prilikom tzv. *fuzzy* pretraživanja i pretraživanja djelomičnih izraza su u većini slučajeva loše.
- Relacijski SUBP ne mogu efikasno spremati i validirati dokumente.

Zbog navedenih ograničenja korisnici relacijskih baza podataka su se u proteklom desetljeću susreli s različitim izazovima koji su se pojavili kao posljedica naglog razvoja Web 2.0 aplikacija i društvenih mreža te pojavom računarstva u oblaku (engl. *cloud computing*), velikih podataka (engl. *big data*) i Interneta stvari (engl. *Internet of Things*). Naime, navedeni događaji uzrokovali su pojavu i nagli rast velike količine strukturiranih, polu-strukturiranih i nestrukturiranih podataka s kojima se tradicionalni relacijski sustavi nisu mogli nositi zbog čega su velike organizacije poput Googlea, Facebooka, Amazona i dr. počele tražiti i razvijati vlastita rješenja, a što je u konačnici rezultiralo pojavom NoSQL baza podataka.

### 3. NoSQL baze podataka

NoSQL (akronim za „*Not Only SQL*“) predstavlja pojam koji se koristi prilikom opisivanja novih tehnologija u svijetu baza podataka koje su razvijene uslijed nemogućnosti tradicionalnih relacijskih sustava za upravljanje bazama podataka da zadovolje zahtjeve razvoja modernih aplikacija i informacijskih sustava. Naime, sama pojava, rast i razvoj NoSQL baza podataka uzrokovani su upravo zbog potrebe modernih organizacija kao što su Google, Amazon, Facebook i sl. za skalabilnošću njihovih sustava zaduženih za obradu velikih količina podataka koje svakodnevno nastaju kao posljedica neprestanog rasta i razvoja novih web tehnologija, kao što su npr. društvene mreže. Uz to, pojava koncepta Interneta stvari (engl. *Internet of things*) uzrokovala je nagli porast broja različitih uređaja i senzora povezanih na Internet. Sve to rezultiralo je suočavanjem tradicionalnih relacijskih baza podataka s novim izazovima vezanima uz performanse i skalabilnost, a što je, između ostalog, uzrokovalo i pojavu različitih alternativnih tehnologija u svijetu baza podataka koje omogućavaju pohranu i rad s velikim količinama podataka, te se mogu nositi s tim izazovima.

NoSQL ne predstavlja samo jednu tehnologiju ili matematički koncept, već je ovim pojmom obuhvaćen skup različitih tehnologija i baza podataka koje su razvijene s ciljem rješavanja različitih problema za koje je relacijski pristup tradicionalno bio vrlo slab. Iz tog razloga NoSQL sustavi se često koriste kao svojevrsna nadopuna, a ne kao zamjena za relacijske sustave za upravljanje bazama podataka.

Kako ne postoji jasna i generalno prihvaćena definicija NoSQL baza podataka, u nastavku ovog rada najprije su opisane neke od značajki koje su zajedničke za većinu NoSQL baza. Nakon toga napravljena je njihova kategorizacija te pregled njihovih prednosti i nedostataka.

#### 3.1. Značajke NoSQL baza podataka

Kako NoSQL obuhvaća veliki broj različitih baza podataka koje se temelje na različitim modelima podataka njihova definicija se u literaturi najčešće svodi na opisivanje različitih značajki koje su karakteristične za većinu NoSQL baza podataka, a to su (Gaspar i Coric, 2017):

- Ne-relacijski model podataka
- Nepostojanje sheme
- Distribuiranost
- CAP i BASE pristup

- Otvoreni izvorni kôd (engl. *open source code*)
- Podrška za velike podatke (engl. *big data*)
- Horizontalna skalabilnost
- Raznolikost modela podataka.

### 3.1.1. Ne-relacijski model podataka

Iako postoji veliki broj različitih implementacija NoSQL baza podataka jedna od njihovih najuočljivijih zajedničkih značajki jest upravo ta da se niti jedna od njih ne temelji na relacijskom modelu. Umjesto spremanja podataka u tablice (relacije), podatci se kod NoSQL baza podataka spremaju u različite strukture kao što su JSON, XML i sl. jer se time omogućava da struktura NoSQL baza ostane fleksibilna i denormalizirana.

Naime, na taj način se, između ostaloga, izbjegavaju često komplicirane JOIN operacije koje se provode u relacijskim bazama podataka kada su podatci koje treba dohvatiti pohranjeni u puno različitih tablica. Osim toga, relacija kao takva predstavlja poprilično restriktivnu podatkovnu strukturu u kojoj pojedini atribut nekog zapisa može imati samo jednu vrijednost, a što znači da kompleksne strukture koje bi mogle nastati ugnježđivanjem jednog zapisa u drugi kod relacijskih baza nisu dozvoljene (Gaspar i Coric, 2017).

Prilikom razvoja NoSQL baza podataka prepoznato je da je korisnicima potrebno omogućiti izvođenje operacija nad podacima koji imaju strukturu koja je kompleksnija od običnih redaka u tablici koji se kreiraju prilikom pohrane podataka koristeći relacijski model. Iz tog razloga Sadalage i Fowler (2013) navode kako se podatci u NoSQL bazama često spremaju u obliku kojeg oni nazivaju **agregat** (engl. *aggregate*). Ovaj pojam potječe iz oblikovanja vođenog domenom primjene (engl. *domain driven design* - DDD) gdje agregat predstavlja kolekciju povezanih objekta koje želimo tretirati kao jednu cjelinu. S time na umu, Sadalage i Fowler (2013) navode kako agregati predstavljaju prirodnu cjelinu temeljem koje se može lakše provoditi replikacija i fragmentacija (engl. *sharding*) baze podataka u distribuiranom sustavu.

Nadalje, valja napomenuti kako NoSQL baze podataka, za razliku od baza temeljenih na relacijskom modelu u kojemu postoje koncepti primarnog i vanjskog ključa, većinom ne pružaju poseban mehanizam za spremanje informacija o tome kako su pojedini podatci međusobno povezani. Kod NoSQL baza ovaj problem povezivanja podataka uglavnom se rješava na način da se isti podatci pohrane više puta. Flower (kao što citiraju Gaspar i Coric (2017, str. 53)) navodi slijedeće koristi ovakvog pristupa:

- **Jednostavna pohrana i dohvat podataka:** Podatci se spremaju i dohvaćaju kao jedan zapis.



- **Brzi upiti:** Podatci koji se trebaju dohvatiti prilikom izvođenja upita su spremjeni kao jedan zapis. U relacijskom modelu, gdje su podatci pohranjeni u različitim tablicama, često je potrebno za njihovu integraciju provoditi različite JOIN operacije što smanjuje brzinu izvršavanja upita.

Gaspar i Coric (2017) napominju kako ove koristi ne bi smjele biti glavni faktor prilikom donošenje odluke o korištenju NoSQL baza podataka. Naime, oba ova pristupa, relacijski (normalizirani) i NoSQL (denormalizirani) pristup predstavljaju različita rješenja vezana uz raspodjelu podataka unutar baze podataka te svaki od njih ima svoje prednosti i nedostatke, a kao glavni nedostatak ne-relacijskih modela ističe se to da obveza upravljanja istim podacima koji su više puta pohranjeni unutar baze podataka u većini slučajeva ostaje na aplikacijama, a ne na samom sustavu za upravljanje podacima.

### 3.1.2. Nepostojanje sheme

Prije spremanja bilo kakvih podataka u relacijsku bazu podataka uvijek je potrebno definirati njezinu shemu, odnosno strukturu koja nam govori koje sve tablice i atributi postoje u toj bazi te kojeg su tipa vrijednosti koje mogu biti zapisane pod određeni atribut.

Po pitanju sheme i spremanja podataka, NoSQL baze podataka imaju puno otvoreniji i manje formalan pristup, tj. one ne zahtijevaju shemu koja će definirati njihovu strukturu prije nego li podatci mogu biti spremjeni u njih. Maleković i Rabuzin (2016, str. 229) tvrde kako je činjenica da su podatci različiti i da kao takvi imaju tendenciju mijenjanja strukture pa stoga fiksnost sheme često nije prihvatljiva opcija.

Nepostojanje sheme pridonosi puno većoj fleksibilnosti oko strukture podataka koji se spremaju u bazu podataka, ali Sadalage i Fowler (2013) napominju kako ovaj pristup ima svoje probleme. Naime, nepostojanje sheme kod NoSQL baza podataka u praksi znači da će se u samome kôdu aplikacije vrlo vjerojatno stvoriti tzv. **implicitna shema** koja će se temeljiti na skupu pretpostavki o strukturi podataka s kojima aplikacija radi. Problem u ovom slučaju jest što sama baza podataka nema pristup toj implicitnoj shemi te ju ne može iskoristiti za učinkovitije spremanje i dohvaćanje podataka, te ne može primijeniti svoju validaciju nad podacima kako bi se osiguralo da druge aplikacije ne manipuliraju podacima na nedosljedan način.

Ukratko, nepostojanje sheme u samoj bazi tjera nas da shemu prebacimo u programski kôd aplikacije koja pristupa toj bazi podataka, a to može izazvati probleme u situacijama kada više aplikacija, koje razvijaju različiti ljudi, počnu pristupati istoj bazi podataka. Jedan od načina kojim se može probati riješiti ovaj problem jest kroz ograničavanje interakcije na način da imamo samo jednu aplikaciju koja će izravno komunicirati s bazom podataka, a integracija s

ostalim aplikacijama provodi se korištenjem web servisa. Drugi način temeljio bi se na jasnom definiranju kojim dijelovima agregata će pojedina aplikacija pristupati. (Fowler i Sadalage, 2013).

### 3.1.3.Distribuiranost

Većina NoSQL baza podataka dizajnirana je za distribuiranu pohranu podataka unutar sustava sastavljenog od više računala spojenih u računalnu mrežu, odnosno **grozd** (engl. *cluster*).

To zapravo znači kako je većina NoSQL baza podataka, za razliku od relacijskih baza, dizajnirana i prilagođena za rad na većem broju manjih servera sastavljenih od hardvera generalne namjene. Ovim pristupom NoSQL baze omogućavaju bolje skaliranje i veću pouzdanost sustava te olakšavaju fragmentaciju baze podataka u odnosu na tradicionalne sustave temeljene na relacijskim bazama podataka koje su generalno dizajnirane za rad na jednom velikom i specijaliziranom serveru.

Zbog distribuiranog pristupa te nepostojanja sheme NoSQL baze podataka umjesto ACID principa konzistentnosti u većini slučajeva primjenjuju tzv. BASE princip (engl. *Basic Availability, Soft-state, Eventual Consistency*). Nadalje, prema CAP teoremu (engl. *Consistency, Availability, Partition Tolerance*), NoSQL baze podataka generalno moraju zbog svoje distribuirane prirode jamčiti toleranciju na particioniranje podataka. Više detalja o samome CAP teoremu te BASE principu nalazi se u nastavku.

### 3.1.4.CAP i BASE pristup



Slika 3. CAP teorem

CAP teorem (poznat i kao Brewerov teorem), prikazan slikom 3, je teorem koji govori kako je nemoguće istovremeno zadovoljiti slijedeća tri svojstva koja su poželjna u distribuiranim sustavima, a to su: **konzistencija** (engl. *consistency*), **dostupnost** (engl. *availability*) i **tolerancija na particioniranje podataka** (engl. *partition tolerance*).

CAP teorem nam zapravo govori da kod implementacije distribuirane baze podataka (a što je čest slučaj kod rada s NoSQL bazama) gdje je tolerancija na particioniranje podataka, odnosno na ispad dijela sustava, zbog skalabilnosti neizostavna, potrebno odlučiti da li će se žrtvovati konzistentnost podataka ili njihova dostupnost. U usporedbi s time, kod tradicionalnih relacijskih sustava fokus se primarno stavlja na konzistentnost i dostupnost, a time se ujedno onemogućava jednostavna distribucija sustava na više čvorova.

Osim CAP teorema, uz NoSQL baze podataka često se vežu tzv. **BASE svojstva** (Sambolik, 2015):

- **Basically Available:** Većina podataka nam je dostupna veći dio vremena.
- **Soft state:** Označava kako se stanje sustava može promijeniti, čak i kada nemamo eksplicitne upite nad bazom podataka. Razlog tomu je što ažuriranje može doći s ažuriranjem čvora kojem pripada baza podataka.
- **Eventually consistent:** Sustav će postati konzistentan s vremenom.

Pristup upravljanju transakcijama se kod većine NoSQL baza temelji upravo na navedenom skupu BASE svojstava. U usporedbi s relacijskim sustavima koji koriste ACID pristup te se, prije svega, fokusiraju na konzistentnost podataka, sustavi temeljeni na BASE pristupu se primarno fokusiraju na dostupnost podataka. Iz tog razloga McCreary i Kelly (2014) navode kako BASE sustavi općenito spremaju nove podatke čak i u situacijama kada postoji rizik da će podatci neko kratko vrijeme biti nesinkronizirani.

S time na umu, Gaspar i Coric (2017) navode kako su BASE sustavi „optimistični“ zato jer se temelje na pretpostavci da će svi njihovi dijelovi nakon nekog vremena postati konzistentni, a što ih, u odnosu na ACID sustave, generalno čini bržima i jednostavnijima pošto se BASE sustavi ne moraju opterećivati operacijama zaključavanja i otključavanja podataka.

### 3.1.5. Otvoreni izvorni kôd

Iako danas postoje NoSQL baze podataka koje su razvijene od strane proizvođača baza podataka kao što su Oracle i Microsoft kao komercijalni proizvodi, u većini slučajeva mnoge od njih su razvijene kao projekti otvorenog kôda iza kojih se nalazi velika zajednica programera koji neprestano rade na njihovim poboljšanjima i novim funkcionalnostima.

No, Gaspar i Coric (2017) ističu kako ovakav pristup može negativno utjecati na stabilnost pojedinih sustava. Iz tog razloga se generalno ne preporuča korištenje najnovijih verzija NoSQL baza podataka, već samo verzija koje se smatraju stabilnima.

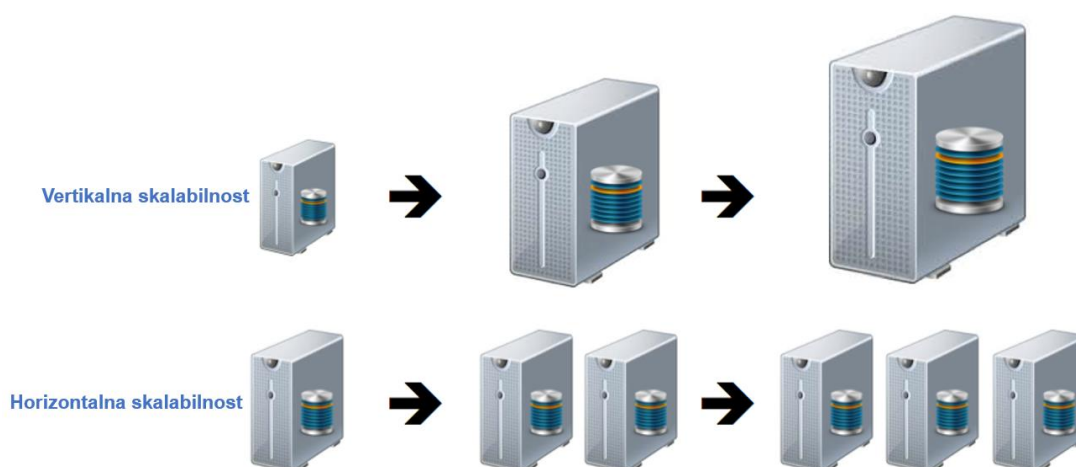
### 3.1.6. Podrška za velike podatke

Kada govorimo o NoSQL bazama podataka često se kao jedan od razloga njihove pojave veže uz tzv. problem velikih podataka (engl. *big data*). Jedna od definicija ovog problema, kao i samog pojma velikih podataka, temelji se na skraćenici **3V** koja dolazi od engl. *Volume*, *Velocity* i *Variety*.

Prvi V (*Volume*) označava velike količine podataka. Ovdje se govori o količinama podataka kod kojih relacijski sustavi više nemaju (ne daju) zadovoljavajuće performanse. Drugi V dolazi od *Velocity* te označava veliku brzinu kojom podatci nastaju te koliko brzo ih je potrebno obraditi, kao i brzinu kojom se same vrijednosti podataka, kao i njihova struktura, mijenjaju. Treći V (*Variety*) označava različitost podataka, tj. u današnjem svijetu velikih podataka čest je slučaj da podatci nisu uvijek potpuni (jer nisu sve informacije uvijek dostupne) i razlike među njima mogu biti znatne. To je posljedica različitih tehnologija i sustava iz kojih podatci dolaze (izvor podataka mogu biti društvene mreže, razni uređaji i senzori, strukturirani izvori itd.). (Maleković i Rabuzin, 2016).

### 3.1.7. Horizontalna skalabilnost

Skalabilnost predstavlja mogućnost sustava da se prilagodi promjenama koje nastaju uslijed povećanog opterećenja s ciljem očuvanja brzine, dostupnosti i pouzdanosti.



Slika 4. Vertikalna i horizontalna skalabilnost (Izvor: „How is scalability achieved?“, 2012)

Općenito, po pitanju hardvera postoje **vertikalna** (engl. *scale-up*) i **horizontalna** (engl. *scale-out*) skalabilnost. Kao što je ilustrirano slikom 4, vertikalna skalabilnost se općenito postiže nadogradnjom hardvera poslužitelja kroz ugradnju boljeg procesora u poslužitelj, ugradnjom više radne memorije, ugradnjom SSD diska i sl., ili pak zamjenom trenutnog poslužitelja s novim, većim (i skupljim) poslužiteljem. Za razliku od tog pristupa, povećanje performansi se kod horizontalnog skaliranja postiže dodavanjem novih poslužitelja (čvorova) u računalnu mrežu.

Relacijske baze podataka općenito su vertikalno skalabilne, a većina NoSQL baza podataka je horizontalno skalabilna što znači da podatci, kao i samo opterećenje nad bazom podataka, mogu biti raspodijeljeni između pojedinih čvorova u mreži. Gaspar i Coric (2017) navode kako je iz tog razloga karakteristika većine NoSQL baza podataka da podržavaju **automatsku distribuciju podataka** na nove poslužitelje kada se dodaju u sustav, a čime se, između ostaloga, postiže i poboljšanje performansi.

### 3.1.8. Raznolikost modela podataka

Nakon puno neuspjelih pokušaja da se pomoću tradicionalnih relacijskih sustava za upravljanje bazama podataka pokušaju riješiti različiti problemi velikih podataka za koje oni nisu dizajnirani, te sama pojava i rast NoSQL baza podataka potvrdili su da ne postoji jedinstveno rješenje za sve probleme koji su vezani uz upravljanje podacima korištenjem baza podataka (Gaspar i Coric, 2017).

Iz tog razloga se razvoj NoSQL baza podataka od samog početka NoSQL pokreta temeljio na pretpostavci da se za rješavanje različitih problema trebaju koristiti različiti modeli podataka. Rezultat toga jest da danas postoji jako veliki broj (na stotine) NoSQL baza koje se razlikuju u strukturi i namjeni, a smatraju se dijelom NoSQL svijeta.

Općenito, NoSQL baze podataka temelje se na jednom od slijedeća četiri modela podataka: **ključ-vrijednost**, **graf**, **dokument** i **stupčani**. U nastavku rada opisati će se četiri osnove vrste NoSQL baza podataka koje se temelje na ovim modelima podataka.

## 3.2. Vrste NoSQL baza podataka

Prema Hillsu (2016), sve NoSQL baze podataka, u svojoj suštini, koriste podatkovne strukture kao što su stabla, povezane liste, redovi, polja i dr. s ciljem organizacije podataka na način kako bi se postigao što brži pristup do njih. Ono što ih razlikuje jest njihov model podataka, odnosno način na koji one predstavljaju te različite podatkovne strukture svojim korisnicima. Prema tome, uobičajena podjela NoSQL baza podataka temelji se na različitim

modelima podataka koji su bili navedeni u poglavlju 4.1.6. S time na umu, postoje četiri osnovne vrste NoSQL baza podataka, a to su:

1. Ključ/vrijednost baze podataka
2. Dokument-baze podataka
3. Stupčane baze podataka
4. Graf-baze podataka.

### 3.2.1. Ključ/vrijednost baze podataka

Ključ/vrijednost baze podataka predstavljaju najjednostavniji oblik NoSQL baza podataka koji se temelji na jednostavnom modelu podataka (prikazan slikom 2) koji je svojom strukturom sličan rječniku jer se sastoji od parova oblika (**ključ, vrijednost**). Ključ u ovom slučaju najčešće predstavlja neku tekstualnu vrijednost kojoj može biti pridružena vrijednost bilo kojeg tipa. U praksi ključ može biti u raznim formatima „kao što je specifikacija direktorija za neku datoteku, niz znakova koji predstavlja nekakav jedinstven kôd, poziv REST mrežnog servisa, internetska adresa i sl.“ (Stojanović, 2016).

Ključ	Vrijednost
K1	AAA, BBB, CCC
K2	AAA, BBB
K3	AAA, 2, 01/01/2018

Slika 5. Ključ/vrijednost model podataka (autorski rad)

Kod ove vrste baza podataka različite vrijednosti (kao što su tekst, slike, video i sl.) obično se spremaju i vraćaju iz baze u obliku BLOB<sup>1</sup> tipa podataka te je odgovornost na samoj aplikaciji da prilikom dohvaćanja podataka iz baze utvrdi o kakvom tipu podatka se zapravo radi (McCreary i Kelly, 2014).

Stojanović (2016) navodi kako, za razliku od relacijskih baza podataka, ove baze podataka nemaju upitni jezik već samo omogućavaju pronalaženje, dodavanje i uklanjanje parova na osnovu ključa. S time na umu, ključ/vrijednost baze podataka općenito podržavaju slijedeće tri operacije (McCreary i Kelly, 2014):

---

<sup>1</sup> *Binary Large Object* (BLOB) – predstavlja kolekciju binarnih podataka spremljenih u obliku jednog zapisa unutar baze podataka.

1. `put(ključ, vrijednost)` – dodaje novi par (*ključ, vrijednost*) ili ažurira vrijednost već postojećeg ključa.
2. `get(ključ)` – vraća vrijednost pridruženu zadanom ključu iz baze, može javiti grešku ukoliko zadani ključ ne postoji.
3. `delete(ključ)` – uklanja par (*ključ, vrijednost*) iz baze, može javiti grešku ukoliko zadani ključ ne postoji.

Uz ove operacije, za ključ/vrijednost baze podataka općenito vrijede i slijedeća dva pravila (McCreary i Kelly, 2014):

1. **Različiti ključevi** – Svi ključevi u određenoj ključ/vrijednost bazi podataka moraju biti jedinstveni.
2. **Nema upita baziranih na vrijednostima** – Ključ/vrijednost baze podataka ne dopuštaju upite bazirane na vrijednostima, tj. u ovakvoj bazi podataka ne možemo koristiti vrijednost za traženje određenog para (*ključ, vrijednost*).

Neki od primjera ove vrste baza podataka su: *Redis, Memcached, Hazelcast, Ehcache, Riak KV, Aerospike, Oracle NoSQL* itd.

### 3.2.2. Dokument-baze podataka

Dokument-baze podataka predstavljaju najpoznatiju, najfleksibilniju i najčešće korištenu vrstu NoSQL baza. Njihova se popularnost, između ostalog, temelji na poprilično intuitivnom modelu podataka čiji osnovni element je **dokument** (slika 6) koji se može definirati kao „uređeni skup ključeva s pridruženim vrijednostima“ (Stojanović, 2016). Dokument i ključ/vrijednost baze podataka slične su u smislu da svaki dokument spremljen u dokument bazi mora imati jedinstveni identifikator (*ID*), baš kao što i svaka vrijednost spremljena u ključ/vrijednost bazi podataka mora imati pripadajući jedinstveni ključ.

```
{
  id: "fvdh47jfd7b",
  ime: "Pero",
  prezime: "Peric"
};

{
  id: "kn8jvf5stt",
  osobni_podaci: { ime: "Pero", prezime: "Peric" },
  sportovi: { "nogomet", "skijanje" }
};
```

Slika 6. Primjeri dokumenata u JavaScript notaciji (autorski rad)

Drugi (desni) primjer dokumenta sa slike 6 prikazuje kako dokumenti, „u stvari, mogu imati hijerarhijsku strukturu zato jer jedan dokument može biti dio drugog, što korisnicima

omogućava fleksibilnost i nudi intuitivan model podataka, posebno ako dolaze iz objektno-orijentiranog razvoja“ (Stojanović, 2016).

Dokumenti koji imaju neku zajedničku svrhu se u ovim bazama podataka obično spremaju u tzv. **kolekcije**. Njihova uloga slična je tablicama koje imamo kod relacijskih baza podataka. U tom smislu na svaki dokument u kolekciji može se gledati kao na redak u relaciji (tablici) gdje je glavna razlika u tome što svaki dokument u kolekciji, za razliku od redaka u nekoj tablici, može imati drugačiju strukturu. Općenito, u kolekcije se stavljaju dokumenti slične strukture kako bi klijenti znali što je spremljeno u kojem dokumentu.

Fowler i Sadalage (2013) navode kako dokument-baze podataka općenito spremaju (i vraćaju) podatke u formatima kao što su XML, JSON, BSON (*Binary JSON*) i sl. Razlog tome je što jednostavnost formata kao što je JSON omogućava prikazivanje bilo kakvih podataka. Uz to, većina programskih jezika omogućava direktnu konverziju objekata u takve formate te se na taj način izbjegava problem oko neusklađenosti modela između baze i aplikacijskog kôda koja postoji kod relacijskih baza podataka, te se samim time ujedno izbjegava upotreba ORM (*object-relational mapper*) alata.

Jedna od glavnih značajki dokument-sustava je ta da se svi podatci unutar dokumenta automatski indeksiraju prilikom dodavanja novog dokumenta kako bi se omogućilo brzo pretraživanje svih dokumenata spremljenih unutar baze podataka. Na taj način atributi svih dokumenata postaju poznati te se značajno smanjuje vrijeme traženja dokumenata s istim atributom. Nedostatak ovog pristupa je taj što indeksi unutar baze mogu postati dosta veliki. (Gaspar i Coric, 2017).

Svi dokument-sustavi podržavaju određeni API ili pak poseban upitni jezik za postavljanje upita i modifikaciju podataka, no same mogućnosti zadavanja upita često se značajno razlikuju od sustava do sustava. No, u usporedbi s ostalim vrstama NoSQL baza, dokument-baze podataka generalno imaju veće mogućnosti indeksiranja i zadavanja kompleksnijih upita.

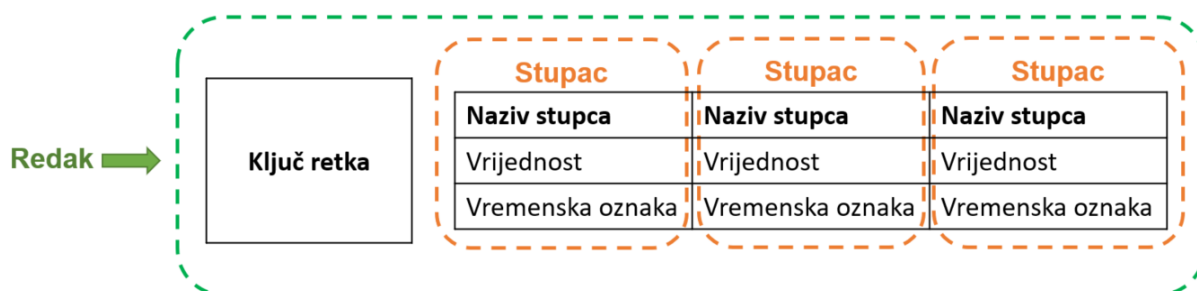
Većina dokument-baza podataka se, u usporedbi s relacijskim bazama, puno više fokusira na skalabilnost. Glavni razlog za to jest taj što model podataka temeljen na dokumentima omogućava puno lakše raspoređivanje podataka na više različitih servera. Omogućavanje veće skalabilnosti ujedno je i jedan od razloga zašto dokument-sustavi, za razliku od relacijskih baza, ne podržavaju operacije pridruživanja (JOIN). (Stojanović, 2016).

Najpopularniji predstavnici ove vrste NoSQL baza podataka su: *MongoDB*, *Couchbase*, *CouchDB*, *Firebase Realtime Database*, *RethinkDB* i dr.



### 3.2.3. Stupčane baze podataka

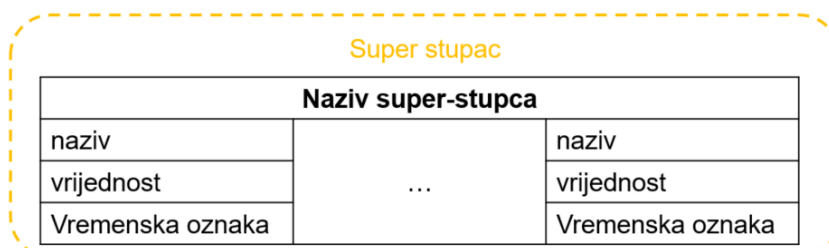
U usporedbi s relacijskim bazama podataka u kojima se podatci spremaju u redove iste strukture, kod stupčanih baza podataka naglasak se stavlja na stupce te se omogućava da svaki redak u ovoj vrsti baza podataka može imati vlastitu strukturu, s proizvoljnim brojem stupaca koji se mogu jednostavno dodavati i brisati, bez utjecaja na ostale redove.



Slika 7. Struktura retka u stupčanim bazama podataka (Prema: „What is a Column Store Database?“, 2016)

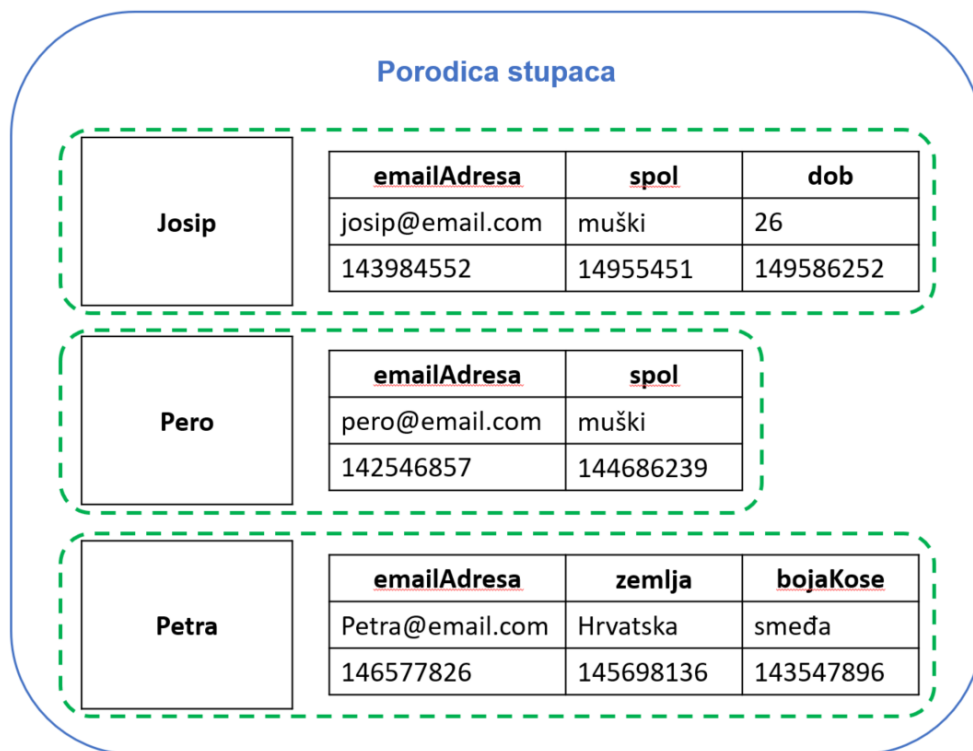
Slika 7 prikazuje strukturu redova u stupčanim bazama podataka. Prema toj slici možemo vidjeti kako svaki redak u stupčanim bazama podataka općenito ima svoj **jedinstveni ključ** (tzv. **ključ retka**, engl. *row key*) te se sastoji od određenog broja **stupaca** koji su ograničeni na taj redak, tj. ne proširuju na ostale redove, kao što to rade stupci u relacijskim bazama podataka. Nadalje, svaki stupac u stupčanim bazama podataka općenito se sastoji od **naziva**, pripadajuće **vrijednosti** i **vremenske oznake** (engl. *timestamp*). Pomoću vremenskih oznaka se u stupčanim bazama podataka označava kada je neka vrijednost unesena u bazu te se omogućava spremanje više različitih verzija podataka u istom stupcu, a što korisnicima omogućava da prate promjene vrijednosti podataka kroz povijest.

Nadalje, Stojanović (2016) navodi kako se stupci kod ove vrste baza podataka mogu kombinirati u veće podatkovne strukture, tzv. **super-stupce**, koji se sastoje od sortiranog niza stupaca te nemaju vlastitu vremensku oznaku (slika 8).



Slika 8. Struktura super-stupaca (autorski rad)

Uz to, stupci se kod stupčanih baza podataka obično grupiraju u strukture koje se nazivaju **porodice stupaca** (engl. *column family*). Svrha ovih porodica stupaca je, u većini slučajeva, usporediva s tablicama iz relacijskog modela, tj. porodice stupaca obično se ponašaju kao kontejneri za sortirani niz redova. Primjer porodice stupaca koja se sastoji od 3 retka prikazana je na slici 9.



Slika 9. Prikaz primjera porodice stupaca (Prema: „What is a Column Store Database?“, 2016)

Kod stupčanih baza podataka obično se spominje i pojam **keyspace**. On zapravo predstavlja poseban objekt koji se koristi za grupiranje pojedinih obitelji stupaca, tj. predstavlja svojevrsni kontejner za više različitih obitelji stupaca. Npr., *keyspace* omogućava zajedničko grupiranje svih obitelji stupaca koji unutar sustava pripadaju istoj aplikaciji. U tom smislu *keyspace* ima sličnu ulogu kao i baza podataka unutra nekog relacijskog SUBP.

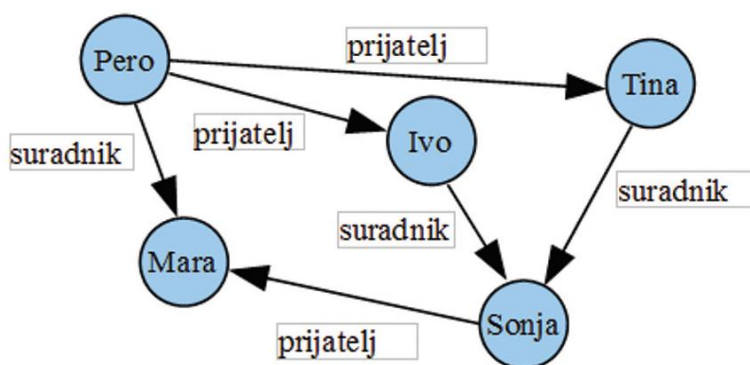
Stojanović (2016) navodi kako je osnovna prednost sustava temeljenih na stupčanim NoSQL bazama podataka ta da su pogodni za obradu vrlo velikih količina podataka jer zbog svojih tehničkih karakteristika omogućavaju poprilično efikasno učitavanje i analiziranje svih vrijednosti jednog stupca (što je također slučaj i s dodavanjem ili brisanjem stupaca).

Glavni predstavnici ove vrste baza podataka su: *Cassandra*, *HBase*, *Accumulo*, *Google Cloud Bigtable* i dr.

### 3.2.4. Graf-baze podataka

Graf-baze podataka koriste koncept **grafa** kao model podataka. U ovom slučaju graf predstavlja strukturu podataka koja se sastoji od **čvorova** i **lukova** koji povezuju te čvorove. Maleković i Rabuzin (2016) navode kako bi čvorovi kod ove vrste baza podataka u osnovi trebali služiti pohrani entiteta, odnosno njihovih atributa, dok bi lukovi trebali predstavljati poveznice između tih entiteta.

Na primjer, na slici 10 prikazan je primjer grafa koji predstavlja grupu ljudi među kojima postoje poveznice *prijatelj* i *suradnik*.



Slika 10. Primjer grafa (Izvor: Stojanović, 2016)

Tipično je da čvorovi i lukovi kod graf-baza podataka sadrže **svojstva** koja se sastoje od skupova *ključ/vrijednost* parova. Fowler i Sadalage (2013) navode kako se dodavanjem svojstava poveznicama koje povezuju čvorove omogućava dodavanje svojevrzne inteligencije vezama koje postoje između pojedinih entiteta.

Korištenjem različitog modela podataka, u usporedbi s ključ/vrijednost, stupčanim i dokument bazama podataka, graf baze podataka omogućavaju efikasno spremanje veza između podataka, tj. ova vrsta baza podataka specijalizirana je za rad s podacima koji su međusobno jako povezani. (Grolinger, Higashino, Tiwari, i Capretz, 2013).

Pošto graf-baze podataka eksplicitno čuvaju veze među čvorovima, izvršavanje upita kod sustava koji se temelje na ovoj vrsti baza podataka sastoji se od praćenja **putanja**, odnosno veza koje postoje između pojedinih čvorova. Ti sustavi su u tome poprilično efikasni jer rade s direktnim vezama među čvorovima, to jest oni te veze ne moraju tražiti. (Stojanović, 2016).

Nadalje, pošto su čvorovi podataka unutar graf-baza međusobno povezani većina sustava koji se temelje na ovoj vrsti baza podataka obično nije horizontalno skalabilna, tj. ne podržava distribuciju čvorova na različite servere. Iz tog razloga neki sustavi temeljeni na ovoj

vrsti baza podataka, kao što je npr. Neo4j, podržavaju ACID svojstva kojima osiguravaju da podatci unutra baze uvijek budu konzistentni.

Uz već spomenuti *Neo4j*, neki od najpoznatijih sustava temeljenih na graf-bazama podataka su: *Giraph*, *JanusGraph*, *InfiniteGraph*, *Dgraph*, i dr.

### 3.3. Prednosti i nedostaci NoSQL baza podataka

Zahvaljujući svojim značajkama NoSQL baze podataka imaju brojne prednosti u usporedbi s relacijskim bazama podataka od kojih valja istaknuti **moгуćnosti upravljanja velikim količinama strukturiranih, polu-strukturiranih i nestrukturiranih podataka, bolje performanse** u odnosu na relacijske baze podataka, **podrška za agilni pristup razvoju** s naglaskom na agilne sprintove, brze iteracije te česte promjene kôda, **prijateljski pristup objektno-orijentiranom programiranju, podrška za efikasniju i horizontalno skalabilnu arhitekturu** itd. („Advantages Of NoSQL“, n.d.).

Osim navedenih prednosti NoSQL baze podataka imaju i određene nedostatke u odnosu na relacijske baze podataka. Naime, činjenica je da NoSQL sustavi predstavljaju relativno **mladu tehnologiju** koji svojom zrelošću i stabilnošću još uvijek nisu na razini relacijskih sustava za upravljanje bazama podataka. Nadalje, često se kad NoSQL baza podataka spominje **problem nedostatka standardizacije** između različitih NoSQL sustava. Naime, velika raznolikost u dizajnu i upitnim jezicima između različitih NoSQL sustava ujedno predstavlja **strmiju krivulju učenja** za programere pošto se znanje o korištenju jednog NoSQL sustava u većini slučajeva ne može iskoristiti prilikom rada s nekim drugim NoSQL sustavom (Tozzi, 2016). Osim toga, NoSQL baze podataka općenito zaostaju za relacijskim bazama podataka po pitanju **očuvanja konzistentnosti podataka** pošto većina njih ne podržava ACID transakcije.

## 4. Pregled i usporedba pojedinih implementacija relacijskih i NoSQL baza podataka

U ovom poglavlju opisuju se i međusobno uspoređuju MySQL, kao predstavnik relacijskih sustava, te Redis, MongoDB, Cassandra i Neo4j kao predstavnici različitih NoSQL SUBP-a.

### 4.1. MySQL

MySQL<sup>2</sup> je predstavnik jednog od najpopularnijih sustava otvorenog kôda za upravljanje relacijskim bazama podataka koji pruža sveobuhvatnu podršku za razvoj većine aplikacija. Danas se ovaj sustav nalazi u vlasništvu Oracle organizacije te je dostupan u nekoliko različitih verzija. Uz besplatnu (*MySQL Community Edition*) verziju, postoje i različite komercijalne verzije ovog sustava u sklopu kojih se nalazi i posebna verzija MySQL-a (*MySQL NDB Cluster*) namijenjena za korištenje u velikim distribuiranim sustavima. MySQL sustav razvijen je u C i C++ programskim jezicima te je dostupan na različitim platformama.

MySQL sustav općenito se smatra brzim, pouzdanim, skalabilnim te jednostavnim za korištenje, a moguće ga je bez problema pokrenuti na raznim stolnim i prijenosnim računalima i serverima s različitim konfiguracijama hardvera.

Što se tipova podataka tiče, MySQL omogućava definiranje različitih tipova podataka za podatke koji će biti pohranjeni u pojedina polja u nekoj tablici. S time na umu, MySQL podržava pohranu podataka različitih **tekstualnih** (CHAR, VARCHAR, TINYTEXT i dr.), **numeričkih** (TINYINT, SMALLINT, MEDIUMINT, INT i dr.) te **datum/vrijeme** (DATE, DATETIME, TIMESTAMP i dr.) tipova podataka. Valja napomenuti kako se odabirom ispravne vrste podataka znatno može utjecati na performanse same MySQL baze podataka. (Mišić, 2009).

Uz navedene tipove podataka, trenutna verzija MySQL sustava (v8.0) pruža podršku i za JSON tip podataka, tj. omogućava efikasnu pohranu JSON dokumenata u posebna JSON polja koja omogućavaju automatsku validaciju JSON dokumenata prilikom njihove pohrane i pristup pojedinim elementima pohranjenog dokumenta tijekom njegovog čitanja.

Poput većine velikih sustava za upravljanje relacijskim bazama podataka MySQL također dolazi s velikim brojem različitih operatora i ugrađenih funkcija, a ima i mogućnost

---

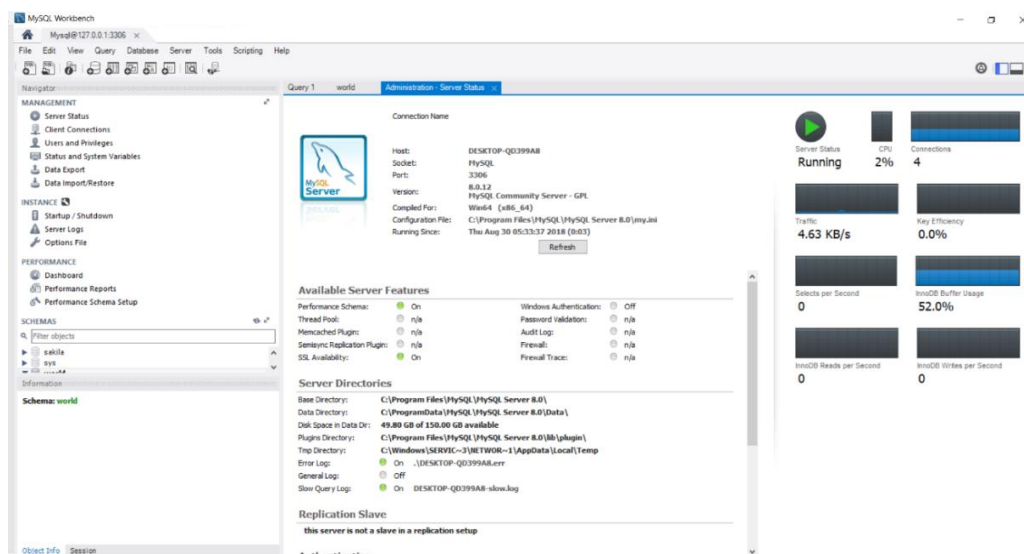
<sup>2</sup> <https://www.mysql.com/>

kreiranja korisnički definiranih funkcija koje su dostupne za korištenje u sklopu SQL naredbi. Uz to, MySQL podržava pohranjene procedure, okidače, poglede, kursore, ima podršku za replikaciju i partitioniranje podataka i još mnogo toga.

Nadalje, sama instalacija MySQL sustava dolazi s mnogo različitih klijentskih programa koji se mogu povezivati na MySQL server te koristiti za izvršavanje raznih zadaća. Neki od njih su:

- **mysql** – alat za izvršavanje SQL naredbi preko komandne linije;
- **mysqladmin** – alat za izvršavanje različitih administrativnih operacija;
- **mysqlcheck** – klijent za provjeru, popravak, analizu i optimizaciju tablica;
- **mysqldump** – klijent za izvoz MySQL baze podataka u SQL, XML ili tekstualnu datoteku; i mnogi drugi.

Za ovaj sustav dostupni su i različiti alati koji omogućavaju interakciju s MySQL serverom preko grafičkog korisničkog sučelja. Jedan od najpoznatijih takvih alata je *MySQL Workbench* (slika 11) koji kroz svoje sučelje omogućava izvršavanje SQL upita, dizajniranje, modeliranje, administraciju i migraciju MySQL baze podataka i sl.



Slika 11. MySQL Workbench sučelje

Uz sve navedeno, uz pomoć *X Plugin* dodatka trenutna verzija MySQL sustava omogućava da se ovaj sustav osim na tradicionalan (SQL) način može koristiti i kao dokument-baza podataka. Općenito, na ovaj i slične načine proizvođači različitih sustava relacijskih baza podataka danas nastoje da kroz ugradnju NoSQL značajki u svoje postojeće proizvode zadovolje potrebe što većeg broja korisnika te da budu konkurentni i u svijetu NoSQL baza podataka.

## 4.2. Redis

Redis<sup>3</sup> (skraćenica od engl. *REmote DIctionary Service*) je primjer ključ/vrijednost NoSQL sustava za upravljanje podacima otvorenog kôda. Ovaj sustav razvijen je u ANSI C programskom jeziku, a njegova prva verzija pojavila se 2009. godine. Danas rad s ovim sustavom podržava većina programskih jezika.

Jedna od glavnih karakteristika ovog sustava jest što za vrijeme rada sprema podatke primarno **u memoriju** te ih, ovisno o konfiguraciji, samo povremeno zapisuje na disk. Ovaj pristup omogućava Redisu veliku brzinu, ali može prouzročiti gubitak podataka ukoliko, npr. dođe do gubitka struje ili iznenadnog pada sustava.

Uz tekst, Redis podržava i pohranjivanje vrijednosti u obliku različitih apstraktnih tipova podataka kao što su: liste, skupovi, sortirani skupovi itd. Za svaki podržani tip podataka ovaj sustav nudi veliki broj različitih operacija kao što je npr. dodavanje znakova u postojeći niz znakova, dodavanje elemenata u listu, određivanje presjeka, unije i razlike skupa itd.

Redis je SUBP koji podržava replikaciju što ga čini skalabilnim. Uz to, ovaj sustav ima ugrađenu podršku za transakcije, LUA skriptni jezik, ključeve koji imaju vrijeme isteka važnosti itd. U usporedbi s poznatim relacijskim sustavima, kao što je npr. MySQL, Redis ima puno manje značajki te kao takav nije zamišljen da se koristi kao primarna baza podataka u aplikacijama. Naime, zbog svojih karakteristika ovaj sustav se u praksi često primjenjuje u ulozi poslužitelja za privremenu pohranu podataka (engl. *cache server*) u situacijama kada je potrebno ostvariti što brži pristup do podataka.

Za izravnu interakciju s bazom Redis svojim korisnicima nudi komandnu liniju koja omogućava slanje različitih naredbi na server te pregled odgovora koji dolaze s servera unutar terminala. Osim ovog, postoji i nekoliko različitih programa kao što su *Redsmin*, *Redis Commander*, *Redis Desktop Manager* i ostali koji omogućavaju interakciju s Redisom preko grafičkog sučelja.

## 4.3. MongoDB

MongoDB<sup>4</sup> predstavlja skalabilnu i fleksibilnu implementaciju dokument-baze podataka koju, uz to, karakteriziraju otvoreni kôd, visoke performanse, dostupnost itd. Sam naziv ovog sustava dolazi od engl. *humongous* što se može prevesti kao „iznimno velik“. Prva javna verzija

---

<sup>3</sup> <https://redis.io/>

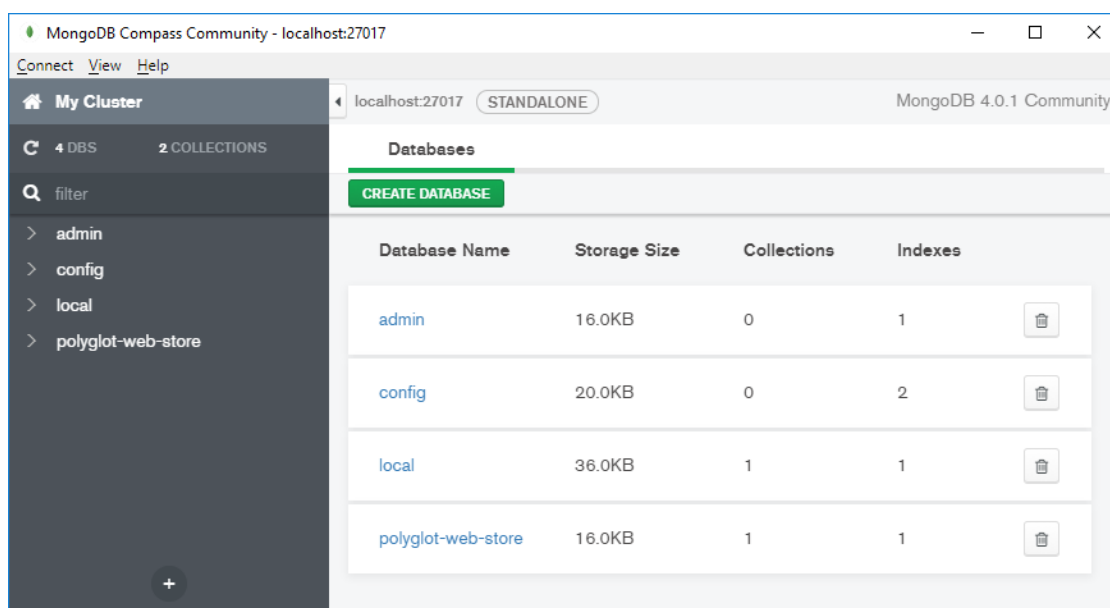
<sup>4</sup> <https://www.mongodb.com/>

ovog sustava pojavila se 2009. godine te danas ovaj sustav predstavlja najpoznatiju NoSQL bazu podataka.

Svi podatci u ovom sustavu pohranjuju se u binarnoj formi JSON dokumenata, poznatoj kao BSON te se grupiraju u kolekcije koje se ponašaju kao tablice u relacijskim bazama podataka.

MongoDB omogućava horizontalnu skalabilnost kroz fragmentaciju (engl. *sharding*) podataka na više različitih servera, a visoke performanse postiže kroz podršku za **ugniježdene dokumente** i **indeksiranje**. Podrškom za ugniježdene dokumente smanjuje se broj operacija čitanja i pisanja podataka, a indeksiranje omogućava brže izvođenje upita unutar samog sustava. Uz to, ovaj sustav ima bogati **upitni jezik** koji omogućava brzo izvršavanje operacija kao što su kreiranje, čitanje, ažuriranje te brisanja dokumenata, pretraživanje teksta unutar samih dokumenata, agregaciju podataka itd.

Za interakciju s bazom podataka MongoDB sustav nudi komandnu liniju (*mongo Shell*) te alat MongoDB Compass (slika 12) koji omogućava interakciju s MongoDB sustavom preko grafičkog korisničkog sučelja.



Slika 12. Sučelje alata MongoDB Compass

MongoDB svojim mogućnostima nudi korisnicima zadovoljavajući kompromis između distribuiranosti i sposobnosti zadavanja kompleksnih upita nad podacima. Iz tog razloga danas ovo rješenje predstavlja vrlo popularan izbor prilikom odabira baze podataka za razne web projekte koji se bave velikim količinama podataka.



## 4.4. Cassandra

Cassandra<sup>5</sup> je predstavnik stupčanih baza podataka. Ovaj sustav je također otvorenog kôda, napisan u Java programskom jeziku te danas predstavlja jedan od projekata najviše razine Apache organizacije.

Sam Cassandra sustav dizajniran je da bude u potpunosti distribuiran na način da nema jedinstvenu točku otkazivanja (engl. *single point of failure*). Uz to, kako bi se postigla otpornost na greške, ovaj sustav podržava automatsku replikaciju podataka na više različitih čvorova u mreži te podržava replikaciju podataka između različitih podatkovnih centara.

Cilj ovog sustava je da bude što je moguće više skalabilan, a to postiže na način da omogućava linearno upravljanje performansama čitanja i pisanja podataka koje se postiže dodavanjem i uklanjanjem čvorova iz sustava.

Uz navedene karakteristike, na službenim web stranicama ovog sustava navodi se kako je jedna od njegovih karakteristika i trajnost podataka, čak i u slučajevima kada cijeli podatkovni centar ispadne iz sustava, a što Cassandru čini pogodnom za korištenje u aplikacijama kod kojih se ne smije desiti gubitak podataka prilikom ispada pojedinih dijelova iz sustava.

Kao sučelje za pristup ovom sustavu koristi se CQL (od engl. *Cassandra Query Language*) upitni jezik koji je svojom sintaksom sličan SQL-u. Cassandra također dolazi s komandnom linijom koja korisnicima omogućava izravnu s sustavom kroz CQL jezik.

Cassandra je jedan od NoSQL sustava koji se danas aktivno koriste u organizacijama kao što su Apple, Netflix, eBay i dr. kod kojih količine podataka u nekim slučajevima prelaze 10 PB te koji su rasprostranjeni na desecima tisuća čvorova, a broj zahtjeva prema bazi u jednom danu može biti u stotinama milijuna (i više od toga).

## 4.5. Neo4j

Neo4j<sup>6</sup> predstavlja najpopularniji NoSQL sustav temeljen na graf-bazama podataka. Razvijen je u Java programskom jeziku te je također otvorenog kôda.

Za razliku od ostalih NoSQL implementacija pregledanih u ovome poglavlju, Neo4j sustav koristi ACID princip kako bi osigurao integritet podataka u bilo kojem trenutku, a da pritom zadržava prednosti kao što su fleksibilna shema, visoke performanse, dostupnost i sl.

---

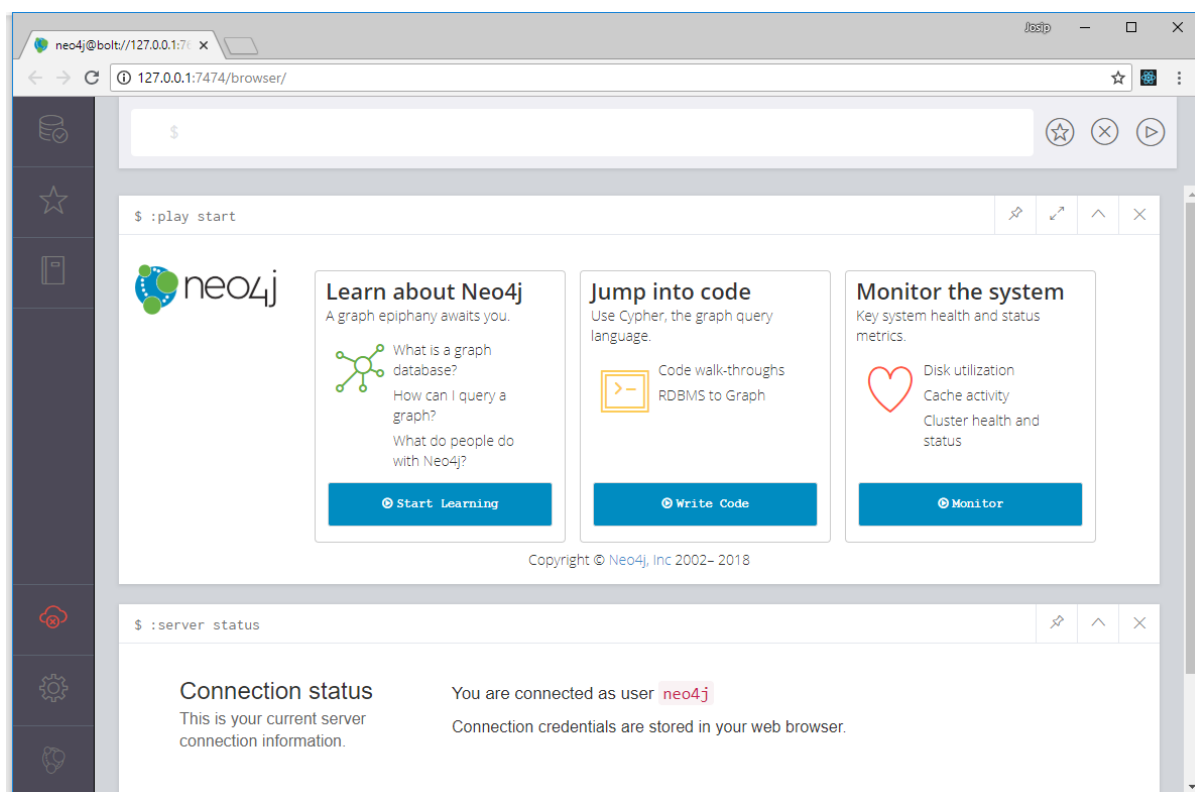
<sup>5</sup> <http://cassandra.apache.org/>

<sup>6</sup> <https://neo4j.com/>

Neo4j je od početka građen da bude graf-baza podataka. To jest, arhitektura ovog sustava je dizajnirana i optimizirana za upravljanje, pohranu i brzo pretraživanje podatkovnih čvorova i veza koje postoje između njih. Veze između podataka su u ovom sustavu na prvom mjestu. Naime, operacije koje se u relacijskim bazama podataka izvode prilikom korištenja JOIN ključne riječi, čije performanse eksponencijalno postaju sve lošije s porastom broja odnosa koje je potrebno pronaći, Neo4j provodi navigacijom i kretanjem između čvorova, pri čemu su performanse linearne. (Neo4j Inc., 2018).

Neo4j pruža podršku za *Cypher* upitni jezik koji je dizajniran za efikasno postavljanje upita te ažuriranje podataka unutar graf-baza podataka.

Izravnu interakciju s bazom podataka Neo4j omogućava kroz komandnu liniju (*Neo4j Shell*) koja omogućava izvršavanjem *Cypher* upita te kroz posebni alat naziva **Neo4j Browser** (slika 13) koji dolazi u obliku web sučelja koje je korisnicima dostupno na lokalnoj IP adresi <http://127.0.0.1:7474/>, nakon instalacije Neo4j Servera. Osim što omogućava grafički pristup Neo4j bazi podataka, Neo4j Browser omogućava izvršavanje različitih *Cypher* upita, uvoz podataka u bazu i sl.



Slika 13. Neo4j Browser sučelje

## 4.6. Usporedba pregledanih rješenja

Općenito, svaka baza podataka podržava pohranu i dohvaćanje podataka. No, između relacijskih i NoSQL baza podataka, kao i između pojedinačnih implementacija NoSQL baza, postoje mnoge razlike koje ih čine jedinstvenima. U nastavku se ukratko opisuju razlike u **modelima podataka, modelima upita, agregaciji podataka i integraciji s aplikacijama** koje postoje između MySQL, Redis, MongoDB, Cassandra te Neo4j sustava.

### 4.6.1. Razlike u modelima podataka

Svaki od pregledanih sustava za upravljanje bazama podataka temelji se na različitom modelu podataka. MySQL se temelji na relacijskom modelu podataka, Redis koristi ključ/vrijednost model, MongoDB podatke sprema u obliku dokumenata, Cassandra se temelji na stupčanom modelu, a Neo4j koristi graf kao model podataka.

Relacijski model od korisnika MySQL baze podataka zahtjeva strogo definiranje sheme te omogućava normalizaciju podataka čime se smanjuje njihova redundancija. Za razliku od toga, modeli podataka kod ostalih sustava su po pitanju sheme znatno fleksibilniji, ali zato u većini slučajeva ne omogućavaju kreiranje veza između podataka (izuzetak ovdje je naravno Neo4j i njegov graf model podataka).

O modelu podataka također ovise konzistentnost podataka te skalabilnost ovih sustava. Naime, modeli podataka prisutni kod MySQL i Neo4j sustava omogućavaju im podršku za ACID princip konzistentnosti, ali ih ograničavaju po pitanju skalabilnosti pošto otežavaju distribuciju podataka na više servera. Za razliku od toga, Redis, MongoDB i Cassandra temeljeni su na modelima podataka koji su dizajnirani tako da omogućavaju horizontalno skaliranje, ali onemogućavaju, odnosno otežavaju korištenje ACID principa.

### 4.6.2. Razlike u modelima upita

O modelu upita primarno ovise mogućnosti baze podataka kada je u pitanju pretraživanje i ažuriranje podataka. S time na umu, svaki od opisanih sustava pruža različite mogućnosti pretraživanja i izvršavanja upita nad podatcima.

Model upita kod MySQL sustava temelji se na SQL jeziku, što korisnicima ovog sustava omogućava kreiranje kompleksnih i fleksibilnih upita nad podatcima pohranjenima u samoj bazi podataka. Pretraživanje podataka unutar MySQL baze podataka tipično se provodi korištenjem kombinacije SELECT naredbe i WHERE klauzule pomoću koje se provodi filtriranje podataka prije njihovog prikaza korisnicima. Osim toga, MySQL omogućava **kreiranje indeksa** za različite attribute koji se obično koriste u sklopu WHERE klauzule čime

se može ubrzati vrijeme izvođenja upita, posebno u situacijama kada se u upitima koji koriste JOIN operacije i vanjski ključevi kako bi se referencirali na različite tablice.

U usporedbi s MySQL sustavom, Redis, koji je u svojoj srži ključ/vrijednost baza podataka, podržava samo osnovni model upita. To znači da se model upita ovog sustava primarno temelji na dohvaćanju i ažuriranju vrijednosti iz baze uz pomoć jedinstvenog ključa kojemu pripada određena vrijednost. To jest, Redis kao takav ne omogućava napredne mogućnosti postavljanja upita temeljenih na konkretnim vrijednostima podataka koje su spremljene u samoj bazi kao što je to moguće kod MySQL i ostalih sustava.

MongoDB koristi upitni jezik koji se temelji na JavaScript programskom jeziku te koji korisnicima ovog sustava omogućava napredno pretraživanje dokumenata spremljenih unutar MongoDB baze podataka. To jest, pretraživanje dokumenata unutar MongoDB baze je moguće provoditi prema bilo kojem atributu koji se nalazi unutra samog dokumenta. Uz to, MongoDB također podržava i kreiranje različitih vrsta indeksa s ciljem optimiziranja upita, kao što su: indeksi temeljeni na specifičnim atributima dokumenata, *multikey* indeksi, tekst i geoprostorni indeksi itd.

U slučaju Cassandra, koja koristi CQL, postavljanje upita provodi se korištenjem SELECT naredbe, na, više-manje, identičan način kao u SQL jeziku. Glavna razlika između CQL i SQL jezika po pitanju korištenja SELECT naredbe jest ta što CQL ne podržava JOIN klauzulu te ne koristi koncept vanjskog ključa. Osim toga, Cassandra također podržava kreiranje indeksa nad pojedinim stupcima.

Neo4j koristi Cypher kao upitni jezik. Taj upitni jezik je također inspiriran SQL-om te je baziran na konceptu podudaranja uzoraka (eng. *pattern matching*). Prema tome, postavljanje upita u Neo4j bazi podataka općenito se sastoji se od definiranja uzoraka temeljem kojih se određuje oblik traženog rezultata te koji se potom upotrebljavaju prilikom pretraživanja različitih struktura unutar grafa.

### **4.6.3. Razlike u agregaciji podataka**

Razlike u agregaciji podataka između MySQL, Redis, MongoDB, Cassandra i Neo4j sustava očituju se u njihovoj mogućnosti da pomoću filtriranja, grupiranja, sortiranja i sličnih operacija kombiniraju i transformiraju podatke.

U MySQL sustavu se agregacija, odnosno grupiranje podataka po jednom ili više stupaca ili izraza obavlja korištenjem GROUP BY klauzule. Jednom kad su grupe kreirane, korisnicima MySQL sustava na raspolaganju je niz funkcija koje se mogu koristiti i primjenjivati nad formiranim grupama. Tu se prvenstveno misli na različite agregirajuće funkcije koje na

temelju skupa vrijednosti vraćaju jednu. Neke od agregirajućih funkcija koje MySQL podržava su: AVG(), COUNT(), MAX(), MIN(), SUM() itd.

Neki NoSQL sustavi, kao što je npr. Redis, nemaju neke osobite sposobnosti agregacije podataka. U takvim je situacijama posao agregacije podataka obično potrebno dodijeliti nekom vanjskom servisu ili pak ga je potrebno preseliti u programski kôd aplikacije.

Nadalje, MongoDB pruža nekoliko različitih mehanizama koji omogućavaju zajedničko grupiranje vrijednosti iz različitih dokumenata spremljenih unutar MongoDB baze podataka, a to su: **agregacijski cjevovodi**, **map-reduce funkcije** te **single purpose agregacijske metode**. Više informacija o ovim načinima agregacije podataka može se pronaći u službenoj dokumentaciji MongoDB sustava, koja je dostupna na <https://docs.mongodb.com/>. Uz to, unutar MongoDB sustava korisnicima su također dostupne različite agregirajuće funkcije koje se mogu izvršavati nad grupiranim podacima.

Cassandra, odnosno CQL upitni jezik kojeg Cassandra koristi također podražava GROUP BY naredbu. Tom naredbom moguće je unutar Cassandra baze podataka međusobno grupirati podatke koji dijele iste vrijednosti nad zadanim setom stupaca te potom upotrijebiti neku od dostupnih ili pak korisnički definiranih agregirajućih funkcija kako bi se dobio željeni rezultat.

Što se Neo4j sustava tiče, njegov Cypher upitni jezik isto tako podržava GROUP BY naredbu koja je u određenim aspektima usporediva s istoimenom naredbom dostupnom unutar SQL jezika. Ova naredba omogućava agregaciju podataka iz svih pod-grafova koji se nalaze unutar Neo4j baze podataka koji zadovoljavaju određeni uzorak. Neo4j također omogućava izvršavanje različitih agregirajućih operacija nad agregiranim podacima, kao što su npr.: *avg()*, *count()*, *min()*, *max()*, *sum()* i dr.

#### 4.6.4. Razlike u integraciji s aplikacijama

MySQL, Redis, MongoDB, Cassandra, te Neo4j, svaki od ovih sustava nudi različita programska sučelja (engl. *Application Programming Interface*, *API*) i drivere preko kojih se aplikacijama koje su napisane u različitim programskim jezicima omogućava povezivanje i upravljanje podacima koji se pohranjuju u ove sustave. Valja istaknuti kako su driveri za ove sustave dostupni u većini danas popularnih programskih jezika.

Nadalje, integraciju MySQL sustava s aplikacijama moguće je, osim kroz drivere, ostvariti i korištenjem različitih ORM alata koji se često koriste prilikom rješavanja problema neusklađenosti koji postoji između relacijskog modela podataka i objektno-orijentiranog programiranja, ali ti alati u svojoj pozadini također koriste maloprije spomenute drivere kako bi komunicirali s bazom podataka.

U ovom dijelu rada također valja istaknuti kako Neo4j sustav, osim pomoću drivera, omogućavaj upravljanje podacima i izvršavanje upita putem posebnog HTTP API-a koji aplikacijama omogućava komunikaciju s Neo4j bazom podataka putem HTTP protokola.

## 5. Aplikacije temeljene na NoSQL i relacijskim bazama podataka

Baza podataka sastavna je komponenta mnogih aplikacija. U aplikacijama one se koriste kao sredstvo za rješavanje različitih problema vezanih uz upravljanje podacima. Općenito, baze podataka omogućavaju aplikacijama efikasnu pohranu velike količine podataka te njihovo brzo pretraživanje i ažuriranje. Osim toga, one omogućavaju aplikacijama rješavanje različitih problema koji se vežu uz točnost, konzistentnost, dostupnost i sigurnost podataka itd.

NoSQL pokret rezultirao je pojavom velikog broja novih SUBP-a temeljenih na različitim modelima podataka koji su razvijeni s ciljem da riješe probleme tradicionalnih relacijskih sustava. Iz tog razloga većina NoSQL sustava općenito je specijalizirana za rješavanje određenih problema što ih čini prikladnima da se, u određenim situacijama, koriste zajedno s relacijskim sustavima kako bi se upravljanje podacima unutar aplikacija maksimalno prilagodilo karakteristikama samih podataka i konkretnom načinu njihova korištenja.

S time na umu, prvi dio ovog poglavlja bavi se općenito aplikacijama (što su aplikacije, koje vrste aplikacija postoje te kakva općenito može biti njihova arhitektura). Nakon toga definira se pojam aplikacijske baze podataka te se opisuju različiti pristupi njenom razvoju. Dio nakon toga u potpunosti je posvećen ideji korištenja više različitih baza podataka unutar iste aplikacije, a koja se u literaturi obično krije pod pojmom „*polygot persistence*“. Posljednji dio ovog poglavlja opisuje aplikacije temeljene na arhitekturi mikroservisa koja omogućava implementaciju same „*polygot persistence*“ arhitekture na prirodan način.

### 5.1. Definicija i vrste aplikacija

Aplikacija je **program** razvijen za korištenje na određenoj **hardverskoj i softverskoj platformi** s ciljem da se njenim korisnicima omogući obavljanje određenih zadataka. Napredovanjem tehnologije omogućeno je pokretanje aplikacija u različitim okruženjima tako da danas postoji jako veliki broj aplikacija koje su dostupne na raznim hardverskim platformama (kao što su računala, mobilni uređaji, poslužitelji (serveri) i dr.) i operacijskim sustavima.

Većina današnjih aplikacija koristi se na jednom od slijedeća tri okruženja, a to su: **desktop** i **mobilno okruženje** te **web**. S time na umu, većina aplikacija može se svrstati u jednu od slijedećih kategorija:

- Desktop (stolne) aplikacije
- Mobilne aplikacije
- Web aplikacije.

### **5.1.1.Desktop aplikacije**

Desktop aplikacije su aplikacije koje se nalaze na stolnim i prijenosnim računalima te se za njihovu izradu obično koriste objektno-orijentirani programski jezici kao što su Java i C#. Nadalje, ova vrsta aplikacija je, u usporedbi s web aplikacijama, puno manje ovisna o Internetu, tj. njihovo korištenje je u većini slučajeva omogućeno i u situacijama kada ne postoji veza računala prema Internetu.

Ovisno o tome za što su namijenjene, neke desktop aplikacije za svoj rad ne trebaju bazu podataka. No, također postoje i brojne desktop aplikacija koje tijekom rada za spremanje i upravljanje podacima koriste upravo baze podataka. Tipično za ovu situaciju je da se instalacija same baze podataka nalazi na posebnom serveru, ili pak negdje u oblaku, te da aplikacija pristupa bazi putem mreže. U ovom slučaju desktop aplikacija generalno ne može raditi ukoliko iz nekog razloga ne može pristupiti bazi podataka putem mreže. U nekim slučajevima ovaj problem je moguće riješiti tako da se desktop aplikaciji doda lokalna baza podataka ili pak kopija serverske baze koja će se periodično, ili pak u trenutku kada se omogući pristup računala mreži, automatski sinkronizirati sa serverskom bazom podataka.

Veliki nedostatak desktop aplikacija je njihovo održavanje, pošto je potrebno napraviti pojedinačno ažuriranje svake njene instalacije ukoliko dođe do neke promjene unutar aplikacije. Drugi nedostatak ove vrste aplikacija je da im korisnici mogu pristupiti samo preko računala na kojem su instalirane, dok se npr. web aplikacijama može pristupiti preko Interneta od bilo kuda. Nadalje, u usporedbi s web aplikacijama, prednosti stolnih aplikacija su sigurnost te veza si Internetom. Desktop aplikacije su sigurnije jer im korisnici ne pristupaju preko mreže te u tom smislu nisu javno dostupne što ih čini težim metama za napadanje. Uz to, kao što je već i spomenuto, većinu desktop aplikacija moguće je koristiti bez pristupa internetu, a što kod web aplikacija nije moguće.

### **5.1.2.Mobilne aplikacije**

S pojavom i razvojem različitih mobilnih uređaja, kao što su pametni telefoni, tablet računala i sl., te razvoj operacijskih sustava, poput Androida i iOS-a, koji su posebno namijenjenih ovoj vrsti uređaja, započeo je razvoj i rast popularnosti tzv. mobilnih aplikacija koje su posebno dizajnirane i prilagođene za korištenje na ovoj vrsti uređaja.



Kao što je spomenuto, ovdje se generalno radi o aplikacijama koje su dizajnirane i namijenjene za korištenje na mobilnim uređajima koji, u usporedbi s desktop računalima, imaju značajno drugačiji faktor oblika (engl. *form factor*) te ograničene hardverske resurse. Zbog toga mobilne aplikacije, u odnosu na desktop i punokrvne web aplikacije, generalno imaju manji broj funkcionalnosti te su jednostavnije za razvoj.

Općenito, postoje tri vrste mobilnih aplikacija, a to su:

- **nativne aplikacije** (engl. *native apps*) – aplikacije izrađene specifično za jednu platformu.
- **mobilne web aplikacije** – web aplikacije koje su posebno prilagođene za prikaz i korištenje na mobilnim uređajima.
- **hibridne mobilne aplikacije** – predstavljaju svojevrsnu kombinaciju nativnih i mobilnih web aplikacija; izrađuju se uz pomoć posebnih tehnologija te se na mobilnom uređaju obično pokreću unutar kontejnera (WebView-a).

Ovdje valja istaknuti kako mobilne web aplikacije i obične (standardne) web aplikacije u određenom smislu predstavljaju istu stvar. Naime, obje ove dvije vrste aplikacija žive na webu te se za njihovu izradu koriste iste tehnologije. To jest, mobilne i obične web aplikacije dijele iste prednosti i mane, samo što su mobilne web aplikacije, za razliku od običnih, dodatno prilagođene za prikaz i korištenje unutar mobilnih web preglednika

Nadalje, slično kao i kod desktop aplikacija, u slučaju kada bilo koja mobilna aplikacija treba koristiti bazu podataka, generalno se ona nalazi na posebnom serveru te mobilna aplikacija komunicira s njom putem mreže. No, i ovdje postoje situacije u kojima je moguće da pojedine mobilne aplikacije imaju vlastitu lokalnu bazu podataka ili pak lokalnu kopiju serverske baze koja će se na odgovarajući način periodično sinkronizirati sa serverskom bazom podataka. Ovim pristupom može se smanjiti ovisnost pojedinih mobilnih aplikacija o mreži te im omogućiti da ostanu funkcionalne i u situacijama kada je njihov pristup do serverske baze podataka ograničen.

### 5.1.3.Web aplikacije

Web aplikacije žive na webu, odnosno web serveru. To zapravo znači kako su, za razliku od desktop i nativnih mobilnih aplikacija koje su dostupne samo na uređaju na kojem su instalirane, web aplikacije tipično dostupne za korištenje putem web preglednika u bilo koje vrijeme, s bilo kojeg mjesta i bilo kojeg računala ili mobilnog uređaja koji imaju pristup Internetu.

Tipična web aplikacija sastoji se od **klijentskog dijela** (engl. *frontend*) i **dijela na strani poslužitelja** (engl. *backend*). Na klijentskoj strani obično se nalazi **klijentska aplikacija** koja je razvijena korištenjem web tehnologija kao što su: HTML, CSS i JavaScript. Ta klijentska

aplikacija živi unutar web preglednika te joj je svrha da omogući interakciju između korisnika i same web aplikacije. Nadalje, dio na strani poslužitelja tipične web aplikacije sastoji se od dvije glavne komponente, a to su: **web server** i **baza podataka**. Kao tehnologija za razvoj poslužiteljskog dijela web aplikacije danas se koriste različiti programski jezici kao što su: Python, Java, PHP, Ruby, C# i dr. Komunikacija između klijentskog i poslužiteljskog dijela obično se odvija putem **HTTP protokola** (engl. *Hypertext Transfer Protocol*, HTTP) na način da web server zaprima HTTP zahtjeve poslane od strane klijenta, obrađuje ih te generira odgovarajući HTTP odgovor. Prilikom obrade tih zahtjeva web aplikacija može izvršavati poslovnu logiku i primjenjivati različita poslovna pravila, čitati i ažurirati podatke iz baze, manipulirati datotekama na disku i dr. kako bi ostvarila svoju zadaću.

U usporedbi s desktop aplikacijama, održavanje i nadogradnja web aplikacija je relativno jednostavna pošto je svaku novu verziju web aplikacije moguće relativno brzo isporučiti na web server.

Na današnjem webu postoji jako puno različitih web aplikacija. Društvene mreže (engl. *social networks*), sustavi za upravljanje sadržajem (engl. *content management system*, CMS), sustavi za rezervaciju i Internet bankarstvo samo su neki od primjera web aplikacija s kojima je većina današnjih korisnika weba upoznata.

## 5.2. Arhitekture aplikacija

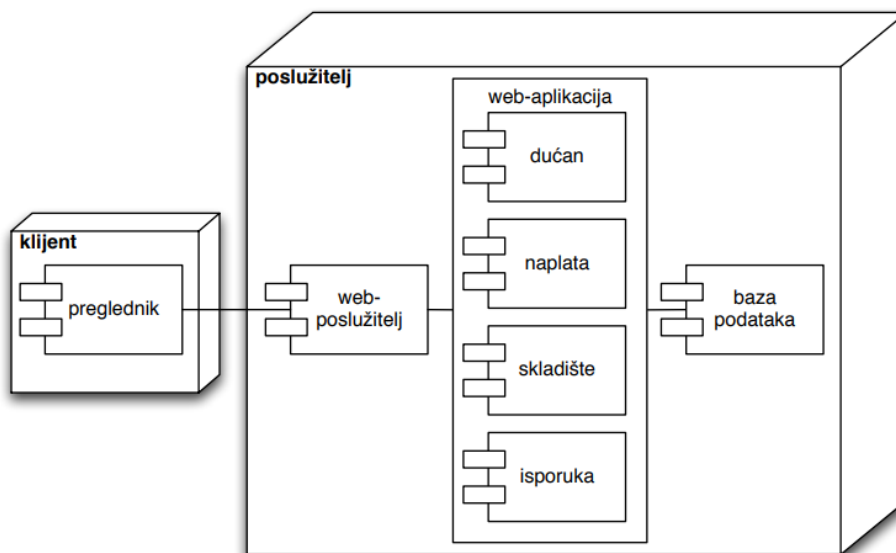
Svrha arhitekture aplikacije je definirati njezinu strukturu, utvrditi od kojih elemenata će se ona sastojati te kako će oni međusobno komunicirati. Sam proces definiranja arhitekture aplikacije je važan te mu je potrebno posvetiti dosta pažnje jer se njime definira temeljna struktura aplikacije koja, jednom kada bude implementirana, ne može biti lako zamijenjena. Ovdje valja istaknuti kako veliku ulogu prilikom odabira arhitekture također imaju i različiti nefunkcionalni zahtjevi aplikacije kao što su: skalabilnost, performanse, sigurnost, dostupnost, kasnije održavanje itd.

S time na umu, arhitekture aplikacija mogu se na najvišoj podijeliti **monolitne** i **distribuirane arhitekture**.

### 5.2.1. Monolitne arhitekture

Monolitna arhitektura predstavlja tradicionalan oblik arhitekture aplikacija prema kojoj se aplikacije razvijaju i grade kao **jedna cjelina**. Osnovna značajka monolitnih aplikacija (tzv. monolita) je ta da je kompletna poslovna logika sustava upakirana u jedinstvenu izvedbenu jedinicu (Musa, 2017). Veliki broj monolitnih aplikacija temelji se na arhitekturnim uzorcima koji se temelje na slojevima, kao što su **dvoslojna**, **troslojna** i ostale **višeslojne arhitekture**.

Naime, kod takvih i sličnih arhitektura aplikacije su općenito građene **bez modularnosti**, tj. svi njihovi dijelovi većinom su međusobno **čvrsto povezani** (engl. *tightly coupled*). Posljedica toga je da će svaka promjena u implementaciji nekog dijela programske logike izazvati potrebu ponovne izgradnje, pakiranja i instalacije cijele aplikacije (Musa, 2017).



Slika 14. Primjer tradicionalne monolitne aplikacije (Izvor: Žarko, Lovrek, Kušek, Pripužić, 2017)

Serverska (web) aplikacija (slika 14) koja je sastavni dio skoro svake tradicionalne arhitekture poslovnih web aplikacija predstavlja klasični primjer monolita. U ovom primjeru na slici 14 svi logički dijelovi serverske aplikacije (dućan, naplata, skladište, isporuka) se nalaze na istoj platformi, dijele iste resurse, dio su istog procesa te kao takvi predstavljaju **jedinstvenu logičku izvršnu cjelinu**.

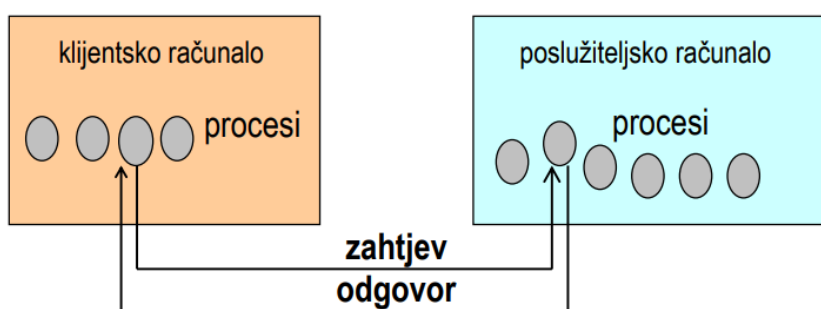
Ovaj pristup arhitekturi, kao i svaki drugi, ima svoje prednosti i nedostatke. Naime, monolitnu arhitekturu je prije svega jednostavno shvatiti i implementirati. Osim toga, ova arhitektura omogućava jednostavno testiranje, mijenjanje i upravljanje razvojem aplikacije, do određene veličine. Zatim, monolitne aplikacije je relativno jednostavno isporučiti i skalirati kroz pokretanje njihovih dodatnih instanci. Ovdje valja istaknuti kako je ovaj pristup skaliranju problematičan jer se na ovaj način umjesto kritičnih poslovnih funkcija skalira cijela aplikacija te se onemogućava kvalitetnije korištenje sistemskih resursa (Musa, 2017). Veliki nedostatak monolitne arhitekture jest njihova vezanost za programski jezik i programske okvire. Naime, ovaj pristup arhitekturi zahtijeva da se cijela aplikacija razvija u jednoj tehnologiji čime se onemogućava korištenje novih i drugačijih pristupa za rješavanje pojedinih problema. Nadalje, kako monolitna aplikacija raste, njezin programski kôd generalno postaje sve manje čitljiv i

teže ga je shvatiti, a samo jedna promjena u kôdu može uzrokovati lanac promjena u čitavoj aplikaciji. (Žarko, Lovrek, Kušek, Pripužić, 2017).

### 5.2.2. Distribuirane arhitekture

Distribuirane arhitekture omogućavaju razvoj **modularnih** i **distribuiranih** aplikacija. Distribuiranost aplikacije se kod distribuiranih arhitektura općenito postiže na način da se funkcionalnosti i poslovna logika aplikacije distribuiraju između većeg broja računala (servera) u mreži. Jedan od načina koji omogućava distribuciju aplikacija na ovaj način jest da se aplikacija razvija u obliku **modularnih i distribuiranih softverskih komponenti** koje imaju jasno definiranu funkciju te su međusobno neovisne jedna o drugoj.

Za dizajniranje i razvoj tih softverskih komponenata obično se koriste različite distribuirane tehnologije kao što su: **programiranje priključnica** (engl. *socket programming*), **poziv udaljenje procedure** (engl. *Remote Procedure Call, RPC*), **Common Object Request Broker Architecture** (CORBA), **Enterprise Java Beans** (EJB), **web usluge** itd.



Slika 15. Klijent-poslužitelj model (Izvor: Žarko, Lovrek, Kušek, Pripužić, 2017)

Najpoznatiji i vjerojatno najjednostavniji arhitektonski uzorak koji se koristi prilikom implementacije distribuirane arhitekture temelji se na poznatom modelu **klijent-poslužitelj** (slika 15). Kod ovog modela klijenti su ti koji zahtijevaju usluge od poslužitelja na način da šalju zahtjeve poslužitelju i čekaju odgovor, a zadaća poslužitelja je da nudi usluge te da prima i obrađuje dolazne zahtjeve te šalje odgovor klijentima. Osim klijent-poslužitelj modela, distribuirane arhitekture obuhvaćaju i veliki broj drugih arhitektonskih uzoraka od kojih svakako valja istaknuti **uslužno-orijentiranu arhitekturu** (engl. *Service Oriented Architecture, SOA*) i **arhitekturu mikroservisa** kod kojih **usluge** predstavljaju primarnu arhitekturnu komponentu za implementaciju različitih funkcionalnosti aplikacije ili sustava. Neki od ostalih arhitektonskih stilova koji omogućavaju implementaciju distribuirane arhitekture su: *peer-to-peer* arhitektura, događajima pokretana arhitektura (engl. *event-driven architecture*) koja se može temeljiti na topologiji brokera ili medijatora itd.

Distribuirane arhitekture, u usporedbi s monolitnim i višeslojnim arhitekturama, pružaju značajne prednosti kao što su: bolja skalabilnost, veća razdvojenost između pojedinih dijelova aplikacije te bolja kontrola nad razvojem, testiranjem i isporukom aplikacije. Komponente unutar distribuiranih aplikacija općenito omogućavaju bolje upravljanje promjenama i lakše održavanje aplikacije čime se omogućava razvoj robusnijih aplikacija koje imaju bolje performanse u odnosu na monolitne aplikacije. No, navedene prednosti distribuiranih arhitektura dolaze pod cijenu značajnog povećanja složenosti samog sustava, a koja može biti izvor različitih kompleksnih problema koji se mogu pojaviti prilikom razvoja i korištenja takvog sustava. (Richards, 2016).

### 5.3. Definicija i pristupi razvoju aplikacijske baze podataka

Martin Fowler (n.d.) definira aplikacijsku bazu podataka kao bazu podataka **kojom upravlja i kojoj pristupa samo jedna aplikacija**. Na taj način omogućava se da ta baza podataka puno lakše zadovolji potrebe te aplikacije, nego u situaciji kada jedna baza podataka ima ulogu tzv. integracijske baze<sup>7</sup> kojoj istovremeno pristupa više aplikacija. Korištenjem aplikacijske baze podataka, odnosno sprječavanje toga da više različitih aplikacija koristi istu bazu podataka omogućava se puno lakše provođenje promjena u dizajnu same baze podataka, čak i nakon što je baza podataka stavljena u produkciju.

Traženje odgovora na pitanje kako pristupiti razvoju aplikacijske baze podataka te koju od različitih tehnologija koje su danas prisutne u svijetu baza podataka koristiti nije uvijek jednostavno. Iz tog razloga potrebno je, prije donošenje te odluke, upoznati se s mogućnostima i značajkama što većeg broja različitih rješenja koja su danas prisutna u svijetu baza podataka. Još važnije, treba se upoznati s potrebama same aplikacije te identificirati i upoznati se s strukturom podataka s kojima će aplikacija raditi. S time na umu, ponekad je prikladno i poželjno prilikom razvoja aplikacije koristiti isključivo neku od NoSQL ili relacijskih baza podataka, a ponekad oboje.

#### 5.3.1. Relacijski pristup

S pojavom NoSQL-a relacijske baze podataka **nisu** postale zastarjele. Relacijske baze podataka danas još uvijek predstavljaju najpopularniju tehnologiju koja se koristi za upravljanje i pohranu podataka u aplikacijama. To jest, danas postoji veliki broj situacija kod kojih relacijske baze podataka predstavljaju logičan, a ponekad čak i jedini izbor kako bi se zadovoljile potrebe same aplikacije.

---

<sup>7</sup> Integracijska baza podataka – baza podataka u kojoj s spremaju podatci iz više različitih aplikacija te se na taj način vrši integracija njihovih podataka.

Naime, brojne poslovne aplikacije jednostavno zahtijevaju da podatci budu dobro strukturirani te da imaju **dobro definiranu, formalnu shemu** (npr. podatci o zaposlenicima, dionicama, policama osiguranja itd.). Uz to, brojne aplikacije također zahtijevaju od baze podataka **dobru podršku za ACID transakcije** kako bi se osigurao integritet podataka (npr. različite financijske aplikacije). Osim toga, relacijske baze podataka koriste SQL jezik te pružaju dobru podršku za postavljanje kompleksnih upita, agregaciju podataka, okidače, pohranjene procedure i još puno toga što je potrebno da se zadovolje potrebe velikog broja različitih aplikacija.

Generalno, relacijske baze podataka predstavljaju dobar izbor za različite aplikacije koje će imati koristi od predefinirane sheme te koje nisu usmjerene na web, tj. ne očekuju eksponencijalni rast broja korisnika i količine podataka koji bi ih mogli dovesti do problema s skaliranjem.

### 5.3.2.NoSQL pristup

**Mogućnost horizontalnog skaliranja te jednostavnost korištenja** vjerojatno su dva glavna razloga prilikom odabira neke NoSQL baze podataka kao aplikacijske baze podataka.

NoSQL pristup razvoju aplikacijske baze podataka općenito je pogodan kod aplikacija koje trebaju upravljati velikim količinama različitih strukturiranih, polu-strukturiranih i nestrukturiranih podataka jer u takvim situacijama NoSQL baze generalno pružaju veću fleksibilnost modela podataka, bolju horizontalnu skalabilnost te generalno bolje performanse u odnosu na tradicionalne relacijske sustave.

Nadalje, nepostojanje sheme kod NoSQL baza podataka čini ovaj pristup razvoju aplikacijske baze podataka puno prikladnijim za korištenje u kombinaciji s današnjim modernim metodama razvoja aplikacija koje karakteriziraju agilni pristup razvoju, brze iteracije te česte promjene kôda.

### 5.3.3.Hibridni pristup

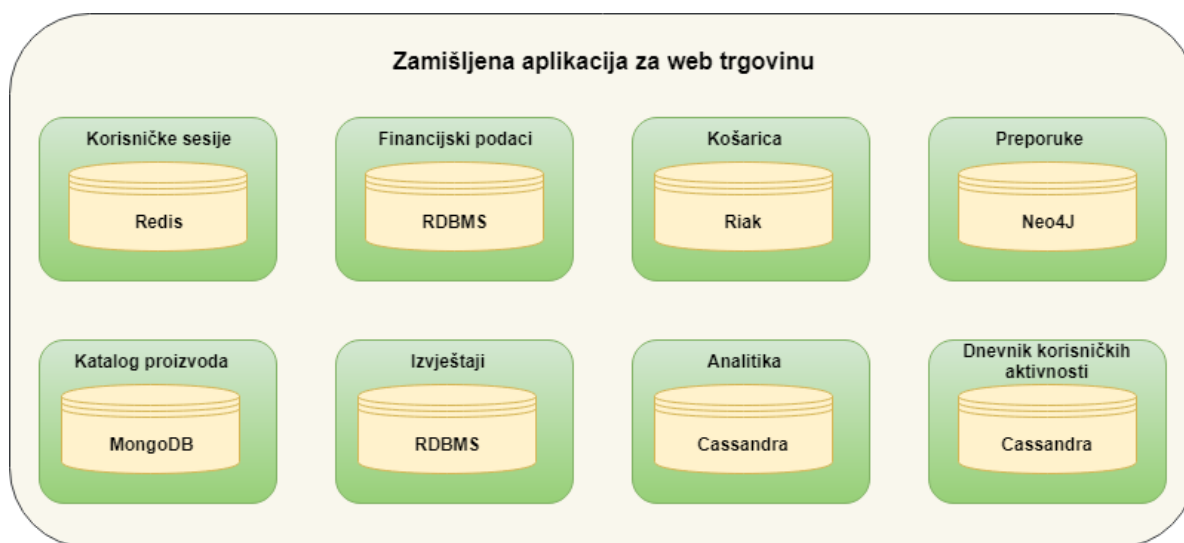
U praksi se često dešavaju situacije kod kojih NoSQL i relacijske baze podataka mogu međusobno nadopunjavati jedna drugu.

Naime, NoSQL baze podataka općenito predstavljaju specijalizirana rješenja koja se mogu nositi s različitim izazovima koji se vežu uz velike količine podataka za čiji opseg, kompleksnost i brzinu dolaska relacijske baze podataka nisu dizajnirane. S druge strane, relacijske baze podataka dokazale su se kao efikasan alat u sustavima koji zahtijevaju konzistentnost podataka i podršku za transakcije. Iz tog razloga, NoSQL i relacijske baze podataka mogu, u određenim situacijama, predstavljati dobru kombinaciju.

Nadalje, s razvojem i rastom popularnosti NoSQL baza podataka proizvođači mnogih relacijskih sustava omogućili su hibridni pristup razvoju aplikacijske baze podataka tako što su počeli u svoje proizvode ugrađivati različite značajke NoSQL baza podataka, kao što je npr. podrška za JSON dokumente, kako bi zadovoljili potrebe što većeg broja korisnika. Na taj način su proizvođači relacijskih sustava počeli omogućavati korištenje relacijskih baza podataka uz istovremeni pristup nekima od novih načina rješavanja problema koje je omogućio razvoj NoSQL-a.

## 5.4. Polyglot persistence

**Polyglot persistence** je pojam koji opisuje korištenje različitih tehnologija pohrane podataka unutar jedne aplikacije ili sustava kako bi se njihovo upravljanje podacima što bolje prilagodilo karakteristikama i načinu korištenja tih podataka. Drugim riječima, *polyglot persistence* predstavlja ideju korištenja različitih baza podataka za rješavanje različitih problema i potreba vezanih uz upravljanje podacima unutar iste aplikacije kako bi se postigao željeni cilj.



Slika 16. Primjer poslovne aplikacije po *polyglot persistenceu* (Prema: Fowler, 2011)

Slika 16 prikazuje *polyglot persistence* pristup korištenju više različitih tehnologija za pohranu podataka na primjeru zamišljene aplikacije za web trgovinu. Tom slikom nastoje se prikazati neki od mogućih odabira različitih tehnologija za pohranu podataka koji se temelje na načinu na koji će se pojedini podaci koristiti u samoj aplikaciji. Na primjer, za rad s financijskim podacima i kreiranje izvještaja vjerojatno su najprikladniji relacijski sustavi, jer podržavaju transakcije te omogućavaju laganu integraciju s većinom postojećih sustava za izvještavanje.

Nadalje, ključ/vrijednost sustavi, poput Redisa i Riaka, prikladni su za rad s podacima kod kojih trajnost nije toliko bitna (kao što su npr. podatci o sesijama korisnika), ali je bitno brzo čitanje i pisanje. Za spremanje podataka koji predstavljaju prirodne agregate (kao što su npr. podatci o proizvodima) vjerojatno je najprikladnije koristiti MongoDB ili neki drugi sustav temeljen na dokument-bazama podataka. Cassandra sustav omogućava analizu podataka unutar velikog grozda računala te istovremeno zapisivanje velike količine podataka ne veliki broj čvorova što ovaj sustav čini prikladnim za analitiku i pohranu podataka korisničkih aktivnosti. Neo4j sustav je poznat kao dobar alat za generiranje preporuka. Ovo su samo neki od primjera koji opisuju kako različite tehnologije pohrane podataka mogu istovremeno biti dio zajedničke arhitekture različitih aplikacija.

### 5.4.1. Pristupi integraciji NoSQL i relacijskih baza podataka

Korištenje različitih baza podataka koje su opisane u onome radu, unutar iste aplikacije, vodeći se *polyglot persistence* principom, izazvati će probleme i izazove zbog neusklađenosti koja nastaje kao posljedica značajnih razlika (u modelima podataka, načinu izvršavanja upita itd.) koje postoje između implementacija tih baza podataka.

S time na umu, u nastavku rada opisuju se četiri različita arhitekturna pristupa za koje Engelschnall (2013) tijekom svoje prezentacije na konferenciji *NoSQL Matters 2013* navodi da omogućavaju integraciju NoSQL i relacijskih baza podataka te implementaciju „*polyglot persistence*“ arhitekture unutar iste aplikacije. To su:

- pristup temeljen na **višestrukim linijama** (engl. *multiple lines*),
- pristup temeljen na korištenju ***polyglot mapper*** alata,
- pristup temeljen na **ugniježđenoj bazi podataka** (engl. *nested database*) te
- pristup temeljen na tzv. **svemogućoj bazi podataka** (engl. *omnipotent database*).

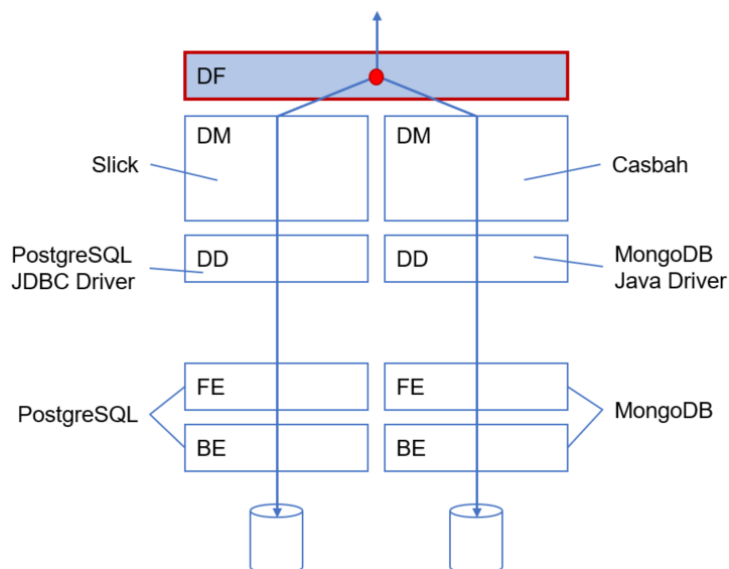
#### 5.4.1.1. Višestruke linije

Kod ovog pristupa se integracija između različitih baza podataka unutar iste aplikacije postiže na način da aplikacija prema svakoj bazi podataka kreira posebnu **perzistentnu liniju** preko koje podatci putuju između aplikacije i odgovarajuće baze podataka.

Prema slici 17, za integraciju podataka kod ovog pristupa može biti zadužena posebna komponenta aplikacije koja predstavlja tzv. **podatkovnu fasadu** (engl. *data facade*, DF) koja se nalazi između ostalih dijelova aplikacije i različitih baza podataka. U ovom slučaju, zadaća te komponente je da raspodijeli rad s podacima u odgovarajuće perzistentne linije od kojih svaka vodi prema odgovarajućoj bazi podataka (komponente označene oznakama FE i BE na slici 17 predstavljaju *frontend* i *backend* odgovarajućeg SUBP-a, respektivno). Nadalje, same perzistentne linije su međusobno neovisne te se mogu sastojati od različitih dijelova koji



omogućavaju komunikaciju između odgovarajuće baza podataka i aplikacije. Na primjer, prema slici 17, to mogu biti razni alati za mapiranje podataka (engl. *data mapper*, DM), driveri koji omogućavaju komunikaciju s bazom podataka (engl. *database drivers*, DD) i sl.



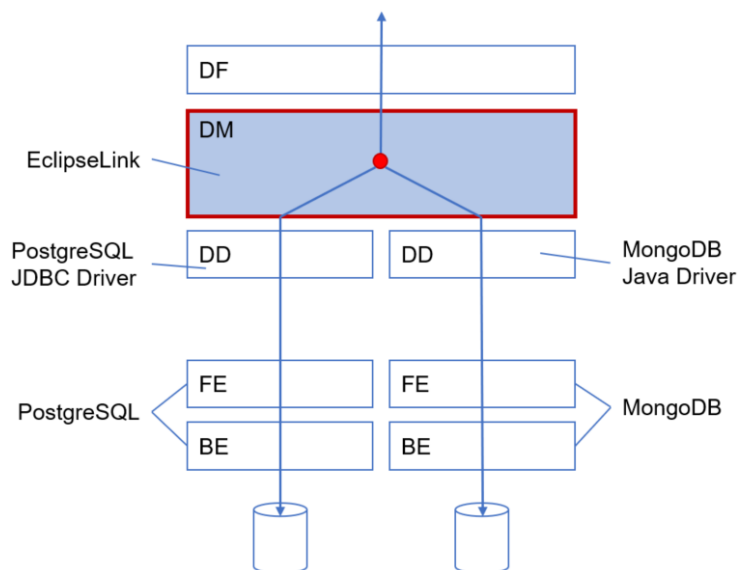
Slika 17. Primjer integracije različitih SUBP-ova korištenjem višestrukih linija (Prema: Engelschnall, 2013)

Glavne prednosti ovog pristupa su što je relativno jednostavan za implementirati te ga je moguće koristiti sa svim vrstama baza podataka. No, ovaj pristup ima i svoje nedostatke jer zahtjeva da se podatci vezani uz neku određenu domenu unaprijed međusobno grupiraju jer je upite između različitih perzistentnih linija dosta teško implementirati.

#### 5.4.1.2. Polyglot Mapper

**Polyglot Mapper** predstavlja posebnu skupinu alata koji su slični klasičnim ORM alatima u smislu da omogućavaju pretvaranje podataka između objekata koji se nalaze u aplikaciji i entiteta koji su pohranjeni u bazi podataka. Glavna razlika između njih je što ORM alati isključivo podržavaju rad s relacijskim bazama podataka, dok *polyglot mapper* alati za mapiranje podataka omogućavaju istovremeno korištenje različitih vrsta baza podataka.

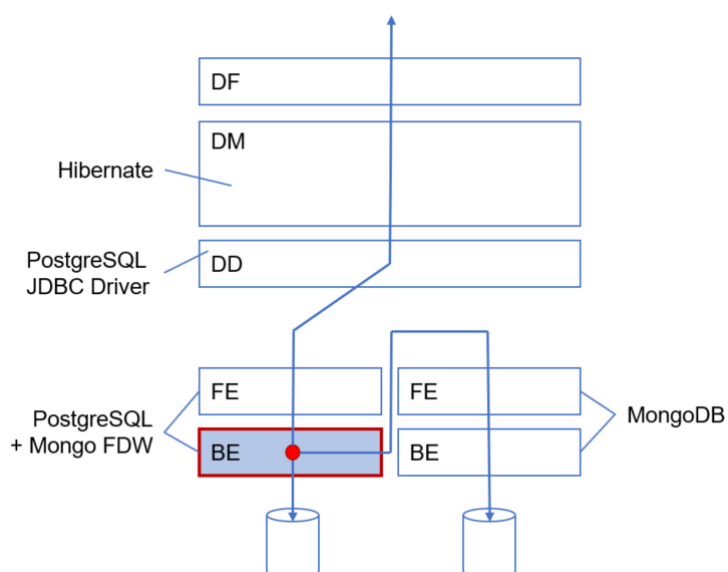
S time na umu, slika 18 prikazuje kako je moguće pomoću *polyglot mapper* alata ostvariti integraciju različitih SUBP-ova unutar iste aplikacije. Kod ovog pristupa zadaća integracije podataka više nije aplikaciji, već isključivo na posebnom alatu za mapiranje podataka koji se nalazi između aplikacije i perzistentnih linija podataka koje vode prema odgovarajućim NoSQL i relacijskim SUBP-ovima. Na ovaj način se aplikacija više ne mora brinuti o tome kako i u koje baze podataka se pojedini podatci spremaju, već se o tome brine isključivo alat za mapiranje podataka.



Slika 18. Pristup integraciji SUBP-ova korištenjem *Polyglot Mapper* alata (Prema: Engelschnall, 2013)

U usporedbi s pristupom integraciji koja se isključivo temelji na višestrukim perzistentnim linijama, korištenje *polyglot mapper* alata omogućava postojanje jedinstvenog modela podataka temeljenog na objektima na razini cijele aplikacije. Uz to, ovaj pristup omogućava zadavanje upita koji se istovremeno protežu kroz više različitih perzistentnih linija podataka, ali nažalost, trenutno samo nekolicina alata, kao što su npr. *EclipseLink* i *DataNucleus*, omogućava ovaj pristup.

#### 5.4.1.3. Ugniježdjena baza podataka



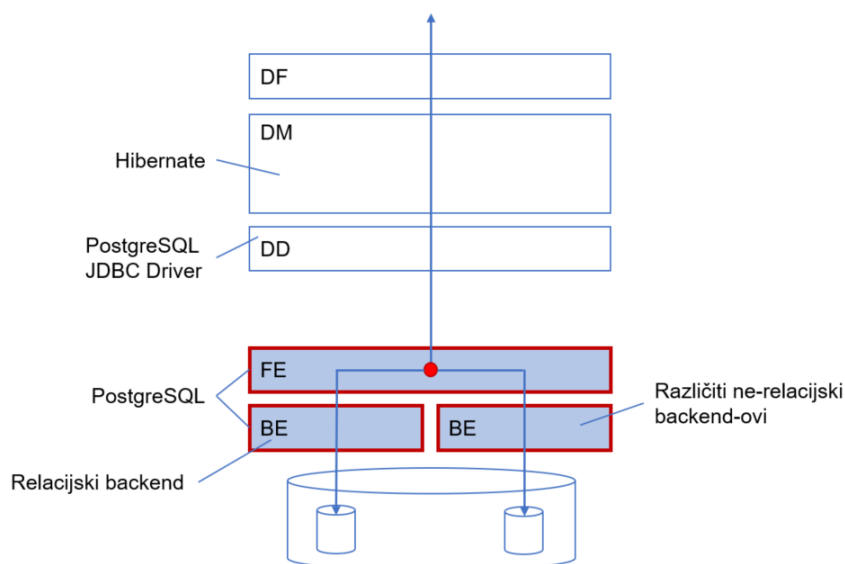
Slika 19. Primjer integracije NoSQL i relacijskih SUBP-ova pomoću ugniježdjene baze podataka (Prema: Engelschnall, 2013)

Slika 19 prikazuje integraciju NoSQL i relacijskog sustava za upravljanje bazama podataka unutar iste aplikacije na primjeru koji se temelji na korištenju tzv. **ugniježdene baze podataka**. Ovi se zapravo radi o sustavu za upravljanje podacima koji na svom *backendu* omogućava mapiranje drugih, relacijskih i ne-relacijskih, baza podataka u svoj vlastiti model podataka. Aplikacija se u ovom slučaju povezuje s glavnim sustavom za upravljanje podacima te pritom ima pristup i podacima koji se nalaze u sekundarnim sustavima koji su na ovaj način mapirani i povezani s primarnim SUBP-om.

Glavne prednosti ovog pristupa su što je na ovaj način *polyglot persistence* u potpunosti sakriven od aplikacije te je moguće izvršavati upite koji se istovremeno protežu između više različitih baza podataka. Glavni nedostaci ovog pristupa su pak što je podržan od strane malog broja sustava te što pristup aplikacije do određenih značajki ovih baza podataka koje su povezane s primarnim sustavom može biti ograničen.

#### 5.4.1.4. Svemoguća baza podataka

Engelschnall (2013) opisuje „svemoguću bazu podataka“ kao bazu podataka koja u svojoj pozadini (*backendu*) ima više različitih mehanizama pohranjivanja podataka (engl. *storage engines*), a koji mogu imati različite relacijske i NoSQL značajke. Kao što je prikazano primjerom na slici 20, ovdje se zapravo radi o bazi podataka koja u svojoj pozadini podržava **više različitih modela podataka**. U slučaju PostgreSQL sustava to je moguće postići uz pomoć različitih ekstenzija.



Slika 20. Primjer integracije NoSQL i relacijskog pristupa korištenjem „svemoguće baze podataka“ (Prema: Engelschnall, 2013)

Slično kao i kod ugniježđenih baza podataka, u ovom slučaju je *polyglot persistence* također u potpunosti sakriven od aplikacije. Uz to, ovaj pristup *polyglot persistence* arhitekturi omogućava jednostavnije održavanje baze podataka, a glavni nedostatak ovog pristupa je što ga podržava mali broj sustava čije NoSQL mogućnosti su većinom, u odnosu na punokrvne NoSQL sustave, dosta ograničene.

#### 5.4.2. Prednosti i mane „*Polyglot persistence*“ arhitekture

Pravilnim korištenjem različitih tehnologija za pohranu podataka unutar iste aplikacije omogućava se da ta aplikacija iskoristi prednosti različitih specijaliziranih baza podataka koje koriste visoko optimizirane formate pohrane podataka te pružaju odlične performanse izvršavanja upita, i većinom su otvorenog kôda.

No, korištenje različitih tehnologija za pohranu podataka unutar iste aplikacije dolazi pod cijenu povećanje sveukupne kompleksnosti njene arhitekture. Uz to, između različitih baza podataka koje implementiraju te tehnologije postoji neusklađenost pošto se temelje na različitim modelima podataka te koriste različite podatkovne strukture i upitne jezike. Nadalje, upravljanje različitim bazama podataka unutar iste aplikacije također je izazovno jer svaka od njih može zahtijevati drugačije pristupe po pitanju održavanja i oporavka u slučaju problema. (Engelschnall, 2013).

S time na umu, može se zaključiti kako ideja *polyglot persistence* predstavlja zanimljiv koncept koji ima svoje prednosti, ali cijena njegove implementacije može biti dosta velika. Iz tog razloga generalno se preporuča koristiti ovaj pristup samo u situacijama gdje to ima smisla te pokušati pritom upotrebljavati što je moguće manje različitih sustava za upravljanje bazama podataka kako bi troškovi implementacije i održavanja sustava bili što manji.

### 5.5. Aplikacije temeljene na mikroservisima

Fowler i Lewis (2014) opisuju mikroservisnu arhitekturu kao „*princip razvoja aplikacija u obliku malih, izdvojenih servisa, pri čemu svaki servis ima svoj proces i ostvaruje komunikaciju putem jednostavnih mehanizama kao što je HTTP API*“.

S time na umu, u kontekstu teme ovog rada, koja se odnosi na aplikacije temeljene na NoSQL i relacijskim bazama podataka, mikroservisna arhitektura vjerojatno predstavlja jedan od **najprirodnijih** načina za njihovu implementaciju. Naime, ova arhitektura omogućava da se implementacija velike i kompleksne aplikacije ostvari kroz **razvoj manjih, nezavisnih servisnih aplikacija** (mini aplikacija) koje kao takve mogu biti napisane u različitim programskim jezicima te mogu za pohranu podataka koristiti različite tehnologije koje najbolje zadovoljavaju njihove potrebe.

### 5.5.1. Karakteristike mikoservisne arhitekture

Za mikroservisnu arhitekturu je karakteristično da se implementacija aplikacija temelji na razvoju **servisa** koji imaju **dobro definirane granice unutar jedne poslovne cjeline**. Kao takvi ti servisi zapravo predstavljaju **komponente** mikroservisnih aplikacija kod kojih se prije svega ističu **neovisnost** i **autonomija**. Naime, u sustavima koji se temelje na mikroservisima trebalo bi biti moguće pokrenuti, zaustaviti i zamijeniti bilo koji servis, u bilo kojem trenutku, bez utjecaja na ostale servise. (Fowler i Lewis, 2014). (Hofmann, Schnabel, i Stanley, 2016).

Nadalje, mikroservisna arhitektura omogućava **decentralizirani** pristup odabiru tehnologija i upravljanju podacima, a što zapravo znači da razvoj mikroservisnih aplikacija nije ograničen na jednu tehnološku platformu. Razlog tome je što ova arhitektura omogućava da se izbor tehnologija za implementaciju svakog mikroservisa prilagodi njihovim potrebama. Ovaj pristup se posebno ističe kod upravljanja podacima unutar mikroservisne arhitekture u kojoj je tipično da svaki mikroservis ima svoje spremište podataka koje ne dijeli s drugim mikroservisima, a rezultat toga je da je za svakog od njih moguće individualno odabrati koju vrstu baze podataka će koristiti.

Važan aspekt mikroservisne arhitekture predstavlja i način na koji mikroservisi međusobno komuniciraju. Naime, kod ove arhitekture komunikacija se najčešće odvija uz pomoć jednostavnih mehanizama koji tu komunikaciju omogućavaju neovisno o tome u kojem programskom jeziku su pojedini mikroservisi sustava razvijeni. S time na umu, ova komunikacija se u mikroservisnoj arhitekturi najčešće ostvaruje korištenjem različitih **protokola** (HTTP, REST, TCP itd.) ili pak kroz korištenje jednostavnog **poručivanja** (engl. *messaging*) kako bi se ostvarila asinkrona komunikacija između mikroservisa.

Fowler i Lewis (2014) također navode kako je jedna od karakteristika mikroservisne arhitekture velika prisutnost **automatizacije infrastrukture** koja okružuje mikroservise. Naime, uz pomoć različitih tehnika automatizacije infrastrukture moguće je, između ostaloga, automatizirati testiranje softvera te značajno smanjiti operativnu složenost izgradnje, implementacije i upravljanja mikroservisima u različitim okolinama.

Nadalje, aplikacije s pravilno implementiranom mikroservisnom arhitekturom **otpornije su na kvarove** te imaju **mogućnost brzog oporavka** u slučaju poteškoća i kvarova. Mikroservisne aplikacije dizajnirane su tako da budu otporne na kvarove, jer se svaka njihova komponenta (mikroservis) može u bilo kojem trenutku pokvariti i postati nedostupna. Iz tog razloga je karakteristično i vrlo važno za aplikacije temeljene na mikroservisnoj arhitekturi da nadgledaju rad svakog pojedinog mikroservisa, da pravovremeno detektiraju greške te da omogućе automatski oporavak, ukoliko je moguć (Fowler i Lewis, 2014).

## 5.5.2. Upravljanje podacima

Dobro upravljanje podacima predstavlja jedan od najvažnijih ciljeva kod razvoja aplikacija temeljenih na mikroservisima.

Hofmann, Schnabel i Stanley (2016) navode kako bi svaki mikroservis trebao biti odvojen od ostalih mikroservisa te bi trebao imati svoje spremište podataka. Kao razloge zašto jedan mikroservis ne smije izravno pristupati podacima ostalih mikroservisa oni navode slijedeće:

- dijeljenjem istog spremišta podataka mikroservisi postaju **čvrsto povezani**; promjena strukture podataka u bazi za potrebe jednog mikroservisa može uzrokovati probleme kod drugoga; zbog čvrste povezanosti isporuka novih verzija mikroservisa mora biti dobro koordinirana što nije poželjno;
- svaki mikroservis treba koristiti vrstu baze podataka koja najbolje odgovara njegovim potrebama; kompromisi prilikom odabira baze podataka mikroservisa ne bi smijali postojati;
- s aspekta performansi bolje je da svaki mikroservis ima svoju bazu podataka jer na taj način skaliranje postaje puno jednostavnije;

Ovo razdvajanje podataka između mikroservisa moguće je postići na različite načine. Konkretno, Hofmann, Schnabel i Stanley (2016) navode slijedeća tri načina na koje je moguće postići razdvajanje podataka između mikroservisa u relacijskim bazama podataka:

- **Shema po mikroservisu**

U ovom slučaju više različitih mikroservisa može koristiti istu bazu podataka, ali unutar te baze podataka svaki mikroservis ima svoju vlastitu shemu.

- **Baza podataka po mikroservisu**

Kod ovog pristupa razdvajanje podatka postiže se tako da svaki mikroservis ima vlastitu bazu podataka unutar zajedničkog sustava za upravljanje bazama podataka.

- **Sustav za upravljanje bazama podataka po mikroservisu**

Na ovaj način postiže se najviša razina odvojenosti podataka među mikroservisima. S aspekta performansi ovo je najbolji pristup.

Nadalje, u mikroservisnoj arhitekturi važno je obratiti pažnju na to kako će se implementirati izvođenje poslovnih transakcija koje se protežu između više različitih mikroservisa. U tom slučaju potrebna je metoda koja će osigurati konzistentnost podataka koji su obuhvaćeni tom transakcijom. Kao najbolji način za upravljanje transakcijama koje se proširuju na više mikroservisa Hofmann, Schnabel i Stanley (2016) navode pristup temeljen

na **arhitekturi vođenoj događajima** (engl. *event-driven architecture*). Oni navode kako je ideja ovog pristupa slijedeća: nakon što jedan mikroservis ažurira svoje podatke on će objaviti događaj, a drugi servis koji je pretplaćen na taj događaj će potom, nakon što primi događaj, ažurirati svoje podatke. Korištenjem takvog **objavi-pretplati** modela komunikacije između mikroservisa sprječava se njihova čvrsta povezanost te se omogućava održavanje konzistentnosti podataka kada se poslovne transakcije protežu kroz više mikroservisa, ali također valja napomenuti kako implementacija takvog modela može biti dosta kompleksna.

### 5.5.3. Prednosti i mane mikroservisne arhitekture

Mikroservisna arhitektura omogućava razvojnim timovima da **relativno neovisno** jedni o drugima razvijaju i isporučuju aplikacije. Nadalje, velika prednost ove arhitekture jest što omogućava da mikroservisi, koji, u ovom slučaju, predstavljaju temeljnu komponentu za izradu aplikacije, mogu biti implementirani sa **različitim tehnologijama** i pisani u različitim jezicima, od strane **različitih timova** unutar organizacije. Ovdje kao prednosti valja istaknuti i mogućnost korištenja **različitih baza podataka**. Nadalje, kod sustava temeljenih na mikroservisima centralizirano upravljanje je minimalno, a svaki mikroservis se može neovisno isporučivati i automatski stavljati na servere. Uz to, mikroservisna arhitektura omogućava često izdavanje novih i ažuriranje postojećih servisa bez da se pri tome ugrožava stabilnost ostatka sustava. (Pečanac, n.d.).

S druge strane, razvoj sustava i aplikacija temeljenih na mikroservisnoj arhitekturi može biti, u usporedbi s npr. monolitnom arhitekturom, **dosta kompleksniji**. Nadalje, sama **međuservisna komunikacija** također može biti poprilično složena. Zbog složene interakcije njihovo testiranje je većinom **otežano**. Uz to, prisutnost različitih baza podataka te upravljanje transakcijama u sustavima koji se temelje na mikroservisnoj arhitekturi mogu predstavljati izazov. Iz ovih razloga u sustavima koji se temelje na ovoj arhitekturi postoji potreba za robusnom **upravljanjem greškama** i **automatskim oporavkom** kao i za **sofisticiranim nadgledanjem rada**. (Pečanac, n.d.).

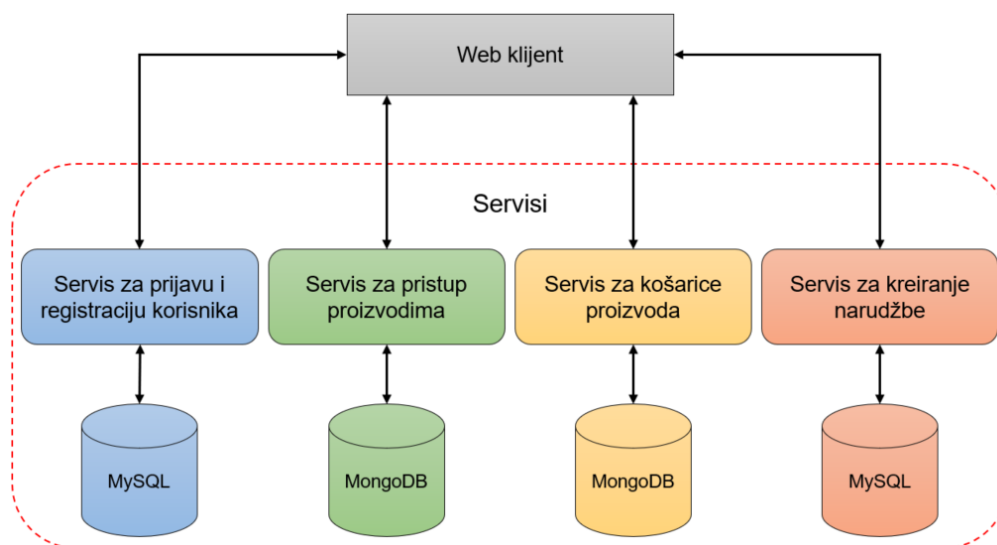
## 6. Primjer razvoja web aplikacije temeljene na NoSQL i relacijskim bazama podataka

U dosadašnjem dijelu rada imali smo priliku upoznati se s relacijskim i NoSQL bazama podataka. Vidjeli smo različite pristupe integraciji ovih tehnologija unutar iste aplikacije te koje su prednosti i izazovi istovremenog korištenja različitih baza podataka u istoj aplikaciji. S time na umu, ovo poglavlje posvećeno je prikazu implementacije web aplikacije koja će se temeljiti na NoSQL i relacijskim bazama podataka, a čija arhitektura je inspirirana mikroservisima.

U nastavku ovog poglavlja najprije se opisuje sama web aplikacija. Njen opis obuhvaća opis njezinih funkcionalnosti i arhitekture. Nakon toga se u nastavku ovog poglavlja opisuje implementacija pojedinih komponenti (servisa) aplikacije, a na kraju se opisuje i prikazuje implementacija klijentske aplikacije koja omogućava jednostavnu interakciju korisnika s web aplikacijom.

### 6.1. Opis web aplikacije

Za potrebe ovog rada razvijena je web aplikacija čije komponente, kada se povežu u jednu cjelinu, omogućavaju osnovnu funkcionalnost fiktivnog online dućana. Konkretno, za potrebe ovog rada razvijena je web aplikacija koja korisnicima omogućava registraciju i prijavu, pregledavanje proizvoda, dodavanje proizvoda u košaricu te kreiranje narudžbe. Arhitekturno, svaka od tih funkcionalnosti razvijena je u obliku zasebnog web servisa od kojih svaki zapravo predstavlja minijaturnu Java web aplikaciju od kojih svaka ima svoju bazu podataka (slika 21).



Slika 21. Arhitektura praktičnog primjera



Ovdje valja istaknuti kako je ova aplikacija razvijena s ciljem demonstracije istovremenog korištenja NoSQL i relacijskih baza podataka u razvoju aplikacija te su njene mogućnosti, u usporedbi s realnim web aplikacijama koje omogućavaju elektroničku trgovinu, poprilično ograničene. Isto tako valja napomenuti da, iako je arhitektura ove aplikacije inspirirana mikroservisima, njena arhitektura kao takva to nije. Prave mikroservisne arhitekture, osim samih servisa, imaju dosta drugih elemenata koji ih čine specifičnima.

S time na umu, funkcionalnost ove aplikacije ostvaruje se kroz četiri različita servisa od koji je svaki ograničen na određenu funkcionalnost uz koju se vežu određeni podatci. Točnije, u ovoj aplikaciji postoje četiri glavna entiteta (korisnici, proizvodi, košarica proizvoda te narudžbe) i za podatke svakog od njih zadužen je odgovarajući servis. Valja istaknuti kako su nazivi tih servisa na slici 21 prikazani su u **kontekstu** u kojem su oni korišteni **od strane klijentske aplikacije**. To npr., znači kako je servis koji se od strane klijentske aplikacije koristi za prijavu i registraciju korisnika zapravo zadužen za općenito upravljanje podacima tog entiteta, a ista stvar vrijedi i za ostale servise prikazane na toj slici.

Nadalje, sama klijentska aplikacija funkcionira na način da, ovisno o akcijama korisnika, šalje HTTP zahtjeve prema odgovarajućim servisima te potom obrađuje njihove odgovore. U ovom slučaju to ujedno i znači kako je sama klijentska aplikacija u pojedinim situacijama zadužena za integraciju podataka koji dolaze od strane različitih servisa prije nego li oni mogu biti prikazani krajnjem korisniku.

Više detalja o samoj implementaciji navedenih servisa, kao i same klijentske aplikacije, nalazi se u nastavku ovog poglavlja.

## 6.2. Implementacija servisa

Svaka komponenta ove web aplikacije zapravo je razvijena kao samostalan REST (*Representational State Transfer*) servis kako bi se omogućilo izvršavanje različitih CRUD (**Create**, **Read**, **Update**, **Delete**) operacija korištenjem HTTP GET, POST, PUT i DELETE metoda.

To je postignuto korištenjem **Spring** razvojnog okvira. Konkretno, svaki od servisa koji je dio ove aplikacije zapravo predstavlja zaseban **Spring Boot** projekt. Bez da ulazimo u previše detalja, sam *Spring* vjerojatno predstavlja najpoznatiji okvir za razvoj poslovnih (enterprise) aplikacija različitih vrsta. On se sastoji od mnogo različitih projekata koji programerima omogućavaju iskorištavanje već postojećih tehnologija kao što su okviri za objektno-relacijsko mapiranje (engl. *Object-relational mapping*, ORM), autentikaciju, i logiranje

itd. U tom okruženju *Spring Boot* predstavlja alat koji nam omogućava brzo i lagano kreiranje novih aplikacija koje se temelje na Spring razvojnom okviru.

No, jedan od najvažnijih razloga za odabir Springa jest što njegova obitelj projekata poznata pod nazivom **Spring Data** omogućava relativno jednostavnu integraciju i rad s različitim NoSQL i relacijskim bazama podataka, a što je za aplikacije temeljene na ovim tehnologijama baza podataka dosta važno.

Svakako valja istaknuti kako su prilikom razvoja ove web aplikacije također korišteni **MySQL** i **MongoDB** sustavi za upravljanje bazama podataka u kombinaciji su Docker<sup>8</sup> sustavom za kontejnerizaciju.

### 6.2.1. Servis za rad s korisnicima

Ovaj servis koristi se MySQL bazom podataka kako bi kroz svoje metode omogućio pristup i upravljanje podacima samih korisnika aplikacije. Ovaj, kao i svi ostali servisi opisani u ovom dijelu rada, povezuje se s konkretnom bazom podataka uz pomoć postavki koje se zadaju u *application.properties* datoteci (programski kôd 1).

```
spring.datasource.url=jdbc:mysql://localhost:3306/user-service-DB?allowPublicKeyRetrieval=true&useSSL=false
spring.datasource.username=userService
spring.datasource.password=userService
```

Programski kôd 1. Primjer postavki za povezivanje servisa s MySQL bazom podataka

```
@Entity @Table(name="users")
public class User {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="id")
    private Integer id;
    @Column(name="first_name")
    private String firstName;
    @Column(name="last_name")
    private String lastName;
    @Column(name="email")
    private String email;
    @Column(name="password")
    private String password;
    public User() {
        super();
    }
    ...
}
```

Programski kôd 2. Java klasa entiteta korisnika

---

<sup>8</sup> <https://www.docker.com/>

Nadalje, pošto se ovdje radi o relacijskoj bazi podataka, Spring omogućava automatsko generiranje tablica za sve entitete koju su u kôdu označeni odgovarajućim anotacijama na način kao što je to prikazano programskim kôdom 2. U slučaju ovog web servisa jedini entitet za kojeg je u bazi podataka trebalo kreirati tablicu bio je entitet korisnika.

```
public interface UserRepository extends CrudRepository<User, Integer> {
    User findByEmail(String email);
    List<User> findByFirstName(String firstName);
    List<User> findByLastName(String firstName);
}
```

### Programski kôd 3. Sučelje repozitorija za entitet korisnika

```
@RestController
public class UserController {
    @Autowired
    private UserRepository userRepository;
    @GetMapping("user-service/users")
    public @ResponseBody Iterable<User> getAllUsers() {
        return userRepository.findAll();
    }
    @GetMapping("user-service/users/{id}")
    public @ResponseBody User findById(@PathVariable int id) {
        Optional<User> user = userRepository.findById(id);
        if (!user.isPresent()) {
            throw new UserNotFoundException("userId-" + id);
        }
        return user.get();
    }
    @PostMapping("user-service/register")
    public void createNewUser(@RequestBody User user) {
        if (userRepository.findByEmail(user.getEmail()) == null) {
            userRepository.save(user);
        } else { throw new DuplicateEmail(); }
    }
    @GetMapping("/user-service/login/{email}/{password}")
    public @ResponseBody User authenticateUser(@PathVariable
        String email, @PathVariable String password) {
        User user = userRepository.findByEmail(email);
        if (user != null) {
            if (user.getPassword().equals(password)) {
                return user;
            } else { throw new WrongCredentials(); }
        } else { throw new WrongCredentials(); }
    }
    @PostMapping("user-service/update")
    public void updateUser(@RequestBody User user) {
        userRepository.save(user);
    }
}
```

### Programski kôd 4. REST kontroler servisa za upravljanje korisnicima

Nakon definiranja entiteta, slijedeći korak u implementaciji servisa bio je kreiranje njegovog sučelja repozitorija (programski kôd 3) koje je, između ostaloga, zaduženo za

izvršavanje različitih promjena nad podacima tog entiteta u samoj bazi podataka. Valja istaknuti kako ovo sučelje nije potrebno samostalno implementirati već se za njegovu automatsku implementaciju brine sam Spring.

Zadnji korak u implementaciji ovog servisa bio je implementacija REST kontrolera koji prima i obrađuje zahtjeve klijenata te im vraća podatke serializirane u JSON formatu. Kao što je prikazano programskim kôdom 4 (str. 52) ovdje se zapravo radi o običnoj Java klasi koja je anotirana posebnom `@RestController` anotacijom te čije su metode također anotirane na način kojim postaju krajnje točke za obradu zahtjeva koji dolaze od strane klijenta.

Implementacija ovog web servisa za rad s korisnicima sastojala se od tri glavna koraka, a to su: definiranje entiteta, kreiranje sučelja repozitorija za taj entitet te kreiranje REST kontrolera unutar kojeg se nalaze posebno anotirane metode koje predstavljaju krajnje točke za obradu HTTP zahtjeva koji dolaze od strane klijenata. Od ovih koraka također se sastojala i implementacija preostalih servisa web aplikacije.

### 6.2.2. Servis za upravljanje proizvodima

Glavni entitet o čijim podacima se brine ovaj servis je proizvod. Općenito, proizvodi u online dućanima mogu imati puno različitih atributa zbog čega za pohranu njihovih podataka puno više odgovaraju NoSQL baze podataka. Iz tog razloga ovaj servis koristi MongoDB sustav kako bi se podatci o proizvodima spremali u obliku JSON dokumenata. Programski kôd 5 prikazuje klasu entiteta proizvoda pomoću kojeg možemo vidjeti koje sve attribute imaju proizvodi ove web aplikacije.

```
@Document(collection = "products")
public class Product {
    @Id
    String id;
    String name;
    String description;
    List<String> keywords;
    String image;
    List<String> age;
    Double price;

    public Product() {
        super();
    }
    ...
}
```

Programski kôd 5. Klasa entiteta proizvoda

Iako ovaj servis koristi drugačiju vrstu baze podataka sam postupak njegove implementacije je, zahvaljujući Springu, bio jako sličan prethodno opisanoj implementaciji

servisa za upravljanje podacima korisnika. I u ovom slučaju je postavke za uspostavljanje veze s bazom podataka trebalo definirati unutar **application.properties** datoteke te je nakon definiranja entiteta bilo potrebno kreirati odgovarajuće sučelje repozitorija za entitet proizvoda kako bi servis mogao upravljati podacima koji se nalaze u bazi podataka.

```
@RestController
public class ProductController {
    @Autowired
    private ProductRepository productRepository;
    @GetMapping("product-service/productList")
    public @ResponseBody List<Product> getAllProudcts() {
        return productRepository.findAll();
    }
    @GetMapping("product-service/products/{id}")
    public Product getProductById(@PathVariable String id) {
        Optional<Product> product = productRepository.findById(id);
        if (product.isPresent()) {
            return product.get();
        } else {
            throw new ProductNotFoundException("product ID: " + id);
        }
    }
    @GetMapping("product-service/products/findByKeywords/{keywords}")
    public List<Product> findByKeywords(@PathVariable String[] keywords) {
        return productRepository.findByKeywordsIn(Arrays.
            asList(keywords));
    }
    @GetMapping("product-service/products/findByName/{name}")
    public List<Product>
        findByKeywordsIn(@PathVariable String name) {
            return productRepository.findByName(name);
        }
    @GetMapping("product-service/products/findByIds/{productIds}")
    public @ResponseBody List<Product>
        findAllProductsById(@PathVariable String[] productIds) {
        List<Product> result = new ArrayList<>();
        productRepository.findAllById(Arrays.
            asList(productIds)).forEach(result::add);
        return result;
    }
    @PostMapping("product-service/saveProduct")
    public void saveProudct(@RequestBody Product product) {
        productRepository.save(product);
    }
    @DeleteMapping("product-service/deleteProduct/{id}")
    public void deleteProduct(@PathVariable String id) {
        productRepository.deleteById(id);
    }
}
```

#### Programski kôd 6. REST kontroler servisa za upravljanje proizvodima

Zadnji korak u implementaciji također je bio kreiranje odgovarajućeg REST kontrolera u kojem se nalaze različite krajnje točke na koje klijenti mogu slati svoje zahtjeve ovisno o tome što žele postići. Sam REST kontroler i putanje do njegovih krajnjih točaka na koje je

moguće slati HTTP zahtjeve prema ovome servisu za upravljanje proizvodima prikazani su programskim kôdom 6.

### 6.2.3. Servis za upravljanje košaricama proizvoda

Korisnici web aplikacije trebaju prije kreiranja nove narudžbe popuniti svoju košaricu željenim proizvodima. Ovaj servis kreiran je kako bi se omogućilo da stanje tih korisničkih košarica ostane spremljeno i nakon što oni napuste web aplikaciju.

Iako bi za implementaciju ovog servisa vjerojatno najbolje odgovarala ključ/vrijednost baza podataka, odlučeno je kako će se zbog jednostavnosti podatci kojima upravlja ovaj servis spremati unutar postojećeg MongoDB sustava. No, bez obzira na to što ovaj servis i servis za upravljanje proizvodima dijele isti sustav za upravljanje podatcima, unutar samog MongoDB sustava njihovi podatci su odvojeni jedan od drugih na način da svaki od ovih servisa u tom sustavu ima svoju bazu podataka.

```
@Document(collection = "carts")
public class Cart {
    @Id
    String id;
    int userId;
    List<CartItem> cartItems;

    public Cart() {
        super();
    }
    ...

    public class CartItem {
        String productId;
        Integer quantity;

        public CartItem() {
            super();
        }
        ...
    }
}
```

Programski kôd 7. *Cart* i *CartItem* Java klase

Programskom kôdom 7 prikazane su klase na kojima se temelji struktura dokumenata u koje se pohranjuju podatci o stanju košarice za svakog korisnika aplikacije.

Nadalje, najvažniji dio ovog servisa predstavlja njegov REST kontroler prikazan programskim kôdom 8 (str. 56). On svojim metodama omogućava spremanje i dohvaćanje podataka o košaricama iz baze podataka. Ovdje valja istaknuti kako je metoda `saveCart` koja je zadužena za spremanje podataka o košaricama implementirana na način koji bi trebao spriječiti kreiranje više različitih košarica unutar baze podataka za istog korisnika.

```

@RestController
public class CartController {

    @Autowired
    private CartRepository cartRepository;

    @GetMapping("cart-service/retrieveUsersCart/{userId}")
    public @ResponseBody List<CartItem> retrieveUsersCart(
        @PathVariable Integer userId) {

        Cart cart = cartRepository.findByUserId(userId);
        if (cart != null && cart.getCartItems() != null) {
            return cart.getCartItems();
        } else {
            return new ArrayList<>();
        }
    }

    @PostMapping("cart-service/saveCart")
    public void saveCart(@RequestBody Cart cart) {
        Cart usersCart = cartRepository.findByUserId(cart.getUserId());
        if (usersCart != null) {

            usersCart.setCartItems(cart.getCartItems());
            cartRepository.save(usersCart);
        } else {
            cartRepository.save(cart);
        }
    }
}

```

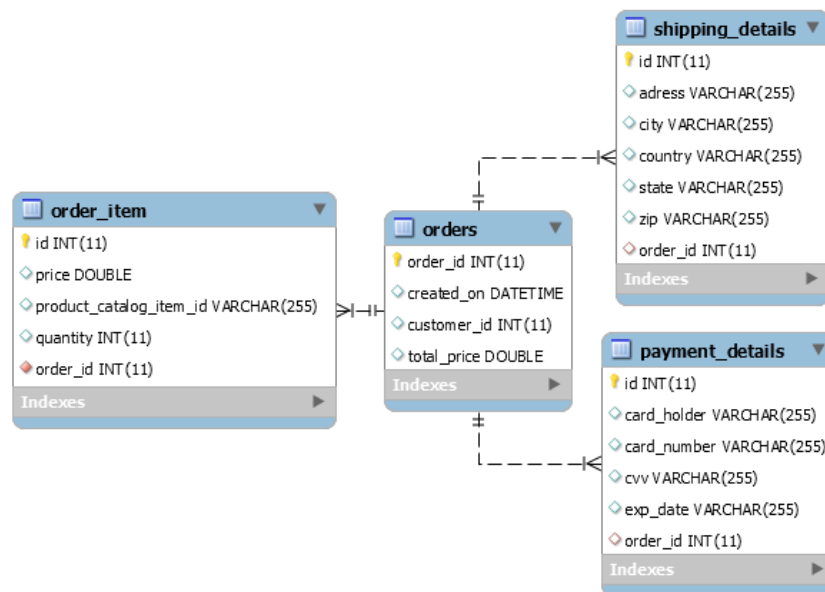
Programski kôd 8. REST kontroler servisa za upravljanje košaricama

## 6.2.4. Servis za upravljanje narudžbama

Ovaj servis primarno je razvijen kako bi se korisnicima web aplikacije omogućilo kreiranje narudžbi za artikle koji se nalaze u njihovoj košarici proizvoda. Za spremanje podataka o tim narudžbama koristi se MySQL baza podataka. Pošto su podatci o samim proizvodima spremljeni u MongoDB sustavu, bilo je potrebno unutar tablice u koju se pohranjuju podatci o stavkama narudžbe, u MySQL bazi, definirati poseban atribut koji će predstavljati vezu između odgovarajuće stavke narudžbe i proizvoda kojeg ta stavka predstavlja. U ovom slučaju je tom atributu dodijeljen naziv `PRODUCT_CATALOG_ITEM_ID`. Iz istog razloga je u samu tablicu narudžbi dodana kolona s nazivom `CUSTOMER_ID` kako bi se mogao identificirati korisnik koji je kreirao narudžbu, a čiji podatci su spremljeni u bazi drugoj MySQL bazi podataka kojom upravlja servis koji je zadužen za korisničke podatke.

Prilikom razvoja ovog servisa su, uz narudžbe i njihove stavke, kao posebni entiteti prepoznati i detalji o adresi dostave te detalji o plaćanju. Prema tome, prilikom razvoja ovog servisa najprije je za svakog od njih trebalo kreirati i pravilno anotirati Java klase kako bi Spring

potom mogao unutar MySQL baze samostalno kreirati tablice i veze koje postoje među njima. Na taj način dobiven je ERA model prikazan na slijedećoj slici.



Slika 22. ERA model baze podataka servisa za upravljanje narudžbama

```
@Entity
@Table(name = "orders")
public class Order {

    @Id Column(name = "order_id")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;

    private Integer customerId;
    private Date createdOn;
    private Double totalPrice;

    @OneToMany(cascade = CascadeType.ALL,
        fetch = FetchType.LAZY,
        mappedBy = "order")
    private List<OrderItem> orderItems;

    @OneToOne(mappedBy = "order", cascade = CascadeType.ALL,
        fetch = FetchType.LAZY,
        optional = false)
    private PaymentDetails paymentDetails;

    @OneToOne(mappedBy = "order", cascade = CascadeType.ALL,
        fetch = FetchType.LAZY,
        optional = false)
    private ShippingDetails shippingDetails;
    ...
}
```

Programski kôd 9. Atributi klase koja predstavlja narudžbu



Programski kôd 9 prikazuje najvažnije dijelove Java klase na temelju koje je Spring unutar baze podataka kreirao tablicu narudžbi prikazanu na slici 22.

Slijedeći koraci u implementaciji ovog servisa bili su kreiranje odgovarajućeg sučelja repozitorija te REST kontrolera kako bi se klijentskoj aplikaciji omogućila interakcija s ovim servisom. U programskom kôdu 10 prikazana je trenutna implementacija REST kontrolera kojim se između ostaloga omogućava kreiranje novih narudžbi, te njihovo dohvaćanje putem jedinstvenog identifikatora narudžbe (ID narudžbe) ili pak preko atributa CUSTOMER\_ID kojim se omogućava dohvaćanje svih narudžbi nekog kupca. Valja istaknuti kako se ovaj, kao i svi ostali REST kontroleri opisani u ovom dijelu rada mogu, ovisno o potrebama, proširiti dodatnim metodama i krajnjim točkama.

```
@RestController
public class OrderController {
    @Autowired
    private OrderRepository orderRepository;

    @PostMapping("order-service/create")
    public void createNewOrder(@RequestBody Order order) {
        order.setCreatedOn(new Date());
        order.getPaymentDetails().setOrder(order);
        order.getShippingDetails().setOrder(order);
        order.getOrderItems().forEach(item -> {item.setOrder(order);});
        orderRepository.save(order);
    }

    @GetMapping("order-service/orders")
    public @ResponseBody Iterable<Order> getAllOrders() {
        return orderRepository.findAll();
    }

    @GetMapping("order-service/orders/{id}")
    public @ResponseBody Order findOrderById(@PathVariable Integer id) {

        Optional<Order> order = orderRepository.findById(id);

        if (!order.isPresent()) {
            throw new OrderNotFoundException("orderId-" + id);
        }
        return order.get();
    }

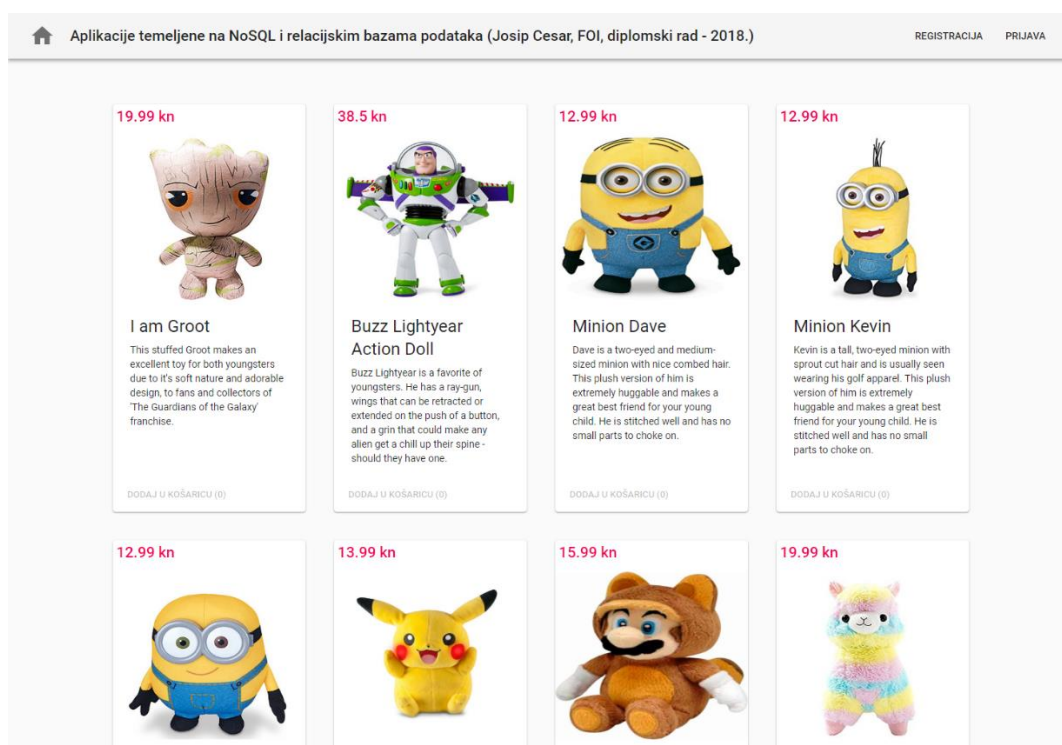
    @GetMapping("order-service/orders/findByCustomerId/{customerId}")
    public @ResponseBody List<Order> findByCustomerId(
        @PathVariable Integer customerId) {
        return orderRepository.findByCustomerId(customerId);
    }
}
```

Programski kôd 10. REST kontroler servisa za upravljanje narudžbama

## 6.3. Implementacija i pregled klijentske aplikacije

Kako bi se omogućila jednostavna interakcija s web aplikacijom napravljena je posebna klijentska aplikacija. Za implementaciju njenog korisničkog sučelja korištena je popularna **React** biblioteka. Ovu biblioteku razvio je Facebook te ju je objavio 2013. godine, a danas ona predstavlja jedan od najpoznatijih alata za razvoj i izradu programskih sučelja. No, valja istaknuti kako se ova biblioteka primarno fokusira na razvoj različitih komponenata korisničkog sučelja te kao takva ne pokriva sva područja vezana uz razvoj klijentskih web aplikacija. Iz tog razloga su, uz React, prilikom implementacije ove klijentske aplikacije korištene i druge popularne biblioteke kao što su *redux*, *react-redux*, *redux-thunk* te *react-router* koje omogućavaju lakšu implementaciju navigacije i lakše upravljanje stanjem aplikacije. Uz to, prilikom razvoja klijentske aplikacije korišten je veliki broj vizualnih komponenata temeljenih na materijalnom dizajnu koje su nam dostupne preko *Material-UI* platforme.

S time na umu, slijedeća slika prikazuje početni zaslon klijentske web aplikacije. Na njoj vidimo kako neregistrirani korisnici mogu pregledavati artikle koji su dio kataloga proizvoda, ali ih ne mogu dodavati u košaricu proizvoda.



Slika 23. Početni zaslon klijentske aplikacije

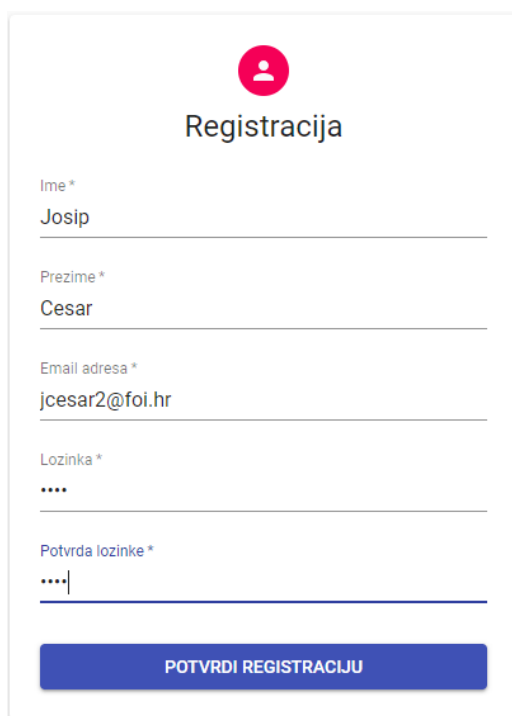
S početnog zaslona neprijavljeni korisnici se mogu prijaviti ili pak se prije toga mogu registrirati, ukoliko je potrebno. U oba ova slučaja aplikacija komunicira isključivo s servisom

koji je zadužen za upravljanje korisnicima. Za slanje zahtjeva prema servisima i dohvaćanje resursa unutar cijele klijentske aplikacije koristi se **Javascript Fetch API**. Slijedeći programski kôd prikazuje primjer korištenja *Fetch API*-ja unutar funkcije koju aplikacija koristi za slanje POST zahtjeva prema servisu za upravljanje korisnicima prilikom registracije novih korisnika.

```
function register(user) {  
  const requestOptions = {  
    method: 'POST',  
    headers: { 'Content-Type': 'application/json' },  
    body: JSON.stringify(user)  
  };  
  return fetch(`/user-service/register`,  
    requestOptions).then(handleResponse);  
}
```

#### Programski kôd 11. Primjer korištenja *Javascript Fetch API*-ja

Nadalje, slika 24 prikazuje izgled registracijske forme prema kojoj se može vidjeti kako korisnik za uspješnu registraciju mora unijeti svoje ime i prezime, email adresu te lozinku.



The image shows a registration form with a white background and a light gray border. At the top center is a red circular icon with a white person silhouette. Below it is the title 'Registracija' in a bold, dark gray font. The form contains five input fields, each with a label above it: 'Ime \*' with the value 'Josip', 'Prezime \*' with the value 'Cesar', 'Email adresa \*' with the value 'jcesar2@foi.hr', 'Lozinka \*' with four dots, and 'Potvrda lozinke \*' with four dots and a cursor. At the bottom is a blue button with the text 'POTVRDI REGISTRACIJU' in white capital letters.

Slika 24. Forma za registraciju novih korisnika

Nakon što korisnik potvrdi registraciju, ukoliko sve prođe u redu on će automatski biti preusmjeren na formu za prijavu prikazanu na slici 25.

The image shows a login form with a white background and a light gray border. At the top center is a red circular icon with a white lock symbol. Below it is the title 'Prijava' in a bold, black font. There are two input fields: the first is labeled 'Email adresa \*' and the second is labeled 'Lozinka \*'. Both labels are in a small, gray font. Below the input fields is a blue button with the text 'PRIJAVI SE' in white, uppercase letters.

Slika 25. Forma za prijavu

```
<React.Fragment>
  <CssBaseline />
  <main className={classes.layout}>
    <Paper className={classes.paper}>
      <Avatar className={classes.avatar}>
        <LockIcon />
      </Avatar>
      <Typography variant="headline">Prijava</Typography>
      <form className={classes.form} onSubmit={this.handleSubmit}>

        <FormControl margin="normal" required fullWidth>
          <InputLabel htmlFor="loginEmail">Email adresa</InputLabel>
          <Input id="loginEmail" name="loginEmail"
            autoComplete="email"
            value={this.state.loginEmail} autoFocus
            onChange={this.handleChange}/>
        </FormControl>











        <FormControl margin="normal" required fullWidth >
          <InputLabel htmlFor="loginPassword">Lozinka</InputLabel>
          <Input name="loginPassword" type="password"
            id="loginPassword" value={this.state.loginPassword}
            onChange={this.handleChange} />
        </FormControl>
        <Button type="submit" fullWidth variant="raised"
          color="primary" className={classes.submit} >
          Prijavi se
        </Button>
      </form>
    </Paper>
  </main>
</React.Fragment>
```

Programski kôd 12. Programski kôd forme za prijavu

Programski kôd 12 prikazuje implementaciju forme za prijavu te predstavlja općeniti primjer toga na koji način su implementirani različiti vizualni dijelovi ove klijentske aplikacije uz pomoć različitih *Material-UI* komponenata.

Nadalje, prijavom u aplikaciju korisnici se vraćaju na početni zaslon na kojem sada imaju mogućnost dodavanja i uklanjanja proizvoda iz svoje košarice. Svaki puta kada neki korisnik doda ili ukloni proizvod iz košarice klijentska aplikacija o tome obavještava odgovarajući servis kako bi se podatci o stanju košarice svakog korisnika spremili u bazu podataka. Na taj način stanje košarice svakog korisnika ostaje sačuvano i nakon što on napusti aplikaciju.

Korisnici aplikacije također imaju mogućnost da stanje svoje košarice provjere na posebnom zaslonu na kojem se nalazi komponenta prikazana slikom 26. Prilikom otvaranja tog zaslona aplikacija treba uzeti podatke o tome koji proizvodi se nalaze u košarici te od servisa koji se brine o podacima proizvoda zatražiti njihove detalje. To je potrebno iz razloga što je sama košarica implementirana tako da se uz jedinstvene identifikatore (tj. primarne ključeve) proizvoda u nju sprema samo podatak o tome kolika količina navedenog proizvoda se nalazi u košarici. Svi ostali podatci o proizvodu kao što su njegova slika, naziv, cijena i sl. se ne spremaju u košaricu već ih klijentska aplikacija u ovom slučaju mora zatražiti od odgovarajućeg servisa.

Košarica proizvoda						
Artikl	Naziv	Količina	Cijena			
	I am Groot	2	19.99	+	-	<button>UKLONI</button> 
	Buzz Lightyear Action Doll	1	38.5	+	-	<button>UKLONI</button> 
	Minion Bob	2	12.99	+	-	<button>UKLONI</button> 
	Pikachu Pokémon	1	13.99	+	-	<button>UKLONI</button> 
	Llama	1	19.99	+	-	<button>UKLONI</button> 
						<button>CHECKOUT</button>

Slika 26. Prikaz košarice proizvoda

Nadalje, prilikom pregledavanja košarice korisnici mogu povećavati i smanjivati količine proizvoda koji se nalaze u košarici, ili pak mogu određeni proizvod u potpunosti maknuti iz

košarice. Osim toga, odavde se korisnici mogu ili vratiti na početni ekran ili pak kreirati narudžbu za proizvode u košarici. Taj postupak kreiranja nove narudžbe korisnici započinju klikom na gumb *Checkout* koji se može vidjeti na slici 26.

Postupak kreiranja narudžbe kod ove klijentske aplikacije predstavlja proces koji se sastoji od tri koraka, prikazanih slikama 27, 28 i 29. U prvom koraku korisnik unosi detalje vezane uz adresu dostave. U drugom koraku korisnik treba unijeti podatke koji se odnose na plaćanje, a u trećem, finalnom koraku korisnicima je omogućeno da još jednom prouče sve detalje vezane uz narudžbu te, ukoliko je sve u redu, kreiranje narudžbe mogu potvrditi klikom na odgovarajući gumb.

The screenshot shows the 'Checkout' screen with a progress bar at the top indicating three steps: 1. Adresa za dostavu (selected), 2. Detalji plaćanja, and 3. Potvrda narudžbe. The form is titled 'Adresa za dostavu' and contains several input fields: 'Ime' (filled with 'Josip'), 'Prezime' (filled with 'Cesar'), 'Adresa' (filled with 'Pavlinska 2'), 'Grad' (filled with 'Zagreb'), 'Županija/Općina' (filled with 'Grad Zagreb'), 'Broj pošte' (filled with '10000'), and 'Zemlja' (filled with 'Hrvatska'). A blue 'DALJE' button is located at the bottom right.

Slika 27. Forma za unos adrese za dostavu

The screenshot shows the 'Checkout' screen with a progress bar at the top indicating three steps: 1. Adresa za dostavu (completed with a checkmark), 2. Detalji plaćanja (selected), and 3. Potvrda narudžbe. The form is titled 'Detalji plaćanja' and contains several input fields: 'Ime na kartici' (filled with 'JOSIP CESAR'), 'Broj kartice' (filled with '4574 1697 8527 8001'), 'Datum isteka' (filled with '10/20'), and 'CVV' (filled with '598'). There is a label 'Zadnje tri znamenke na magnetskoj traci' below the CVV field. A 'POVRATAK' button is at the bottom left, and a blue 'DALJE' button is at the bottom right.

Slika 28. Forma za unos detalja o plaćanju

## Checkout

✓ Adresa za dostavu — ✓ Detalji plaćanja — **3** Potvrda narudžbe

### Pregled narudžbe

I am Groot X 2	39.98
Buzz Lightyear Action Doll X 1	38.5
Minion Bob X 2	25.98
Pikachu Pokémon X 1	13.99
Llama X 1	19.99
<b>Ukupno</b>	<b>138.44</b>

### Dostava

Josip Cesar  
Pavlinska 2, Zagreb, Grad Zagreb, 10000,  
Hrvatska

### Detalji plaćanja

Vlasnik kartice	JOSIP CESAR
Broj Kartice	4574 1697 8527 8001
Datum isteka	10/20

[POVRATAK](#)
[NARUČI](#)

Slika 29. Forma za finalni pregled detalja o narudžbi te potvrdu njenog kreiranja

Potvrdom kreiranja klijentska aplikacija će sve ove podatke proslijediti servisu za upravljanje narudžbama koji će ih potom spremi u bazu podataka. Ovdje valja istaknuti kako će se tom prilikom u istoj bazi spremi samo jedinstveni identifikatori proizvoda i korisnika kako bi se na taj način stvorila veza između konkretne narudžbe i proizvoda koje je korisnik imao u svojoj košarici za vrijeme njenog kreiranja te njega samog. To je važno istaknuti jer bez toga ponovno sastavljanje narudžbe u oblik u kojem je prikazana na slici 29 ne bi bio moguć.

## 7. Osvrt na aplikacije temeljene na NoSQL i relacijskim bazama podataka

Nakon detaljnog proučavanja tematike vezane uz aplikacije temeljene na NoSQL i relacijskim bazama podataka, te vlastitog pokušaja implementacije takve aplikacije mislim da je važno da da u malo više detalja opišem svoj pogled na ovu vrstu aplikacija.

Najprije treba istaknuti kako smatram da istovremeno korištenje različitih baza podataka u razvoju većine jednostavnih aplikacija nije potrebno. Razlog tome je što, iako ovaj pristup može omogućiti bolje upravljanje podacima, njegova primjena u većini slučajeva dolazi pod cijenu povećanja kompleksnosti koju moramo biti spremni platiti. Naime, to povećanje kompleksnosti osjetio sam već i prilikom razvoja svojeg podosta jednostavnog praktičnog primjera. Naime, iako sam s jedne strane postigao bolje upravljanje podacima, cijena toga se ponajviše mogla osjetiti prilikom razvoja klijentske aplikacije u trenucima kada je trebalo podatke koju su rasprostranjeni između različitih servisa i baza podataka ponovno spojiti zajedno, u jednu cjelinu, na klijentskoj strani. U takvim situacijama je trebalo uložiti dodatni trud kako bi se postigao željeni rezultat.

Uz to, valja istaknuti kako idealan način integracije različitih baza podataka unutar iste aplikacije jednostavno ne postoji. Svaki od pristupa koji su opisani u poglavlju 5.4.1 imaju svoje prednosti i nedostatke te programeri i arhitekti aplikacija moraju sami odlučiti koji će pristup ili kombinacija pristupa najbolje odgovarati njihovoj aplikaciji.

Ovdje prije svega želim istaknuti kako istovremeno korištenje različitih baza podataka unutar iste aplikacije može biti dosta komplicirano. Iz tog razloga ovaj pristup bismo trebali primjenjivati samo u situacijama kod kojih za time postoji **stvarna potreba** te tom prilikom zadržati ukupni broj različitih baza podataka što manjim.

Uz to, još jedna stvar na koju treba obratiti pažnju u ovom slučaju jest da su mnogi veliki proizvođači relacijskih sustava za upravljanje podacima već u svoje proizvode ugradili određene NoSQL značajke. Samim time, ta dva svijeta su se počela polagano spajati. Za nas to znači da ukoliko razvijamo aplikaciju koja se temelji na relacijskoj bazi podataka, ali u kojoj također postoji potreba za određenim NoSQL značajkama, kao što je npr. podrška za spremanje JSON dokumenata i sl., tada vjerojatno možemo u potpunosti izbjeći dodavanje posebnog NoSQL sustava u arhitekturu aplikacije.



## 8. Zaključak

U ovom radu imali smo priliku upoznati se s tehnologijama NoSQL i relacijskih baza podataka koje predstavljaju dva različita pristupa upravljanju podacima. Relacijski pristup podrazumijeva pohranu podataka unutar relacija koje imaju dobro definiranu strukturu, dok je NoSQL pristup puno fleksibilniji te omogućava izbor između različitih modela podataka. U današnje vrijeme relacijske baze podataka još uvijek u većini slučajeva predstavljaju prvi izbor što i ne čudi pošto one predstavljaju dobro razvijenu i zrelu tehnologiju. NoSQL baze podataka se s druge pak strane u produkciji većinom koriste kao rješenja za specifične probleme koji su se pojavili s razvojem weba i tehnologije općenito, a za koje relacijske baze podataka nisu dizajnirane.

Izbor baze podataka prilikom razvoja aplikacija ponajviše ovisi o njenim potrebama. Ukoliko se radi o nekoj poslovnoj aplikaciji koja će biti korištena samo unutar organizacije tada će relacijska baza podataka vrlo vjerojatno moći zadovoljiti većinu njenih potreba. S druge strane, ukoliko se radi o aplikaciji koja će biti usmjerena na web te kod koje se očekuje nagli rast u broju korisnika i količini podataka tada NoSQL ima više smisla. No, ovaj izbor se ne mora uvijek isključivo temeljiti na količini podataka i skalabilnosti. Važnu ulogu u procesu odabira tehnologije i pristupa razvoju aplikacijske baze podataka mogu imati i karakteristike samih podataka te način na koji se oni koriste unutar aplikacije. Činjenica je da se svi podatci ne uklapaju savršeno u relacijski model podataka. Postoje slučajevi u kojima se unutar jedne aplikacije ili sustava mogu obrađivati različite vrste podataka, a u takvim slučajevima istovremeno korištenje različitih NoSQL i relacijskih baza podataka može predstavljati dobar pristup ka rješavanju problema i ostvarivanju boljeg i efikasnijeg upravljanja podacima.

U ovom radu bili su opisani i različiti pristupi integraciji NoSQL i relacijskih baza podataka unutar iste aplikacije. Tada se pokazalo kako mikroservisna arhitektura vjerojatno predstavlja jedan od najboljih načina na koji se takvo nešto može postići. No, arhitektura mikroservisa nije uvijek najbolji i najprikladniji izbor arhitekture. Nadalje, treba imati na umu kako će istovremeno korištenje različitih baza podataka unutar iste aplikacije vjerojatno povećati kompleksnost i troškove njenog razvoja, a i kasnijeg održavanja.

Moj je konačni zaključak kako je razvoju aplikacije temeljene na NoSQL i relacijskim bazama najbolje pristupiti s oprezom, jer ipak je generalna preporuka da broj baza u sustavu uvijek bude što manji. No, isto tako smatram da NoSQL i relacijske baze podataka mogu funkcionirati zajedno unutar iste aplikacije upravo iz razloga što se ovdje radi o različitim tehnologijama koje ne mogu u potpunosti zamijeniti jedna drugu, ali zajedno mogu riješiti većinu problema vezanih uz upravljanje podacima koji se mogu pojaviti u nekoj aplikaciji.

## Popis literature

*Advantages Of NoSQL* (n.d.). Preuzeto 10.05.2018. s

<https://www.mongodb.com/scale/advantages-of-nosql>

Brkić, L., i Mekterović, I. (2017). *NoSQL. Napredni modeli i baze podataka* [Nastavni materijali]. Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, Zagreb. Preuzeto 25.04.2018. s

[https://www.fer.unizg.hr/download/repository/6.NoSQL\\_\(1\\_od\\_4\)\\_-\\_RDB\\_podsjetnik.pdf](https://www.fer.unizg.hr/download/repository/6.NoSQL_(1_od_4)_-_RDB_podsjetnik.pdf)

Chamberlin, D. D. (2012). Early History of SQL. *IEEE Annals of the History of Computing*, 34(4):78-82.

Engelschall, R. S. (2013). *Polyglot Persistence: Boon and Bane for Software Architects* [Video file]. Preuzeto 12.05.2018. s

<https://www.youtube.com/watch?v=3rhlitKEUXo>

Fowler, M. (16.11.2011). *PolyglotPersistence* [slika]. Preuzeto 26.06.2018. s

<https://martinfowler.com/bliki/PolyglotPersistence.html>

Fowler, M. (n.d.). *ApplicationDatabase*. Preuzeto 23.06.2018. s

<https://martinfowler.com/bliki/ApplicationDatabase.html>

Fowler, M., i Lewis, J. (25.3.2014). *Microservices*. Preuzeto 25.07.2018. s

<https://martinfowler.com/articles/microservices.html>

Fowler, M., i Sadalage, P. J. (2013). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. USA: Addison-Wesley Professional.

Gaspar, D., i Coric, I. (2017). *Bridging Relational and NoSQL Databases*. E. Chocolate Avenue Hershey, PA, USA: IGI Global.

Grolinger, K., Higashino, W. A., Tiwari, A., i Capretz, M AM. (2013). Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2:22

Harrison, G. (2015). *Next Generation Databases: NoSQL and Big Data*. Berkely, CA, USA: Apress.

Hills, T. (2016). *NOSQL AND SQL DATA MODELING: Bringing Together Data, Semantics, and Software*. Technics Publications. Basking Ridge, NJ, USA: Technics Publications.

Hofmann, M., Schnabel, E., i Stanley, K. (2016). *Microservices Best Practices for Java*. IBM Redbooks.

*How is scalability achieved?* [Slika] (04.13.2012). Preuzeto 3.06.2018. s <https://hadoop4usa.wordpress.com/2012/04/13/scale-out-up/>

Maleković, M., i Rabuzin, K. (2016). Uvod u baze podataka. Varaždin: Mini-Print-Logo d.o.o.

McCreary, D., i Kelly, A. (2014). *Making Sense of NoSQL: A guide for managers and the rest of us*. Shelter Island, NY, USA: Manning Publications Co.

Mišić, I. (10.7.2009). *MySQL Tipovi podataka – Data Types*. Preuzeto 26.07.2018. s <https://www.hdonweb.com/programiranje/mysql-tipovi-podataka-data-types>

Musa, K. (22.04.2017). *Održivi mikroservisi*. Preuzeto 15.08.2018. s <https://blog.croz.net/blog/odrzivi-mikroservisi/>

Neo4j Inc. (2018). *The Neo4j Developer Manual v3.4*. Preuzeto 04.08.2018. s <https://neo4j.com/docs/developer-manual/current/>

Pećanac, N. (n.d.). *Mikroservisna Arhitektura: Dropwizard Vs Spring Boot*. Preuzeto 13.08.2018. s <http://2017.javacro.hr/content/download/6712/114399/file/313-Pe%C4%87anac+mikroserv+arh.pdf>

Rabuzin, K. (2011). *Uvod u SQL*. Varaždin: Tiva.

Richards, M. (2016). *Microservices vs. Service-Oriented Architecture* (prvo poglavlje). Preuzeto 15.08.2018. s <https://www.safaribooksonline.com/library/view/microservices-vs-service-oriented/9781491975657/ch01.html>

- Sambolek, S. (2015). *NewSQ: pregledni rad*. Preuzeto 26.06.2018. s [http://www.inf.uniri.hr/files/studiji/poslijediplomski/kvalifikacijski/KDI\\_Sasa\\_Sambolek.pdf](http://www.inf.uniri.hr/files/studiji/poslijediplomski/kvalifikacijski/KDI_Sasa_Sambolek.pdf)
- Šestak, M. (2016). *Usporedba jezika za graf baze podataka* (diplomski rad). Fakultet organizacije i informatike, Varaždin, Sveučilište u Zagrebu.
- Stojanović, A. (2016). *OSVRT NA NOSQL BAZE PODATAKA – ČETIRI OSNOVNE TEHNOLOGIJE*. Polytechnic and design, 4 (1), 44-53. Preuzeto 4.05.2018. s <https://doi.org/10.19279/TVZ.PD.2016-4-1-06>
- Tozzi, C. (31.05.2016) *The Limitations of NoSQL Database Storage: Why NoSQL's Not Perfect*. Preuzeto 20.08.2018. s <https://www.channelfutures.com/cloud-services/limitations-nosql-database-storage-why-nosqls-not-perfect>
- What is a Column Store Database?* (23.06.2013). Preuzeto 25.07.2018. s <https://database.guide/what-is-a-column-store-database/>
- Žarko, I. P., Lovrek, I., Kušek, M., i Pripužić, K. (9. 1 2017). *Mikrousluge*. Raspodijeljeni sustavi [Nastavni materijal]. Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, Zagreb. Preuzeto 9.08.2018 s [https://www.fer.unizg.hr/download/repository/RS-2017\\_11\\_dodatno.pdf](https://www.fer.unizg.hr/download/repository/RS-2017_11_dodatno.pdf)

# Popis slika

Slika 1. Strukturalni koncepti relacija u relacijskom modelu .....	3
Slika 2. Primjer problema neusklađenosti modela (Izvor: Fowler i Sadalage, 2013) .....	6
Slika 3. CAP teorem .....	11
Slika 4. Vertikalna i horizontalna skalabilnost (Izvor: „How is scalability achieved?“, 2012) ..	13
Slika 5. Ključ/vrijednost model podataka (autorski rad).....	15
Slika 6. Primjeri dokumenata u JavaScript notaciji (autorski rad) .....	16
Slika 7. Struktura retka u stupčanim bazama podataka (Prema: „What is a Column Store Database?“, 2016).....	18
Slika 8. Struktura super-stupaca (autorski rad) .....	18
Slika 9. Prikaz primjera porodice stupaca (Prema: „What is a Column Store Database?“, 2016).....	19
Slika 10. Primjer grafa (Izvor: Stojanović, 2016) .....	20
Slika 11. MySQL Workbench sučelje .....	23
Slika 12. Sučelje alata MongoDB Compass.....	25
Slika 13. Neo4j Browser sučelje .....	27
Slika 14. Primjer tradicionalne monolitne aplikacije (Izvor: Žarko, Lovrek, Kušek, Pripužić, 2017).....	36
Slika 15. Klijent-poslužitelj model (Izvor: Žarko, Lovrek, Kušek, Pripužić, 2017) .....	37
Slika 16. Primjer poslovne aplikacije po polyglot persistenceu (Prema: Fowler, 2011).....	40
Slika 17. Primjer integracije različitih SUBP-ova korištenjem višestrukih linija (Prema: Engelschnall, 2013) .....	42
Slika 18. Pristup integraciji SUBP-ova korištenjem Polyglot Mapper alata (Prema: Engelschnall, 2013) .....	43
Slika 19. Primjer integracije NoSQL i relacijskih SUBP-ova pomoću ugniježdene baze podataka (Prema: Engelschnall, 2013) .....	43
Slika 20. Primjer integracije NoSQL i relacijskog pristupa korištenjem „svemoguća baze podataka“ (Prema: Engelschnall, 2013).....	44
Slika 21. Arhitektura praktičnog primjera .....	49

Slika 22. ERA model baze podataka servisa za upravljanje narudžbama .....	57
Slika 23. Početni zaslon klijentske aplikacije .....	59
Slika 24. Forma za registraciju novih korisnika .....	60
Slika 25. Forma za prijavu .....	61
Slika 26. Prikaz košarice proizvoda .....	62
Slika 27. Forma za unos adrese za dostavu .....	63
Slika 28. Forma za unos detalja o plaćanju .....	63
Slika 29. Forma za finalni pregled detalja o narudžbi te potvrdu njenog kreiranja .....	64

## Popis programskih kôdova

Programski kôd 1. Primjer postavki za povezivanje servisa s MySQL bazom podataka .....	51
Programski kôd 2. Java klasa entiteta korisnika .....	51
Programski kôd 3. Sučelje repozitorija za entitet korisnika .....	52
Programski kôd 4. REST kontroler servisa za upravljanje korisnicima .....	52
Programski kôd 5. Klasa entiteta proizvoda .....	53
Programski kôd 6. REST kontroler servisa za upravljanje proizvodima .....	54
Programski kôd 7. Cart i CartItem Java klase .....	55
Programski kôd 8. REST kontroler servisa za upravljanje košaricama .....	56
Programski kôd 9. Atributi klase koja predstavlja narudžbu .....	57
Programski kôd 10. REST kontroler servisa za upravljanje narudžbama .....	58
Programski kôd 11. Primjer korištenja Javascript Fetch API-ja .....	60
Programski kôd 12. Programski kôd forme za prijavu .....	61

## Prilog 1 – Izvorni kôd i upute za pokretanje praktičnog primjera

Programski kôd i resursi potrebni za pokretanje praktičnog primjera nalaze se na javnom Git repozitoriju dostupnom na poveznici <https://github.com/josip-cesar/master-thesis>.

Za pokretanje u lokalnom okruženju potrebno je imati instalirane slijedeće alate: **Java JDK 8**, **Docker** i **Docker Compose**, **npm** upravitelj paketima koji dolazi kao dio instalacije Node.js platforme te **Spring Tool Suite (STS)** ili **Eclipse** razvojno okruženje s instaliranim **Spring Tools** paketom. Nakon instalacije potrebnih alata koraci za pokretanje praktičnog primjera su slijedeći:

1. Treba izvršiti preuzimanje (download) Git repozitorija na lokalno računalo.
2. Unutar mape **implementation/docker** treba otvoriti komandnu liniju te izvršiti naredbu `docker-compose build`, a potom naredbu `docker-compose up -d`. Time će se u pozadini pokrenuti tri Docker kontejnera s potrebnim bazama podataka, te će se tom prilikom automatski izvršiti skripta kojom će se u odgovarajuću MongoDB bazu podataka dodati i popuniti kolekcija proizvoda.
3. Unutar mape **implementation/web-app** nalaze se četiri dodatne mape koje predstavljaju četiri Spring Boot projekta unutar kojih se nalaze implementacije pojedinih web servisa. Za brzo pokretanje ovih servisa potrebno je najprije otvoriti svaki od ovih projekata unutar STS razvojnog okruženja te ih pokrenuti kao standardne Java aplikacije.
4. Naposljetku, klijentska aplikacija pokreće se tako da se unutar mape **implementation/demo-web-store-ui** otvori komandna linija te se potom najprije izvrši naredba `npm install` kako bi se instalirali svi potrebni npm moduli, a potom naredba `npm start` kojom će klijentska web aplikacija postati dostupna na web adresi <http://localhost:3000>.