SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Pušić

INTEGRACIJA HETEROGENIH BAZA PODATAKA NA PRIMJERU KOMPLEKSNE REAL-TIME APLIKACIJE

DIPLOMSKI RAD

Varaždin, 2014.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Pušić

Matični broj: 41752/12–R

Studij: Baze podataka i baze znanja

INTEGRACIJA HETEROGENIH BAZA PODATAKA NA PRIMJERU KOMPLEKSNE REAL-TIME APLIKACIJE

DIPLOMSKI RAD

Mentor:

Doc.dr.sc. Markus Schatten

Varaždin, rujan 2014.

Sadržaj

1. Uvod

Danas veliki broj aplikacija uvelike se oslanja na pohranu podataka u baze podataka. Te aplikacije mogu biti web aplikacije, mobilne, aplikacije za stolna računala i slično. S vremenom ukoliko broj korisnika našeg sustava raste, imamo veće opterećenje na bazi podataka i počinjemo razmišljati o rješenjima kako skalirati sustav.

Sustav je brz koliko je brz njegov najsporiji dio. Taj najsporiji dio je često sama baza podataka. U mnogo slučajeva kad kažemo baza podataka misli se striktno na relacijske baze podataka. Iako su prisutne već nekoliko desetljeća, jako su pouzdane i postoji velik broj aplikacija koje ih podržavaju, ipak se ne nose dobro s nekim od današnjih zahtjeva, kao što su performanse, skaliranje, distribuiranje podataka na više čvorova i slično.

U posljednje vrijeme su se počeli pojavljivati NoSQL[1] sustavi koji su inspirirani nekim od nedostataka koje imaju relacijske baze podataka i nastoje riješiti neke od glavnih nedostataka relacijskih baza podataka i neki od njih se nastoje posebno specijalizirati za određene aplikacijske domene. Iako donose neke prednosti, također donose i određene nedostatke sa sobom.

Vidjeti ćemo u nastavku da neke od nedostataka koje donose NoSQL baze podataka nismo u mogućnosti prihvatiti u svim domenama. Tako da niti jedne niti druge nisu savršen izbor, nego moramo u skladu s trenutnim zahtjevima aplikacije odabrati onu tehnologiju koja nam više odgovara. Ili ipak možemo dobiti najbolje od oba svijeta?

Ideja ovog diplomskog rada je razviti aplikaciju koja koristi više baza podataka (relacijskih i NoSQL) i na praktičnom primjeru istražiti mogućnosti ovog koncepta. Integracija heterogenih baza podataka zahtjeva pomno promišljanje o tome je li će naš sustav imati beneficije od dodatnog napora koji se mora uložiti prilikom njegovog razvijanja i održavanja. Neke od najpoznatijih kompanija na svijetu (Google, Amazon, Ebay) koriste ovaj pristup, zato jer imaju dovoljno resursa da si to mogu priuštiti, a i zato jer vjerojatno bez ovakvog pristupa ne bi bile u mogućnosti skalirati sustav i prilagođavati ga sve većih zahtjevima korisnika. Zasigurno svi sustavi neće imati beneficije od ovakvog pristupa, jer jednostavno broj korisnika i kompleksnost same aplikacije nije dorasla beneficijama koje pruža ovaj pristup, te u mnogo slučajeva može samo nepotrebno povećati kompleksnost aplikacije i otežati održavanje sustava.

[1] NoSQL – Not only SQL, No SQL. Naziv za sutave baza podataka koji se ne temelje na relacijskom modelu i ne koriste SQL kao jezik za manipuliranje podacima

U nastavku ćemo predstaviti različite sustave baza podataka, navesti njihove prednosti i nedostatke, te pokazati na koji način možemo sklopiti više različitih tehnologija i ideja u jednu cjelinu, koja funkcionira na jednom praktičnom primjeru. Također ćemo pokazati aplikaciju koja je razvijena u sklopu ovog rada i na koji način je ona integrirala više baza podataka.

2. Relacijski model podataka

Relacijski model baza podataka je danas najrasprostranjeniji i mnoge velike firme su mu dale svoje povjerenje. Prema (Ullman, 1994) ovaj model podataka je temeljen na ideji organiziranja podataka u dvodimenzionalne tablice koje nazivamo relacije.

2.1. Osnovni pojmovi

U ovom poglavlju ćemo predstaviti neke osnovne pojmove koji su karakteristični za relacijske baze podataka.

- **Usmjereni graf.** Graf kojem su svi rubovi usmjereni zovemo usmjereni graf ili digraf (Goodrich & Tamassia, 2005).
- **Podatkovni graf.** Digraf $G_D = (V, B)$ čiji su krajevi označeni (obilježeni s $o \sim b, b \in B$) i bridovi su podaci (mogu biti atomarni ili složeni) zovemo podatkovni graf (Schatten & Ivković, 2012).
- **Relacijska shema**. Neka je *R* skup atributa i *F* skup funkcijskih zavisnosti nad *R*. Par *(R, F)* zovemo relacijskom shemom (Maleković, 2012).
- **Relacija**. Relaciju možemo definirati kao konačan skup slogova nad *R.*
- **Entitet.** Entitet je objekt čija je reprezentacija spremljena u bazi podataka.
- **Veza.** U relacijskom modelu ovaj pojam se odnosi interakciju između entiteta u bazi podataka.
- **Atribut.** Svojstva entiteta i veza između njih predstavljamo atributima. Svaki atribut *A* ima pridruženi skup vrijednosti, koji možemo predstaviti s $Dom(A)$, odnosno domena atributa *A* (Simovici, 2011).
- **Ključ.** Recimo da imamo $(R, F), K \subseteq R$. *K* je ključ ako je *K* jedinstveni identifikator ($F \vdash K \rightarrow R$), te ne postoji neki podskup *K1* od *K* za koji možemo reći da je ključ ($K1 \subset K : \vdash K1 \rightarrow R$) (Maleković, 2012).

2.2. SUBP

Sustav upravljanja bazama podataka je agregacija podataka, hardwera, sofrwera i korisnika koji pomažu organizacijama pri upravljanju podacima (Simovici, 2011). Ovakvi sustavi su sposobni upravljati velikim količinama podataka i u mogućnosti su pružiti istovremeni pristup bazama podataka od strane više korisnika. Kada više korisnika istovremeno pristupa bazi podataka, ovakvi sustavi su u stanju osigurati konzistentnost

podataka. Korisnici su u interkaciji s bazom podataka putem upitnih jezika. Upitni jezici imaju dva glavna zadatka (Simovici, 2011):

- Definirati strukture podataka
- Omogućiti brzo modificiranje i dohvaćanje podataka

Neki od glavnih dijelova sustava za upravljanje bazama podataka su (Simovici, 2011):

- **Query processor.** Komponenta sustava koja pretvara korisničke upite u instrukcije koje sustav za upravljanje bazama podataka može efikasno procesirati.
- **Memory manager**. Ova komponenta dohvaća podatke iz baze podataka kao odgovor na upite koji su procesirani u prethodno opisanoj komponenti sustava.
- **Transaction manager**. Komponenta sustava koja osigurava da izvršavanje više transakcija u bazi podataka zadovoljava ACID svojstva, te također posjeduje mehanizme za oporavak od neočekivanih ispada pri radu sustava.

2.3. SQL

Ukoliko koristimo relacijski model, sigurno ćemo biti upoznati sa SQL jezikom, koji je *de-facto standard*[2] korišten za manipuliranje podacima u bazi podataka. Prema (Stephens, 2002) uz pomoć SQL jezika smo u mogućnosti sljedeće:

- Modificiranje strukture baze podataka
- Promjena sigurnosnih postavki sustava
- Upravljanje korisničkim dozvolama
- Postavljanje upita na bazi podataka
- Manipuliranje podacima

SQL jezik se sastoji od sljedećih komponenti (Brooklyn, 2014):

- **DDL (Data Definition Language).** Standardni podskup SQL jezika koji se koristi pri definiciji tablica i meta podataka baze podataka. Neke od naredbi koje uključuje ovaj skup su CREATE DATABASE, CREATE TABLE, DROP TABLE, ALTER TABLE.
- **DML (Data Manipulation Language).** Standardni podskup SQL jezika koji se koristi pri manipulaciji podacima u bazi podataka. Naredbe koje uključuje su INSERT, SELECT, UPDATE i DELETE.
- **DCL (Data Control Language).** Standardni podskup SQL jezika koji uključuje naredbe za upravljanje sigurnošću na razini baze podataka. Uključuje naredbe GRANT i REVOKE.

[2] De-facto standard – Konvencija, produkt ili sustav koji je opće prihvaćen kao dominantan

2.4. ACID

Termin ACID predstavlja poželjna svojstva za transakcije u bazi podataka. Transakcije su podržane u svim poznatijim sustavima za upravljanje relacijskim bazama podataka (npr. MySQL, PostgreSQL, Oracle, MS SQL Server), no one nisu nužno definirane nad relacijskim bazama podataka.

- **Atomarnost (engl. Atomicity).** Transakcija mora pratiti pravilo: „Sve ili ništa“. Svaka transakcija treba biti atomarna. Ukoliko jedan dio transakcije pođe po zlu, sve prošle operacije moraju biti opozvane, sve buduće moraju biti prekinute i baza podataka mora biti u stanju u kojem je bila prije početka transakcije.
- **Konzistentnost (engl. Consistency).** Svojstvo koje govori da samo valjani podaci mogu biti zapisani u bazu podataka. Ukoliko izvršena transakcija može naštetiti konzistentnosti baze podataka, transakcija se mora prekinuti i baza podataka vratiti u konzistentno stanje. Ukoliko je transakcija valjana, ona uvijek mora voditi bazu podataka iz jednog konzistentnog stanja u drugo konzistentno stanje.
- **Izolacija (engl. Isolation)**. Svojstvo koje zahtjeva da transakcije koje se izvršavaju u isto vrijeme nemaju utjecaja jedna na drugu. Npr. ukoliko imamo dvije transakcije koje izvršavaju neke operacije nad bazom podataka u isto vrijeme, sustav mora osigurati da transakcije nemaju utjecaja jedna na drugu, jer inače može doći do nepoželjnih nekonzistentnih stanja.
- **Trajnost (engl. Durability).** Svojstvo koje traži da svaka uspješno završena transakcija u bazi podataka mora biti trajna, odnosno da sve promjene nad podacima moraju ostati i u slučaju neočekivanih ispada.

3. NoSQL modeli podataka

U novije vrijeme imamo priliku vidjeti razvoj novih vrsta baza podataka. Za razliku od relacijskog modela, ove baze podataka se temelje na nekim drugim modelima podataka. Neki *NoSQL* prevode kao *no SQL*, dok neki prevode s *not only SQL*. U svakom slučaju oba prijevoda govore da je arhitektura ovih sustava drukčija od onih koji se temelje na relacijskom modelu.

Prema (Weber) NoSQL baze podataka su to distribuirane, horizontalno skalabilne baze podataka otvorenog koda koje se ne temelje na relacijskom modelu podataka.

Postavlja se pitanje u kojim situacijama možemo izvući korist od ovakvih sustava?

Prema (Sadalage & Fowler, 2012), ovdje ćemo navesti dva vrlo dobra razloga, mada ima ih još dosta:

- **Produktivnost.** Mnogo vremena pri razvoju aplikacija se potroši na prilagođavanje relacijskog modela potrebama aplikacije. Kod ovakvih sustava u mnogo slučajeva modeli su mnogo fleksibilniji i omogućavaju skraćivanje poslova koji se tiču planiranja samog modela podataka, jer sami po sebi nude fleksibilnost.
- **Skaliranje.** Relacijske baze podataka su razvijene u vremenu kada nismo imali naglašenu potrebu za obradom velike količine podataka koju imamo danas. Stoga sustavi nisu primarno dizajnirani za horizontalnu skalabilnost, dok su mnogi NoSQL sustavi dizajnirani upravo za potrebe obrade velikih količina podataka velikim brzinama i za horizontalnu skalabilnost. Kod većine NoSQL sustava horizontalna skalabilnost je nešto sasvim normalno, jednostavno za kofigurirati i održavati.

Ime NoSQL prvi puta se pojavilo krajem devedesetih godina prošlog stoljeća, kao ime baze podataka *Strozzi* NoSQL čiji je razvoj vođen od Carla Strozzi-a. Ime je dobila zbog činjenice da ne koristi SQL jezik, nego koristi *shell skripte*[3] i standardne UNIX *pipelines*[4] za postavljanje upita nad podacima.

Google i Amazon su dvije velike kompanije koje su među prvima osjetile potrebu za alternativnim bazama podataka i obje su kompanije predstavile članke koje pišu o takvim sustavima. To su bili *BigTable* od Google-a i *Dynamo* od Amazon-a.

Ime NoSQL kakvo nam je danas poznato dolazi iz 2009 godine, kao jedan od produkata konferencije koja se održavala u San Franciscu.

[3] Shell skripte – Tekstualne datoteke sa naredbama koje omogućuju različite operacije u UNIX sustavima
[4] Pipelines – Mehanizam u UNIX sustavima kojim možemo preusmjeriti izlaz iz jednog programa na ulaz u drugi program

Vidjeli smo da relacijske baze podataka podržavaju ACID svojstva, dok za ove baze podataka često se vežu BASE svojstva. Pogledajmo njihovu definiciju prema (Sadalage & Fowler, 2012):

- **Basically Available.** Garantirana dostupnost.
- **Soft state.** Stanje sustava se može promijeniti, čak i u slučaju 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.

Osim BASE svojstava, za NoSQL baze podataka često se veže i CAP teorem koji govori da mogu samo dva od sljedeća tri aspekta biti garantirana u isto vrijeme u distribuiranom sustavu (Weber):

- **Consistency**. Podaci su uvijek isti na svakom poslužitelju na kojem su podaci replicirani.
- **Availability.** Podaci uvijek moraju biti dostupni.
- **Partition Tolerance.** Baza podataka radi normalno i u slučaju ispada u mreži ili računalu.

Trenutno postoji nekoliko NoSQL modela podataka koji su široko rasprostranjeni. To su:

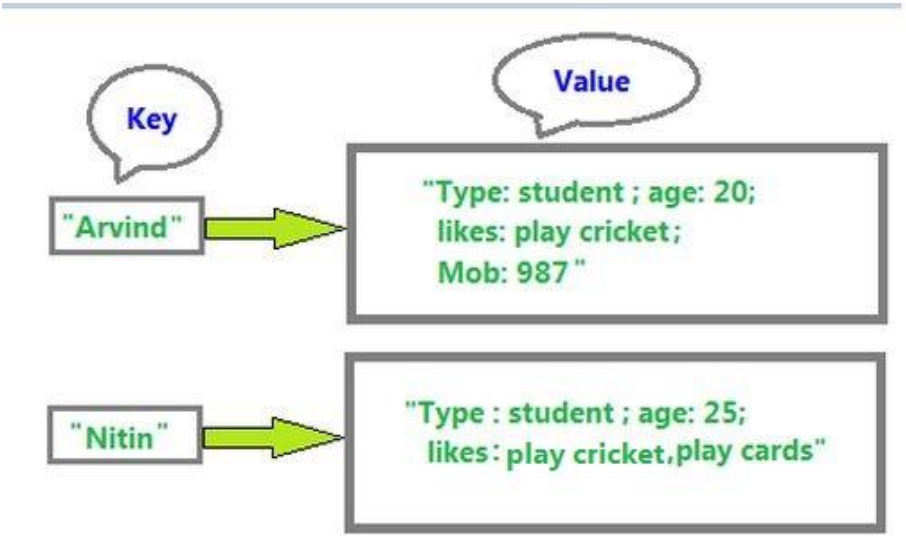
- ***Key value*** model
- ***Document*** model
- ***Column familly*** model
- ***Graph*** model

Svaki od njih ćemo posebno objasniti.

3.1. Key Value model

Prema (Sadalage & Fowler, 2012) ovaj model podataka je zapravo jednostavna hash tablica[5], koja se primarno koristi kada pristupamo podacima iz baze podataka isključivo preko nekog ključa. Da još malo pojednostavimo, ovaj model možemo gledati kao jednu tablicu u relacijskom modelu, koja ima dvije kolone. To su identifikator i vrijednost, gdje spremamo neki stvarni podatak.

[5] Hash tablica – Struktura koja koristi hash funkciju za preslikavanje ključeva u vrijednosti



Slika 1. Primjer Key Value modela. (Izvor slike: http://www.dbtalks.com/UploadFile/e6aced/contestant-of-nosql/Images/key-value.jpg)

Baze podataka koje se temelje na ovom modelu iz perspektive programskog API [6]sučelja su najjednostavnije. Klijent ima tri osnovne operacije:

- Dohvaćanje vrijednosti na određenom ključu
- Spremanje vrijednosti na određeni ključ
- Brisanje vrijednosti koja se nalazi na određenom ključu

Vrijednost koja se sprema u ovakve baze podataka je BLOB[7] i sama baza podataka nije zainteresirana za sadržaj koji se sprema, drugim riječima baza podataka ne radi nikakve dodatne provjere podataka, kao što su validacija i slično.

S obzirom da ovakve baze podataka koriste jedinstveni identifikator za dohvaćanje vrijednosti, ovakvi sustavi imaju iznimno visoke performanse pri čitanju podataka.

Neki od poznatijih sustava temeljenih za ovom modelu podataka su *Riak, Redis, Memcached HamsterDB, Amazon DynamoDB* i *Project Voldemort* odnosno *open source*[8] implementacija *Apache DynamoDB-a*.

Uz spomenute jednostavne operacije nad podacima, ovakvi sustavi pružaju i naprednije, npr. *Redis* baza podataka pruža rad s listama i skupovima.

3.1.1. Kada koristiti?

Navedimo nekoliko slučajeva kada su ove baze podataka izvrstan izbor:

- **Korisničke sesije**. Jedan od vrlo čestih slučajeva korištenja ovakvih baza podataka je za spremanje podataka o korisničkim sesijama. Ovi podaci će biti zahtijevani

[6] API – Application Program Interface, odnosno set metoda i atributa preko kojih možemo programski pristupati nekom sustavu.
[7] BLOB – Binary large object, odnosno binarni podaci koji su spremljeni u atribut u bazi podataka.
[8] Open source – Aplikacije čiji je izvorni kod dostupan i slobodan za modificiranje i djeljenje

prilikom svakog novog zahtjeva na poslužitelj od korisnika koji pristupa nekom dijelu sustava koji zahtjeva autentikaciju, stoga je jako bitno da pristup podacima sesije bude brz. Kao što smo već rekli spremamo podatke oblika ključ i vrijednost. Ključ nam može biti neki jedinstveni identifikator korisničke sesije koji možemo spremati u kolačić korisnika, dok nam vrijednost mogu biti podaci o korisniku.

- **Postavke**. Znamo da su postavke zapravo nizovi ključeva i vrijednosti nekih podataka, stoga su i ovi podaci idealni za spremanje u ovakve baze podataka.
- **Košarica**. Kod sustava koji implementiraju korisničke košarice, znamo da je svaka košarica vezana uz nekog korisnika. To može biti korisnički ID zapisan npr. u relacijskoj bazi podataka, no s obzirom da je jedinstven, može se koristiti kao ključ pri spremanju i dohvaćanju podataka iz sustava temeljenih na *key value* modelu podataka.

3.1.2. Kada ne koristiti?

Neki od slučajeva kada ovakav model nije prihvatljiv su:

- Kada neki skup podataka ne možemo jedinstveno identificirati nekim ključem.
- Kada imamo potrebu pretraživati podatke po više kriterija, a ne samo po primarnom ključu.
- Kada imamo potrebu za vezama između podataka
- Kada želimo raditi operacije nad više ključeva. Naime ove baze podataka su napravljene na način da omogućuju operacije nad jednim ključem u određenom trenutku.

3.2. Document model

Kao što su relacije glavni koncept kod relacijskog modela, tako su i dokumenti glavni koncept kod dokument modela podataka. Ove baze podataka spremaju i omogućuju pretraživanje XML[9], JSON[10], BSON[11] i ostalih formata dokumenata.

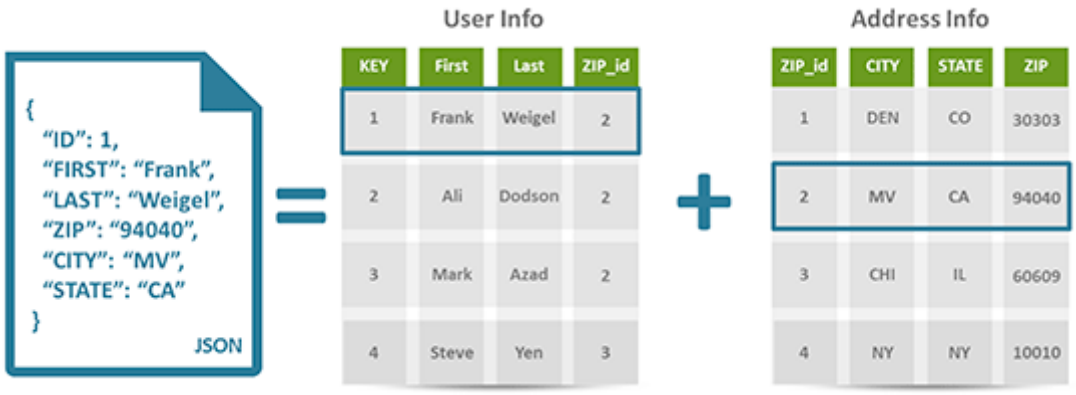
MongoDb	SQL baza podataka
Dokument	Red
Kolekcija	Tablica
_id	Primarni ključ

[9] *XML (EXtensible Markup Language)* – Jezik za označavanje podataka čija je ideja da bude čitljiv i ljudima i računalima
[10] *JSON (JavaScript Object Notation)* – Alternativa XML-u čija je primarna namjena razmjena podataka
[11] *BSON (Binary JSON)* – Binarno enkodirana forma JSON zapis

DBRef	Vanjski ključ
Agregacija	1:N relacija
DBRefs polje	M:N relacija

Na ovoj tablici vidimo usporedbu glavnih koncepata dokument modela i relacijskog modela. Vidimo da je tablica zapravo u ovom modelu kolekcija, jedan red je dokument, dok je primarni ključ zapravo *_id* atribut dokumenta.

Također jedna od karakteristika ovih baza podataka je da podaci nisu normalizirani, pa je normalno vidjeti situaciju u bazu podataka kao na sljedećoj slici:



Slika 2. Usporedba dokument zapisa i relacijskog zapisa. (Izvor: http://www.couchbase.com/sites/default/files/uploads/all/images/WhyNoSQL/Figure5a.png)

Ovdje vidimo da jedan dokument zapravo sadrži podatke iz dvije povezane tablice. Ovo je prihvatljivo za ovakav model podataka, jer struktura dokumenata u kolekciji nije definirana i samim time je fleksibilna, te možemo dodavati ili izostavljati atribute po potrebi.

Neki od poznatijih predstavnika ovakve vrste baza podataka su *MongoDb*, *CouchDB*, *OrientDB*, *RavenDB*, *eXistDB*.

3.2.1. Kada koristiti?

Slučajevi u kojima se ovakve baze podataka pokazuju kao dobar izbor su:

- **Log zapisi**[12]**.** Svaka aplikacija (ili većina njih) ima nekakve log zapise, koji se generiraju pri radu same aplikacije. Struktura log zapisa se može mijenjati kroz vrijeme i zavisna je od aplikacije. Ova činjenica nam govori da bi ovakav model podataka bio dobar u ovom slučaju. Imamo brzinu i fleksibilnost modela podataka. Također za razliku od *key value* modela podataka ovdje možemo postavljati upite i

[12] Log zapisi – Informacije koje računalni program generira prilikom svog rada, a odnose se na aktivnosti računalnog programa

nad više atributa, odnosno možemo raditi sve kao i što smo navikli kod baza podataka koji koriste SQL upitni jezik.

- **CMS sustavi[13]. Blog sustavi[14].** Također zbog činjenice da ove baze podataka koriste fleksibilan model podataka, pogodne su za spremanje podataka kod ovakvih sustava.
- **E-commerce[15].** Ovakvi sustavi često zahtijevaju fleksibilan model podataka za proizvode i narudžbe, također uz mogućnost da mogu brzo i bezbolno mijenjati strukturu zapisa u bazi podataka.

3.2.2. Kada ne koristiti?

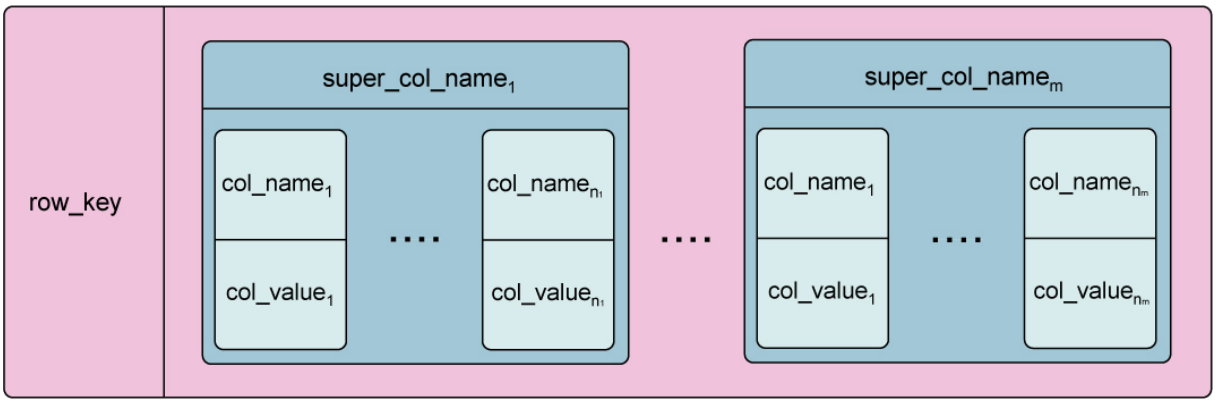
- **Kompleksne transakcije.** Ukoliko imamo potrebu za atomarnim operacijama u više dokumenata, onda ovaj model podataka neće biti najprihvatljiviji. Međutim npr. *RavenDB* podržava ovakve operacije, dok drugi spomenuti sustavi trenutno ne podržavaju. Znači spremanje podataka kao što su transakcije novca i slično, nisu baš najbolji posao za ovakve baze podataka.

3.3. Column family model

Jedna od prvih NoSQL baza podataka bila je Google *BigTable*. Iz ovog modela su se razvile dvije poznate baze podataka. To su *HBase* i *Cassansra*,

Ovaj model može podsjećati na relacijski, no postoje bitne razlike između ova dva modela. Jedna od razlika je način internog spremanja podataka. Dok relacijske baze podataka spremaju podatke po redovima, *column familly* baze podataka spremaju podatke po stupcima.
U ovim bazama podataka imamo pojmove c*olumn* i s*uper column* i oni zauzimaju 0B na disku ukoliko nema vrijednosti u njima. Jedino što definiramo kod stupaca je ime vrijednost i *timestamp*[16], znači nemamo potrebu definirati shemu. *Super column* može sadržavati druge c*olumn-e*, ali ne i druge s*uper column-e*.
Kod ovakvih baza podataka srećemo i pojam *column family*. To je struktura koja može držati neograničen broj redova. Ukoliko tražimo usporedbu s relacijskim modelom, onda ovo najbolje odgovara tablici. *Column family* također ima i ime, te se sastoji od strukture mape s ključem i vrijednosti. Ovdje je vrijednost zapravo ponovno struktura mape koja sadrži spomenute *column* zapise.

[13] CMS sustavi – Računalni sustav koji omogućava upravljanje sadržajem web stranice
[14] Blog sustavi – Oblik objavljivanja informacija na internetu
[15] E-commerce – Elektronička trgovina
[16] Timestamp – Informacija o tome kada je se dogodila neka operacija, neki događaj (engl. event) u sustavu



Slika 3. Format zapisa u column familly bazama podataka. (Izvor: http://www.sinbadsoft.com/wp-content/uploads/2014/04/super_column_family.png)

Keyspace je zapravo kontejner za više *columnFamily* zapisa. Također je bitno naglasiti da ne postoje veze između *columnFamily* zapisa. Oni su potpuno nezavisni.

Kod ovakvih baza podataka svaki put kada unesemo neki podatak, podaci će ostati sortirani. Ova činjenica ide u prilog krajnjim performansama sustava.

3.3.1. Kada koristiti?

- **Brzina.** Baze podataka s ovakvim modelom su jako brze zbog modela podataka kojeg podržavaju. Brzine se najviše očituju kada možemo pretraživati podatke po ključevima koji su kao što smo već rekli sortirani.
- **Skalabilnost.** Ukoliko želimo da se naš sustav dobro ponaša i s povećanjem broja korisnika i opterećenja sustava, ovakav tip baze podataka bi vjerojatno bio dobar izbor.

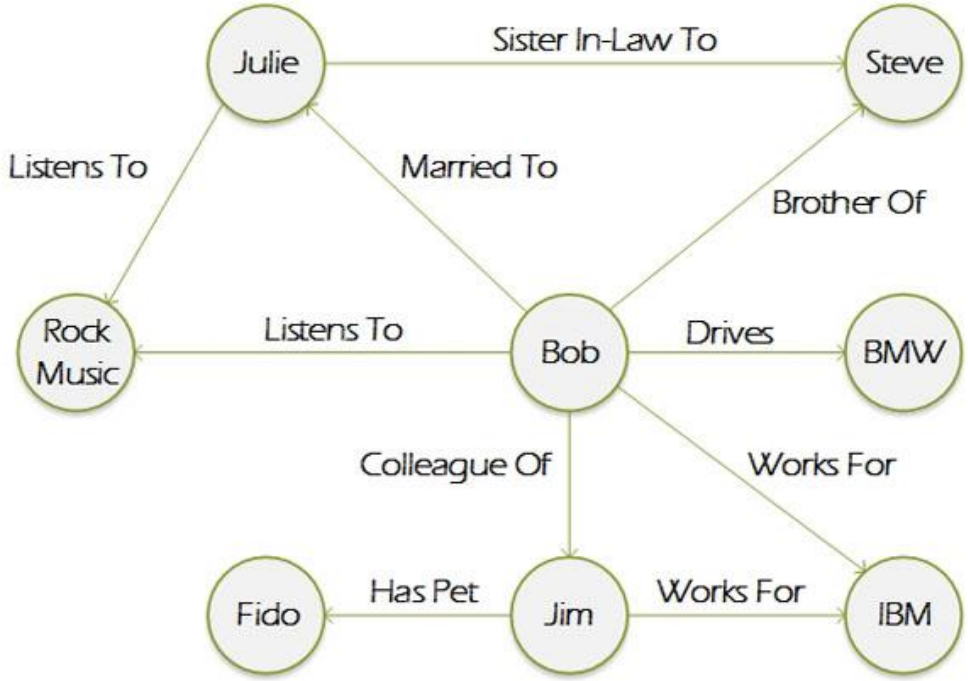
3.3.2. Kada ne koristiti

- **Pohranjene procedure, ograničenja, okidači.** Ovakve baze podataka nisu dizajnirane za spomenute značajke.
- **Postavljanje složenih upita.** Pretpostavka ovakvih sustava je da će sadržavati milione zapisa, i stoga postavljanje upita nad bazom podataka nije bezazleno. Iz ovog razloga sustavi koji implementiraju ovaj model podataka nude manju fleksibilnost pri kreiranju upita nad bazom podataka.
- **Transakcije.** Sustavi koji implementiraju ovaj model podataka ne posjeduju ACID svojstva, i stoga je moguće da baza podataka bude u nekonzistentnom stanju.

3.4. Graf model

Ove baze podataka su temeljene na modelu grafa, te su inspirirane lošom podrškom u relacijskim bazama podataka u području kompleksnih veza između podataka.

Kod ovog modela podataka imamo čvorove i veze između čvorova. Svaka veza i čvor može nositi određenu informaciju. Koristeći ovaj model podataka možemo postavljati upite koje su temeljene na vezama između čvorova. Ovo je jedna od važnih razlika između relacijskih i baza podataka temeljenih na modelu grafa. Iako relacijske baze podataka mogu implementirati vanjske ključeve, operacije pretraživanja korištenjem vanjskih ključeva su uglavnom skupe, dok kod baza podataka temeljenih na modelu grafa ove operacije su uobičajene i brze.



Slika 4. Model grafa. (Izvor: http://cdn.ttgtmedia.com/rms/editorial/Graph-database-sketch-580px.jpg)

Na slici vidimo osobe i njihove interese. Uz pomoć ovog modela možemo lako saznati interese neke osobe. Npr. možemo vrlo brzo saznati da osoba *Julie* sluša rok muziku.
Predstavnici ovakvih baza podataka su Neo4j, OrientDB, AllegroGraph te InfiniteGraph.
Neo4j je jedna od najpoznatijih. Implementira ovaj model podataka i dodatno se sastoji od Lucene sustava za indeksiranje podataka. Podržava ACID svojstva, te posjeduje REST [17]API sučelje.

[17] REST – Arhitektualni princip razvoja jezično neovisnih servisa koji su dostupni različitim klijentima putem razmjene HTTP poruka

3.4.1. Kada koristiti?

- **Pretraživanje podataka po vezama između njih.** Ukoliko smo u situaciji da podatke možemo prikazati ovim modelom, možemo dobiti velike prednosti u vidu performansi upita.
- **Socijalne mreže.** Jedna od najpoznatijih kompanija u svijetu koja koristi ovaj model podataka je Facebook. Uz pomoć ovog modela podataka u mogućnosti su prikazati podatke ljudi i odnosa između njih korištenjem grafa. Također, u mogućnosti su postavljati upite na koje će u kratkom vremenu dobiti odgovor.

3.4.2. Kada ne koristiti?

- **Ne postoje veze između podataka.** Ideja ovog modela podataka je da koristi veze između čvorova, odnosno podataka kako bi omogućila postavljanje kompleksnih upita nad tim podacima. Ukoliko nemamo veze između podataka, onda ovaj tip baza podataka vjerojatno nije dobar izbor.
- **Velike količine podataka.** Sustavi za upravljanje ovakvim baza podataka uglavnom imaju ograničenja broj čvorova i veza između njih. Ukoliko uzmemo za primjer sustav Neo4j, onda maksimalan broj čvorova i veza između njih je 2^{35}.

4. Relacijske vs NoSQL baze podataka

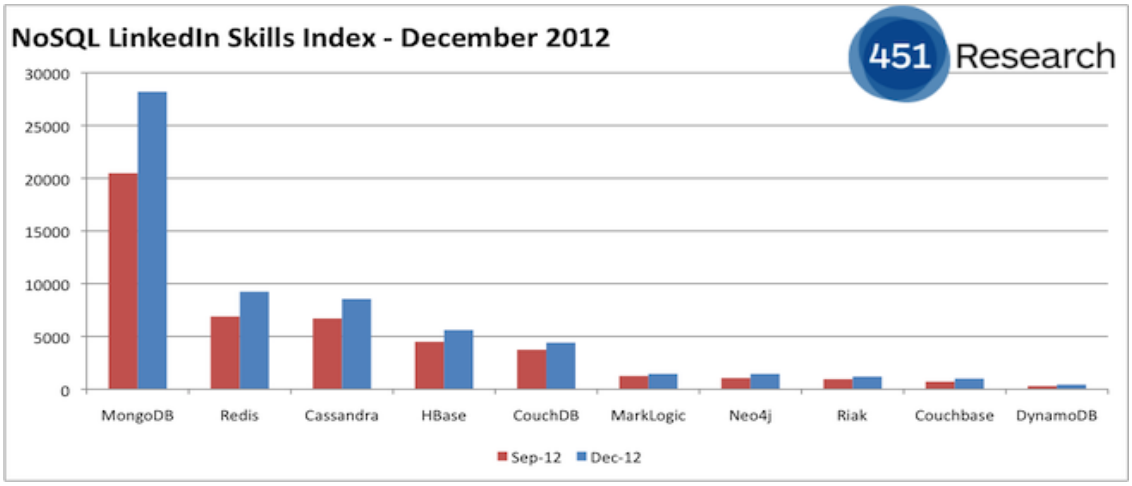
Vidjeli smo neke od glavnih principa na kojima se temelje danas raspoložive baze podataka, te da kod relacijskih baza imamo relacijski model, dok kod NoSQL baza podataka imamo nekoliko različitih modela podataka.

Kod relacijskih baza podataka shema podataka je fiksna, odnosno striktno definirana, dok kod NoSQL baza podataka shema je fleksibilna i podaci su uglavnom u strukturiranom ili polustrukturiranom stanju.

Relacijske baze podataka su vertikalno skalabilne, odnosno brzinu same baze podataka možemo povećati npr. ugradnjom boljeg procesora u poslužitelj, ugradnjom više memorije, ugradnjom SSD[18] diska i slično, dok su NoSQL baze podataka horizontalno skalabilne, odnosno performanse sustava povećamo dodavanjem novih poslužitelja.

Vidjeli smo da relacijske baze podataka koriste SQL kao jezik za različite akcije koje možemo raditi u takvim bazama podataka, dok NoSQL baze podataka koriste specifične jezike za operacije, koji su često prilagođeni domeni i modelu kojeg baza podataka podržava. Prednost SQL jezika je što se bolje nosi s kompleksnim upitima, dok različiti DSL[19] upitni jezici NoSQL baza podataka u nekim slučajevima nisu najbolje prilagođeni za kompleksne upite.

Relacijske baze podataka su na tržištu par desetljeća i mnogi ljudi su specijalizirani za ovakve baze podataka. Također, postoje mnogi alati koji podržavaju isključivo ovakve modele podataka. Za razliku od njih NoSQL baze podataka uglavnom imaju manje dostupnih stručnjaka i alata od relacijskih baza podataka.



Slika 5. Udjeli NoSQL baza podataka na kraju 2012 godine. (Izvor: http://blog.togo.io/wp-content/uploads/2012/12/NoSQL_Dec.png)

[18] SSD (Solid state drives) – Tvrdi diskovi za pohranu podataka koji posjeduju visoke performanse
[19] DSL (Domain-Specific Language) – Jezici koji su namjenjeni za rješavanje problema u specifičnoj domeni

No s vremenom to se mijenja i NoSQL baze podataka dobivaju sve više interesa od strane stručnjaka za baze podataka te od strane kompanija koje razvijaju alate za ovakve baze podataka.

Već smo vidjeli ACID svojstva na kojima se temelje relacijske baze podataka. Ova svojstva mogu biti ključna za mnoge *enterprise*[20] aplikacije i jedan veliki minus za NoSQL sustave koji ne podržavaju ova svojstva. Naime ukoliko imamo osjetljive podatke, često moramo imati osiguranje da će naši podaci biti spremljeni na pravilan način i da ih nećemo izgubiti. ACID svojstva su prednost, no nažalost donose dodatno opterećenje za sustav za upravljanje bazom podataka. Jedan od razloga za brzinu NoSQL baza podataka je i činjenica da mnogi od njih ne podržavaju ACID svojstva i samim tim dobivaju prednost u vidu performansi sustava.

I jedni i drugi sustavi imaju svoje prednosti i nedostatke, a odabir pravog gotovo uvijek će ovisiti o domeni u kojoj će se sustavi koristiti.

4.1. Integracija heterogenih baza podataka

Jedan od najizazovnijih koraka pri integraciji heterogenih izvora podataka je naći dovoljno dobro rješenje za problem interoperabilnosti heterogenih izvora podataka. Jedno od rješenja može biti kreiranje globalne sheme podataka umjesto više lokalnih shema (Ali, 2009). Globalna shema nam omogućuje uniformni pristup prema komponentama heterogenog sustava. Integracijom više sustava dolazimo do sustava koji se temelji na više lokalnih sustava, i prema (Ali, 2009) možemo ga nazvati *MDBS (multidatabase system)*, odnosno sustav koji će se krajnjim korisnicima predstavljati kao jedinstvena baza podataka, dok se zapravo na nižim slojevima sastoji od spomenutih lokalnih sustava.

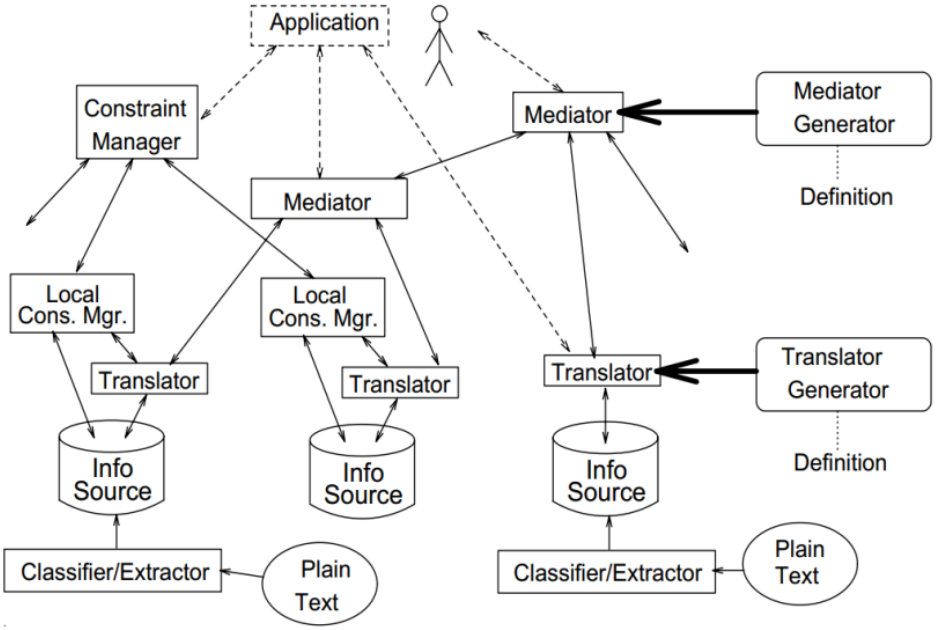
Prilikom integracije podataka može se pojaviti niz problema. Neki od njih su (Rajeswari & Varughese, 2009):

- Različite baze podataka koriste različite reprezentacije za isti podatak. To može značiti da npr. koriste različite tipove pri reprezentaciji podatka, ili npr. možemo imati situaciju da je jedan podatak predstavljen u više različitih mjera.
- U nekim slučajevima podatak u jednoj bazi podataka može biti atribut, dok u drugoj bazi podataka može biti npr. tablica.
- Za iste podatke možemo naći više različitih ograničenja.

Jedan od zanimljivijih projekata koji nastoje omogućiti integraciju podataka iz heterogenih izvora je i TSIMMIS projekt. Cilj projekta je da razvoje alate kojima možemo integrirati

[20] Enterprise – Aplikacije koje su razvijene da zadovolje potrebe organizacija

strukturirane i nestrukturirane izvore podataka. (Chawathe, Garcia-Molina, Hammer, Ireland, Papakonstantinou, & Widom). Jedna od najzanimljivijih stvari u ovom sustavu je njegova arhitektura.



Slika 6. Arhitektura TSIMMIS sustava (Izvor: http://ilpubs.stanford.edu:8090/66/1/1994-32.pdf)

Vidimo da iznad svakog izvora podataka imamo *translator*, koji logički prevodi podatke iz lokalnog izvora podataka u zajednički model podataka (engl. common information model). Za potrebe TSIMMIS projekta razvijen je zajednički model podataka koji se naziva *Object Exchange Model (OEM)*. Osnovna ideja ovog modela je da svi podaci imaju oznake kojima se opisuje njihovo značenje.

Jedna jako bitna komponenta ove arhitekture je i *Mediator*, odnosno sustav koji prilagođava informacije koje dolaze iz jednog ili više izvora. *Mediator* može imati znanje koje je potrebno za procesiranje specifične vrste podataka. *Mediator* također može primati upite od korisnika. Prije nego što pošalju odgovore na upite korisniku *mediator-i* mogu dodatno obraditi podatke, npr. mogu pretvoriti datume u neki standardni oblik.

Sljedeća komponenta sustava je upravljač ograničenjima. Ova komponenta ima zadaću osigurati određen nivo konzistentnosti podataka koji se nalaze u heterogenim izvorima. Zadnja komponenta koju ćemo opisati u ovom sustavu je *Classifier/Extractor*. Podaci su često nestrukturirani, npr. podaci mogu biti smješteni u tekstualnim datotekama, i svrha ove komponente je da dohvaća i raspoređuje podatke smještene u nestrukturiranim izvorima.

Integracija heterogenih izvora podataka je zahtjevno područje, gdje se često možemo susresti s različitim neočekivanim situacijama, i trenutno je još u razvoju.

5. Korištene tehnologije

U ovom dijelu ćemo vidjeti neke od tehnologija koje su korištene za programsku realizaciju ovog diplomskog rada. Opisati ćemo samo tehnologije koje su najuočljivije u arhitekturi ovog sustava.

5.1. MongoDb

Jedna od glavnih tehnologija na ovom projektu bila je baza podataka MongoDb. U usporedbi s relacijskim bazama podataka možemo reći da je ovaj tip baza podataka prilično nov. Kod relacijskog modela navikli smo imati niz tablica koje su povezane određenim vezama. Također nam je normalno da koristimo SQL za postavljanje upita nad bazom podataka. Kod MongoDb baze podataka stvari su malo drukčije.

Slika 7. Logo MongoDb sustava

MongoDb baza podataka se ne temelji na relacijskom modelu, nego na dokument modelu podataka.

5.1.1. Instalacija

MongoDb baza podataka dostupna je za više platformi. To su Windows, Linux, Mac OSX te Solaris.

Npr. ukoliko koristimo Linux operacijski sustav i Fedora distribuciju, možemo na vrlo jednostavan način instalirati MongoDb sustav. Fedora distribucija koristi *yum package manager*[21] i s obzirom na tu činjenicu možemo dodati novi izvor paketa:

```
[mongodb]
name=MongoDB
Repositorybaseurl=http://downloads-
distro.mongodb.org/repo/redhat/os/x86_64/
```

[21] Yum package manager – Program za instaliranje, ažuriranje i brisanje programa na određenim Linux operacijskim sustavima

```
gpgcheck=0
enabled=1
```

Nakon ovoga možemo s jednom naredbom instalirati sustav MongoDb:

```
sudo yum install mongodb-org
```

Nakon instalacije potrebno je pokrenuti MongoDb bazu podataka kao servis.

```
sudo service mongod start
```

Ovaj servis se može pokretati na zahtjev, ili možemo reći operacijskom sustavu da ga pokreće svaki put za nas prilikom podizanja sustava.

```
sudo chkconfig mongod on
```

Log podatke možemo naći u */var/log/mongo/mongod.log* datoteci.

Sličan način instalacije je i za ostale Linux distribucije.

Ukoliko koristimo Mac OSX ili Windows operacijski sustav, instalacija sustava se svodi na preuzimanje instalacijskih datoteka sa službenih stranica i pokretanje instalacijskog procesa sa standardnim načinom instalacije programa na ovim sustavima.

Slika 8. Instalacija sustava MongoDb na Windows OS

Kada smo instalirali sustav, potrebno je pokrenuti MongoDb poslužitelj sljedećom naredbom:

```
C:\Program Files\MongoDB\bin\mongod.exe
```

Kako bi pristupili konzolnom sučelju možemo koristiti sljedeću naredbu:

```
C:\mongodb\bin\mongo.exe
```

```
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
        http://docs.mongodb.org/
Questions? Try the support group
        http://groups.google.com/group/mongodb-user
>
>
> show dbs
admin  (empty)
local  0.078GB
>
```

Slika 9. Konzolno sučelje sustava MongoDb

Ovo su bile osnovne upute kako instalirati i pokrenuti ovaj sustav na Windows i Linux operacijskim sustavima. Pogledajmo sada način na koji možemo upravljati ovim sustavom.

5.1.2. Osnovne naredbe

Jedna od prvih stvari koju želimo napraviti prilikom rada s bazama podataka je kreirati bazu podataka. Kod MongoDb baze podataka to je vrlo jednostavno.

```
use mydb
```

Kako bismo bili sigurni da se nalazimo u pravoj bazi podataka, možemo iskoristiti sljedeću naredbu, koja će nam ispisati naziv baze podataka u kojoj se trenutno nalazimo:

```
> db
mydb
```

Način na koji smo kreirali bazu podataka s razlogom može biti malo čudan. Naime kod MongoDb sustava nije potrebno eksplicitno pisati naredbe za kreiranje baze podataka. Ukoliko baza podataka koju želimo koristiti ne postoji, MongoDb sustav će ju kreirati za nas, a ukoliko postoji, prebaciti će nas u postojeću bazu podataka.

MongoDb sustav posjeduje standardne *insert, update* i *delete* naredbe. Kao što smo već rekli, dokument model se sastoji od kolekcija, pa kreirajmo kolekciju korisnika.

```
> db.users.insert({name: 'Ivan'})
WriteResult({ "nInserted" : 1 })
```

Vidimo da ponovo nismo eksplicitno kreirali kolekciju, nego smo samo upisali podatak u nju. Isto kao u slučaju s kreiranjem baze podataka, ukoliko resurs ne postoji, MongoDb sustav će ga kreirati za nas, a ukoliko postoji, koristit će se postojeći resurs.

Kada smo unijeli podatak u kolekciju, možemo ga pronaći korištenjem naredbe *find*.

```
> db.users.find()
{ "_id" : ObjectId("53e60d398ce1886d236586b3"), "name" :
"Ivan" }
```

Vidimo *_id* atribut koji nismo ekspicitno dodali u naš dokument. Prema (Banker, 2012) ovaj atribut možemo gledati kao primarni ključ kolekcije.

Možemo također doznati broj dokumenata koji se nalaze u kolekciji.

```
> db.users.count()
1
```

Napomenuti ćemo da ova operacija i nije baš najefikasnija u ovom sustavu, stoga ju je potrebno izbjegavati ukoliko je to moguće.

Ukoliko želimo ažurirati neki dokument, možemo koristiti naredbu „update“.

```
> db.users.update({name: "Ivan"}, {$set: {name: "Ivan
updated"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1
})
```

Prvi argument funkcije *update* je upit kojim ćemo dohvatiti sve dokumente čiji sadržaj želimo ažurirati. Nakon toga drugim argumentom možemo reći MongoDb sustavu što želimo napraviti s tim dokumentima. U ovom slučaju smo ažurirali sve korisnike čije je ime *Ivan* na vrijednost *Ivan updated*.

Općenit potpis ove funkcije je sljedeći:

```
db.collection.update(
   <query>,
   <update>,
   {
     upsert: <boolean>,
     multi: <boolean>,
     writeConcern: <document>
   }
)
```

Vidimo da ova funkcija prima i treći argument kojim možemo proslijediti neke dodatne opcije prema kojima će ažuriranje biti izvršeno. Npr. opcijom *multi* govorimo sustavu je li želimo ažurirati jedan ili više dokumenata.

Ukoliko želimo brisati dokumente iz sustava možemo koristiti funkciju remove.

```
> db.users.remove({name: "Ivan"})
WriteResult({ "nRemoved" : 0 })
```

Ovime smo izbrisali sve dokumente iz sustava koji imaju vrijednost *Ivan* na atributu ime.

5.1.3. Postavljanje upita

Vidjeli smo primjer korištenja *find* funkcije koja služi za pretraživanje dokumenata. Funkciju smo vidjeli kako radi u najjednostavnijoj inačici, a sada ćemo vidjeti kako možemo koristiti neke naprednije tehnike za dohvaćanje željenih dokumenata.

Jezik za postavljanje upita ima JSON sintaksu i prilično je intuitivan. Funkcije koje ćemo sada pokazati se mogu koristiti i pri pronalaženju dokumenata koje ćemo brisati i ažurirati, stoga je jako bitno da pokažemo načine na koje možemo dohvaćati dokumente u ovom sustavu.

Jedna od prvih stvari koje ćemo pokazati je na koji način možemo implementirati straničenje korištenjem MongoDb sustava. U MongoDb sustavu postoje funkcije *skip* i *limit* koje će nam poslužiti upravo toj svrsi. (MongoDb, 2014)

```
> db.users.find({name: "Ivan"}).skip(5).limit(50)
```

Upit će nam prvo naći sve korisnike s imenom *Ivan*, preskočiti će prvih 5 i uzeti 50 preostalih rezultata.

Kod postavljanja upita vidjeli smo funkciju *find*, no postoji i funkcija *findOne* čijim korištenjem će nam sustav vratiti samo jedan dokument.

```
> db.users.findOne({name: "Ivan"})
```

5.1.4. Operatori usporedbe

Znamo da putem SQL jezika možemo postaviti upit tipa veće od, manje od i slično, a to isto možemo i kod MongoDb sustava uz pomoć operatora *$gt, $gte, $lt, $lte, $ne.*

```
db.users.find({age: {$gte:20, $lte: 30}})
```

Ovime ćemo dohvatiti sve korisnike čija je dob između 20 i 30 godina. Koristili smo operatore *$gte* i *$lte*, što znači veće ili jednako od i manje ili jednako od. Ukoliko bi izostavili slovo e, odnosno *equal*, onda bi dobili rezultate gdje su godine od 21 do 29.

```
db.users.find(name: {$ne: "Ivan"} }
```

Prethodni upit će nam vratiti sve korisnike čija vrijednost atributa *name* nije jednaka *Ivan*.

5.1.5. Skupovni operatori

MongoDb posjeduje operatore za rad sa skupovima. Prema (Banker, 2012) to su *$in, $all* i *$nin*[22]. Korištenjem operatora *$in* dobit ćemo sve dokumente koji imaju jednu od vrijednosti u nekom skupu.

```
db.users.find({_id: { $in:
[ObjectId("6a5b1476238d3b4dd5000048"),
ObjectId("6a5b1476238d3b4dd5000051"),
ObjectId("6a5b1476238d3b4dd5000057") ] } } )
```

Ovime ćemo dobiti dve korisnike čija je vrijednost atributa *_id* u jednoj od navedenih vrijednosti. Ukoliko bismo umjesto operatora *$in* iskoristili operator *$nin* dobili bismo sve korisnike čija vrijednost atributa *_id* nije jednaka niti jednoj od ponuđenih vrijednosti.

```
db.users.find(tags: { $all: ["programmers", "architects"] }
```

[22] Mongodb in action 82str.

Korištenjem operatora *$all* smo rekli da želimo sve korisnike koji imaju oba taga *programmers* i *architects*.

5.1.6. Logički veznici

MongoDb posjeduje logičke veznike i to *$not, $or, $nor, $and* te *$exists*.

Operatorom *$ne* možemo pronaći sve dokumente pomoću negiranja vrijednosti atributa, dok putem *$not* operatora možemo dobiti sve dokumente negiranjem rezultata drugog MongoDb upita ili nekog regularnog izraza.

```
db.users.find(name: {$not: /^iv/} }
```

Ovim upitom ćemo dobiti sve korisnike čija vrijednost atributa *name* ne započinje vrijednošću *iv*.

Upitom *$or* možemo dobiti sve dokumente koji vraćaju istinu za jedan od postavljenih uvjeta.

```
db.users.find({ $or: [{name: 'Ivan'}, 'lastname':
'Ivanovic'}] })
```

Upit će nam vratiti sve korisnike koji se zovu *Ivan* ili prezivaju *Ivanovic*. Ukoliko bismo operator *$or* zamijenili s operatorom $and, dobili bismo sve korisnike koji se zovu *Ivan* i prezivaju *Ivanovic*.

Korištenjem operatora *$exists* možemo dobiti sve dokumente koji sadržavaju ili ne sadržavaju neki atribut.

```
db.users.find( {lastname: { $exists: true } } )
```

Ovaj upit će nam vratiti sve korisnike koji imaju definiran atribut „lastname“.

5.1.7. Pod dokumenti

Rekli smo već da kod dokument modela imamo fleksibilnu strukturu i da možemo imati dokumente u dokumentu. Postavlja se pitanje kako pretraživati pod dokumente. Pogledajmo sljedeći primjer:

```
{
     _id: ObjectId("4c4b1476238d3b4dd5003981"),
     name: "Ivan",
     details: {
          age: 23,
          address: "NYC",
          sex: 'm'
     }
```

```
}
```

Pretpostavimo da imamo n ovakvih dokumenata, s različitim podacima. Kako bismo mogli uz pomoć MongoDb sustava postaviti upit koji će nam vratiti sve osobe muškog spola?

```
db.users.find({'details.sex': 'm'});
```

Vidimo da uz pomoć *točka* notacije možemo pristupiti atributima pod dokumenta i doći do željenih dokumenata.

5.1.8. Polja

Polja su jedno od velikih prednosti dokumentnog modela podataka. MongoDb sustav naravno podržava postavljanje upita nad ovakvim strukturama. Vidjeli smo već primjer korištenja *$in* operatora, koji nam može vratiti sve dokumente koji se nalaze u nekom skupu (polju) vrijednosti, a pogledajmo sada kako možemo postaviti upit koji će nam vratiti sve dokumente čija polja sadržavaju jednu specifičnu vrijednost.

Uzmimo sljedeći primjer:

```
{
     _id: ObjectId("4c4b1476238d3b4dd5003981"),
     name: "Ivan",
     tags: [„programmers“, „architects“]
}
```

Recimo da želimo pronaći sve korisnike programere:

```
db.users.find({tags: " programmers "})
```

Jedna od zgodnih stvari koju i sustav MongoDb posjeduje su indeksi i s njima možemo ubrzati odgovore na upite ukoliko indeksiramo polja po kojima često radimo upite.

Npr. ukoliko bismo htjeli postaviti indeks nad *tags* poljem, to bi mogli učiniti sljedećom komandom:

```
db.products.ensureIndex({tags: 1})
```

Ukoliko bismo imali dokument koji sadrži polje s drugim dokumentima, postavljanje upita bismo ponovo radili putem točka notacije.

Ovo su bili neki od osnovnih operatora i načina postavljanja upita. Postavljanje upita u sustavu MongoDb je široka tema, stoga nećemo ići u daljnje detalje.

Jedna od slabosti ovakvog tipa baza podataka je i prevelika fleksibilnost. Postoje

ODM[23] sustavi koji smanjuju ovu fleksibilnost definiranjem i validacijom dokumenata koji se zapisuju u kolekcije, no i dalje to nije osiguranje da će neko polje u nekom dokumentu biti određenog tipa, jer sama provjera se nalazi na aplikacijskom nivou, koji može imati pogreške, a ne nivou same baze podataka.

Sve u svemu danas se ovakve baze podataka jako dobro uklapaju u nove tehnologije s kojima se možemo susresti. Pošto je cijela sintaksa upitnog jezika u obliku JSON-a i sami dokumenti su zapravo JSON objekti, možemo lako zaključiti da je MongoDb dobar prijatelj s programskim jezikom Javascript.

5.2. PostgreSQL

PostgreSQL je o*pen source* baza podataka koja je široko rasprostranjena i korištena od velikih kompanija kao što su Apple, Red Hat, Cisco i drugi.

Slika 10. Logo PostgreSQL sustava

Za razliku od MongoDb baze podataka koja se temelji na dokumentnom modelu, ova baza podataka temelji se na relacijskom modelu. Prednost ovakvog tipa baze podataka je što možemo definirati jasna pravila koja moraju poštovati svi podaci koji se nalaze u bazi podataka i uz ovo osiguranje možemo biti sigurni da su podaci očekivanog tipa.
Također ova baza podataka je već dugi niz godina u kontinuiranom razvoju, te možemo biti prilično sigurni da će raditi očekivano u različitim uvjetima.

Još jedna od prednosti ove baze podataka je što implementira SQL jezik kao jezik za postavljanje upita nad podacima u bazi podataka. Uz ovo sve baza podataka ima ACID svojstva, zbog čega možemo biti prilično sigurni u valjanost i konzistentnost baze podataka u različitim situacijama.

[23] ODM (Object-Document Mapper) – Sustav koji pretvara dokumente iz baze podataka u objekte koje možemo koristiti putem nekog programskog jezika

Ova baza podataka je dostupna za različite platforme, što je još jedan od razloga zašto se široko koristi. Posjeduje sučelja prema različitim programskim jezicima, kao što su C/C++, Java, C#, Python, Javascript (Node.js) i drugi.

Baza podataka nosi sa sobom i određena ograničenja. U trenutnoj inačici PostgreSQL baze podataka postoji ograničenje na veličinu tablice i iznosi 32TB, na veličinu reda 1.6TB, te na veličinu pojedinačnog polja u bazi podataka 1TB. Ograničenje broja atributa koje može imati tablica kreće se od 250-1600, ovisno o tipovima atributa.

Baza podataka podržava nekoliko tipova indeksa. To su B-tree, Hash, GiST te GIN. Svaki od indeksa se temelji na drukčijem algoritmu koji se dobro ponaša kod određene vrste upita. Pretpostavljeni (engl. default) indeks je B-tree. Ova vrsta indeksa se dobro ponaša kod upita gdje imamo prepoznavanje uzoraka (engl. pattern matching), operatore jednakosti te upite kojima postavljamo nekakve raspone.

Hash indeksi se koriste uglavnom za jednostavne operacije usporedbe. GiST indeksi nisu zasebna vrsta indeksa, nego su zapravo infrastruktura unutar koje se različite strategije indeksiranja mogu implementirati. Gin indeksi se koriste u situacijama kada primjerice radimo s poljima. Kao i GiST Gin može podržavati različite korisnički definirane strategije indeksiranja.

5.2.1. Instalacija

Instalacija sustava je vrlo slična instalaciji MongoDb sustava. Ukoliko koristimo Fedora Linux distribuciju instalacija PostgresSQL poslužitelj se svodi na jednu naredbu.

```
sudo yum install postgresql-server postgresql-contrib
```

Nakon što je sustav instaliran možemo ga pokrenuti sljedećom naredbom.

```
sudo systemctl start postgresql
```

I to je to. Imamo instaliran sustav i pokrenut poslužitelj za bazu podataka. Zadnja stvar koju moramo napraviti je kofigurirati sustav. Srećom PostgreSQL dolazi s naredbom kojom možemo postaviti početne postavke sustava kao zadane.

```
sudo postgresql-setup initdb
```

Instalacija sustava na ostalim Linux distribucijama je gotovo identična, dok se instalacija za Windows okružje ponovno svodi na standardni windows instalacijski proces.

5.2.2. Kreiranje korisnika i baze podataka

Nakon što smo instalirali i pokrenuli ovaj sustav, vjerojatno ćemo imati potrebu kreirati korisnike i baze podataka na sustavu. Za razliku od MongoDb sustava gdje nismo morali eksplicitno kreirati resurse, u ovom sustavu to moramo.

```
CREATE USER someuser WITH PASSWORD 'somepwd';
```

Ovom naredbom smo kreirali korisnika *someuser* i dodjelili smo mu lozinku *somepwd* (PostgreSQL, 2014).

```
CREATE DATABASE someuser;
```

Sada smo kreirali i bazu podataka s imenom *someuser*. Zadnje što trebamo napraviti je dodijeliti prava korisniku nad ovom bazom.

```
GRANT ALL PRIVILEGES ON DATABASE someuser TO someuser;
```

Sada ukoliko smo prijavljeni u UNIX operacijski sustav kao *someuser* i u naredbenu liniju napišemo `psql` naredbu, automatski ćemo biti prijavljeni u PostgreSQL sustav kao *someuser* i biti ćemo prebačeni u bazu podataka *someuser*.
To možemo i eksplicitno zadati sljedećom naredbom.

```
psql –d databasename –U username
```

Ukoliko želimo promijeniti trenutnu bazu podataka iz PostgreSQL naredbenog sučelja, to možemo učiniti na sljedeći način.

```
\c databasename
```

U nastavku se nećemo baviti načinima kreiranja tablica, postavljanja upita putem SQL jezika i slično, nego ćemo pokazati neke od specifičnosti PostgreSQL sustava.

5.2.3. Konverzija vrijednosti

Kod programskih jezika imamo mogućnost pretvaranja podataka iz jednog tipa u drugi. Kod ovog sustava imamo također istu mogućnost.

```
SELECT CAST(birth as varchar(30)), CAST(weight as integer)
from users
```

U ovom slučaju pretvaramo kolonu *birth* koja je tipa *date* u tip *varchar*. Prema (Shaw, 2013) ova sintaksa može biti pomalo čudna pa su razvijatelji PostgreSQL sustava dodatno pojednostavili samu sintaksu kojom možemo pretvarati podatke.

```
SELECT birth::varchar(30), weight::integer FROM users;
```

Pretvorba podataka nije ograničena samo na postavljanje upita, nego možemo koristiti konverziju i prilikom unošenja podataka.

```
insert into users values (2, '2012-3-3'::date, 5.5::integer)
```

5.2.4. Polja

Polja nisu uobičajena pojava kod relacijskih baza podataka, no u ovom sustavu imamo mogućnost spremanja podataka i u ovakve strukture. Definirajmo sada jednu tablicu koja će sadržavati atribut koji je zapravo polje.

```
create table lotonumbers(
   name varchar(30),
   numbers integer[6]
);
```

Kada želimo unijeti podatke u ovakvu tablicu to možemo učiniti na sljedeći način:

```
insert into lotonumbers(name, numbers) values('peter', '{
1,2,3,4,5,6}');
```

Imamo također podršku i za postavljanje upita nad ovakvim strukturama.

```
select * from lotonumbers where numbers[1] = 1;
```

Ovim upitom ćemo dobiti sve redove iz tablice *lotonumbers* gdje prvi broj ima vrijednost 1.

5.2.5. Nasljeđivanje

Kod objektno orijentiranih programskih jezika smo navikli da možemo definirati neke osnovne strukture iz kojih možemo izvoditi složenije putem nasljeđivanja, a nešto slično možemo naći i u PostgreSQL sustavu. Način na koji ovo radi je jako jednostavno.

Recimo da smo kreirali tablicu korisnika sa standardnim atributima koje imaju korisnici i želimo kreirati tablicu administratora, gdje ove dvije tablice dijele sve atribute i tablica administratora ima još neke dodatne. Umjesto da ponovno pišemo SQL za kreiranje svih atributa, možemo učiniti sljedeće.

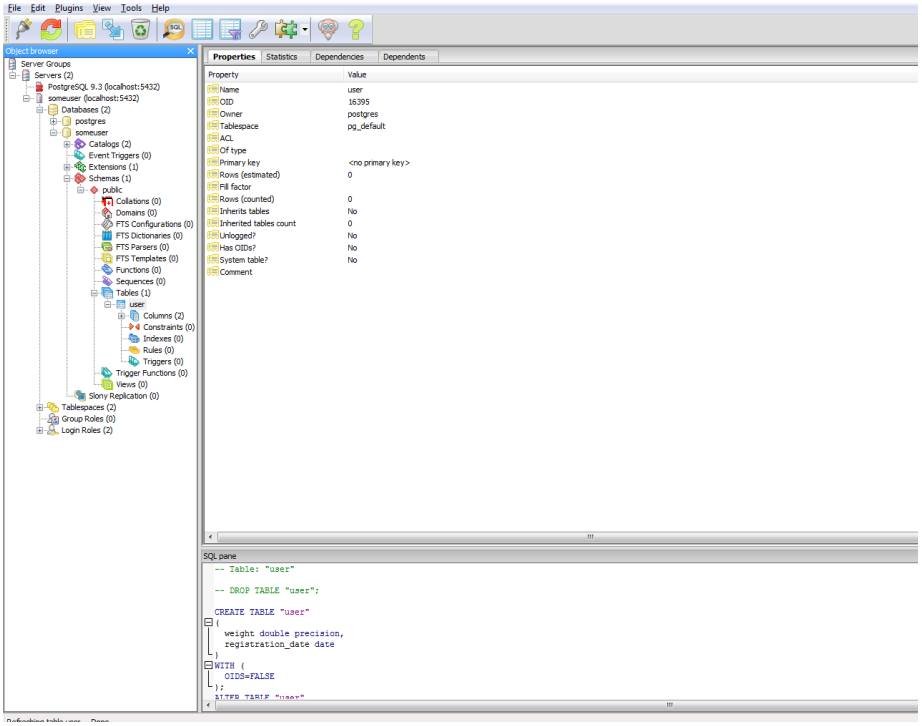
```
create table admins (
   active boolean
) inherits(users);
```

Ukoliko sada odaberemo neki redak iz tablice *admins*, vidjet ćemo da tablica ima i sve atribute iz tablice *users*.

5.2.6. pgAdmin

Kao što smo već rekli jedna od prednosti relacijskih sustava je i programska podrška. Razvijatelji aplikacija mnogo napora ulažu kako bi razvili alate kojima će administratorima i

dizajnerima relacijskih baza podataka olakšati posao. Jedan od alata koji dolazi uz PostgreSQL sustav je i pgAdmin alat.



Slika 11. Sučelje pgAdmin alata

Uz pomoć ovog alata možemo jednostavno kreirati tablice, unositi vrijednosti, kreirati baze podataka i ostale resurse u sustavu.

Napomenuti ćemo da je ovo jedan od osnovnih alata i da postoje mnogo moćniji alati od ovoga, no u svakom slučaju ovo je također jedna od prednosti ovog sustava, jer u usporedbi s MongoDb sustavom koji ne donosi grafičku programsku podršku, ovaj sustav to donosi.

5.3. Redis

Redis je također open source baza podataka koji se temelji na modelu drukčijem nego prethodno opisane baze podataka. Model podataka koji se koristi kod Redis baze podataka je spomenuti *key value* model.

Slika 12. Logo Redis sustava

Kao i za prošle baze podataka i za ovu bazu podataka postoje biblioteke za pristup mogućnostima baze podataka iz različitih programskih jezika.

Kada dizajniramo model za relacijsku bazu podataka razmišljamo o tablicama, indeksima, vezama između tablica i slično. Kod ovog modela baze podataka ideja je da identificiramo ključeve koje će identificirati objekte.

Ovakvi tipovi baza podataka odlikuju se iznimno velikim brzinama čitanja podataka, jer omogućavaju direktan pristup traženome podatku. Uzimajuć ovu činjenicu u obzir odlični su kandidati za spremanje nekih privremenih podataka, za spremanje korisničkih sesija i slično.

5.3.1. Instalacija

Ovaj sustav za razliku od prethodna dva ne podržava Windows okružje. Instalacija ovog sustava na Linux operacijski sustav je malo drukčija nego kod prethodna dva sustava. Kod ovog sustava biti će potrebno prevesti izvorni kod sustava.

Kako bi instalirali zadnju stabilnu inačicu ovog sustava možemo se poslužiti sljedećim naredbama. (Redis, 2014)

```
wget http://download.redis.io/redis-stable.tar.gz
tar xvzf redis-stable.tar.gz
cd redis-stable
make
```

Poslužitelj možemo pokrenuti putem naredbe *redis-server*.

Kako bi provjerili je li poslužitelj uspješno pokrenut možemo iskoristiti sljedeću naredbu.

```
redis-cli ping
PONG
```

Ovaj sustav posjeduje konzolno sučelje putem kojega možemo biti u interakciji sa samom bazom podataka. Konzolno sučelje možemo pokrenuti naredbom *redis-cli.*

Programsko sučelje ovakve baze podataka je prilično jednostavno. Osnovne dvije naredbe su GET i SET, putem kojih dohvaćamo i postavljamo vrijednosti ključeva u bazi podataka.

5.3.2. Tipovi podataka

String je najjednostavniji tip podataka u ovom sustavu. String vrijednost može biti bilo koja *byte array* vrijednost. Ukoliko bismo htjeli spremiti i dohvatiti vrijednosti ovog tipa, možemo se poslužiti sljedećim naredbama.

```
SET pages:about "about us"
GET pages:about
```

```
about us
```

Drugi tip podataka koji ovaj sustav podržava su Hash strukture. Umjesto da pristupamo jednoj vrijednosti na nekom ključu, možemo pristupati više vrijednosti na nekom ključu.

Recimo da želimo kreirati ovakvu strukturu koja će spremati podatke o korisnicima i da želimo dodati jednog korisnika u nju.

```
hset users ivan "user Ivan"
```

Sada možemo dohvatiti korisnika.

```
hget users ivan
```

Znači na ključu *users* u strukturi Hash dohvaćamo vrijednost na ključu *ivan*.

Treći tip podataka u ovom sustavu su liste. Ova struktura predstavlja niz vrijednosti koje se nalaze na jednom ključu. Podržane su standardne operacije nad poljima, kao što su dodavanje, brisanje i druge. Ukoliko želimo dodati podatke u neku listu možemo se poslužiti sljedećom naredbom.

```
lpush userslist "ivan"
```

Naredba *lpush* dodaje element na lijevu stranu liste, a ukoliko želimo dodati element na desnu stranu liste možemo se poslužiti naredbom *rpush*.

Ukoliko želimo dohvatiti jednu vrijednost na određenom indeksu u polju možemo iskoristiti sljedeću naredbu.

```
lindex userslist 1
```

Ovime ćemo dohvatiti korisnika na indeksu 1 u listi koju smo maloprije kreirali.

Ukoliko želimo dohvatiti sve korisnike u nekom rasponu indeksa, možemo iskoristiti naredbu *lrange*.

```
lrange userslist 0 2
```

Kada želimo brisati elemente iz liste možemo koristiti naredbe *lpop* i *rpop*.

```
lpop userslist
```

Jedno od zanimljivih stvari u radu s listama je to da ne moramo eksplicitno kreirati i brisati liste. Redis će to učiniti za nas. Kada pokušamo dodati element u listu koja još uvijek ne postoji Redis će ju kreirati za nas i kada izbrišemo zadnji element iz liste, Redis će ju obrisati iz baze podataka za nas.

Četvrta struktura koju podržava ovaj sustav su skupovi. Ova struktura je slična listama, no ovom strukturom osiguravamo da nemamo dupliciranih vrijednosti u strukturi.

Dodavanje elemenata u ovu strukturu ide putem naredbe *sadd*.

```
sadd usersset  someuser
```

Ovaj sustav podržava skupove koji su sortirani. Kada dodajemo vrijednost u ovu strukturu putem naredbe *zadd* sustav će automatski sortirati vrijednosti u ovoj strukturi, odnosno smjestiti će novu vrijednost na pravo mjesto.

Ukoliko želimo doznati koje vrijednosti ima skup, možemo iskoristiti sljedeću naredbu.

```
smembers usersset
```

5.3.3. Ostale značajke

Jedna od korisnih naredbu u sustavu je i dohvaćanje svih ključeva.

```
keys *
```

Baza podataka nudi i mogućnost kreiranja ključeva koji traju određeno vrijeme, odnosno za neki ključ možemo postaviti *expire* svojstvo.

```
expire mykey 10
```

Također možemo doznati preostalo vrijeme koje će ključ biti valjan.

```
ttl mykey
```

Za razliku od MongoDb baze podataka Redis baza podataka podržava ACID svojstva. Ova baza podataka može biti konfigurirana tako da povećamo brzinu ali smanjimo sigurnost podataka. Ovo možemo uraditi tako da povećamo vrijeme nakon kojeg će podaci biti zapisani na disk. Početne postavke su da nakon svake operacije podaci se zapisuju na disk, no to možemo promijeniti tako da podatke zapisujemo nakon 1 sekunde. Treba još naglasiti da je ova baza podataka jednodretvena.

5.4. Cassandra

Cassandra je jedna od NoSQL baza podataka koju ćemo opisati, a koja se temelji na „Column familly“ modelu.

Slika 13. Logo sustava Cassandra

Ovaj sustav je posebno dizajniran za velike količine podataka i za korištenje istih podataka od strane više poslužitelja. Znači sustav je distribuiran. Ideja sustava je da bude korišten od strane više čvorova, što nam osigurava visoku raspoloživost sustava čak i u slučajevima kada dođe do neočekivanih situacija u nekom od čvorova.
Ovaj sustav ima visoke performanse prilikom čitanja i pisanja podataka. Sustav podržava *out of box*[24] podršku za replikaciju podataka u više podatkovnih centara.

5.4.1. Instalacija

Pokazat ćemo način instalacije na Fedora Linux distribuciji. Prvo što je potrebno napraviti je dodati novi repozitorij koji sadrži podatke potrebne za instalaciju ovog sustava. Potrebno je kreirati datoteku `/etc/yum.repos.d/datastax.repo` sa sljedećim sadržajem.

```
[datastax]
name = DataStax Repo for Apache Cassandra
baseurl = http://rpm.datastax.com/community
enabled = 1
gpgcheck = 0
```

Nakon ovoga možemo instalirati sustav sljedećom naredbom:

```
yum install dsc20
```

Nakon ovoga možemo pokrenuti poslužitelj.

```
service cassandra start
```

Kada je poslužitelj pokrenut možemo pokrenuti naredbeno sučelje ovog sustava.

```
cqlsh
Connected to Test Cluster at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.9 | CQL spec 3.1.1 | Thrift
protocol 19.39.0]
Use HELP for help.
cqlsh>
```

Putem ovog sučelja možemo upravljati značajkama cijelog sustava (Elasticsearch, 2014).
Nakon instalacija ovog sustava imamo i dostupno web sučelje, putem kojem možemo vizualno upravljati sustavom.

[24] Out of box – Nešto što se može koristiti jednostavno, bez mnogo dodatnog napora

Slika 14. Web sučelje sustava Cassandra

5.4.2. Komponente sustava

Sustav Cassandra se sastoji od mnogo manjih podsustava. Neki od njih su više neki manje bitniji za krajnje funkcioniranje sustava. Prema (Neeraj, 2013) neki od njih su:

- **Messaging service.** Mehanizam koji se brine o komunikaciji između različitih čvorova u sustavu. Kako bi čvorovi mogli komunicirati kreiraju po 2 soketa za svaki čvor. To znači ukoliko imamo 101 čvor, svaki čvor će imati otvorenih 200 soketa. Svaka poruka koja se razmjenjuje sadrži *verb* dio koji čvoru daje osnovne informacije o akcijama koje treba poduzeti s primljenom porukom.
- **Gossip.** Kao što i samo ime kaže, ovaj sloj sustava rasprostranjuje informacije na način na koji se rasprostranjuju uredski tračevi. Također se može usporediti s načinom na koji se širi virus. Ne postoji središnji odašiljač, ali se informacija širi na sve čvorove. Cassandra koristi ovaj sloj kako bi našla informacije o stanju i lokaciji čvorova u sustavu. Cassandra provjerava stanje sustava putem ovoga mehanizma u vremenskom intervalu od jedne sekunde.
- **Failure detection.** Ovaj sustav je jedan od osnovnih slojeva kod distribuiranih i velikih sustava. Sloj osigurava da se zahtjevi ne proslijeđuju na čvorove koji su nedostupni.
- **Partitioner.** Već smo rekli da je Cassandra distribuirani sustav, stoga su podaci raspodijeljeni između više čvorova u mreži. Kada unesemo neki podatak u ovaj sustav, sustav prema ključu koji smo unijeli sprema podatak na jedan od čvorova. Znači posao p*artitioner* podsustava je da „odreže“ dijelove podataka i sprema ih

na određeni čvor. Postoji nekoliko algoritama prema kojima p*artitioner* raspodjeljuje podatke između čvorova.

- **Replication.** Već smo vidjeli da se sustav Cassandra uvelike oslanja na spremanje podataka u različite čvorove. Ovo ima brojne prednosti, no jedan od nedostataka je situacija kada neki od čvorova postane nedostupan. Kako bi riješili ovaj problem koristimo replikaciju podataka na više čvorova. Replikacija podataka također ima više algoritama i kod sustava Cassandra je konfigurabilna. Jedna od glavnih opcija koje možemo postaviti kod replikacijskog procesa je *replication factor*. Ovime možemo reći sustavu Cassandra na koliko čvorova želimo imati kopiju podataka.
- **CommitLog.** Jedno od ACID svojstava koje Cassandra osigurava je *durability*, odnosno garancija da će podaci biti zapisani, odnosno kada dobijemo poruku *write_sucesufull* imamo garanciju da će podaci biti trajno zapisani. Ovo je osigurano na isti način na koji to osiguravaju relacijski sustavi. Zapisivanjem log informacija u određene datoteke putem kojih se podaci mogu obnoviti u slučaju neočekivanih situacija. Ova datoteka se zove *commitLog*.

5.4.3. CQL

CQL (Cassandra Query Language) je jezik sintaksom sličan SQL-u, i služi za pokretanje različitih operacija u Cassandra bazi podataka. Kod relacijskih sustava jedna od prvih stvari koje moramo napraviti je kreirati bazu podataka, a kod sustava Cassandra to je *keyspace*. Za svaki *keyspace* moramo definirati strategiju replikacije.

```
cqlsh> CREATE KEYSPACE keyspacename WITH REPLICATION = {
'class': 'NetworkTopologyStrategy'};
```

Ovime smo kreirali *keyspace*. Kako bismo se prebacili u novokreirani resurs, možemo iskoristiti sljedeću naredbu.

```
use keyspacename
```

Osim kreiranje *keyspace-a* možemo i ažurirati postojeći novim postavkama.

```
cqlsh> ALTER KEYSPACE keyspacename WITH REPLICATION =
{'class': 'SimpleStrategy', 'replication_factor': 1}
```

Kada smo kreirali i odabrali „bazu podataka“, možemo kreirati *column familly* u našem *keyspace-u*.

```
cqlsh:keyspacename > CREATE TABLE users (
     username varchar PRIMARY KEY,
     password varchar,
```

```
     email varchar
);
```

Kako bi pogledali dostupne *columny familly* resurse možemo iskoristiti sljedeću naredbu.

```
SHOW TABLES;
```

Ukoliko želimo doznati informacije o specifičnoj tablici možemo iskoristiti sljedeću komandu.

```
SHOW TABLE users;
```

Vidjeli smo da do sada CQL uvelike podsjeća na SQL jezik. Slična situacija se nastavlja i prilikom unošenja podataka u *keyspace*.

```
cqlsh:keyspacename> INSERT INTO users (username, password,
email) VALUES ('iivic', 'supe
rpwd', 'iivic@ivic.com');
```

Postavljanje upita je također može se reći identično načinu na koji smo navikli kod SQL jezika.

```
cqlsh:keyspacename> SELECT * FROM users;


 username | email          | password
----------+----------------+----------
    iivic | iivic@ivic.com | superpwd
```

U mogućnosti smo koristiti u mnoge druge naredbe koje smo navikli koristiti kod SQL jezika, kao npr WHERE, ORDER BY, itd..

Dodavanje novih kolona u *column familly* je također moguće putem CQL.

```
cqlsh:keyspacename> ALTER TABLE users ADD gender varchar;
```

Uz pomoć ALTER naredbe možemo ažurirati u npr. tipove podataka i ostale definirane vrijednosti objekata u bazi podataka.

Ažuriranje podataka radi na očekivan način.

```
cqlsh:keyspacename> UPDATE users SET
email='iivic@iivicnew.com' WHERE username='iivic';
```

Ukoliko želimo izbrisati neki red, to možemo učiniti na sljedeći način.

```
cqlsh:keyspacename> DELETE FROM users WHERE username='iivic';
```

CQL podržava upite kojima možemo dobiti više informacija o *keyspace-u*, klasteru i ostalim resursima u sustavu. Ukoliko želimo doznati više o „keyspace-u“ možemo iskoristiti sljedeće.

```
DESCRIBE KEYSPACE
```

Također, možemo doznati informacije o inačici samog CQL jezika, inačici poslužitelja Cassandra i slično. Npr. upit za dohvaćanje točne inačice CQL jezika.

```
cqlsh:keyspacename> SHOW VERSION
[cqlsh 4.1.1 | Cassandra 2.0.9 | CQL spec 3.1.1 | Thrift
protocol 19.39.0]
```

Također možemo koristiti *tracing* opciju u sustavu. Kada omogućimo *tracing*, sustav će nam ispisivati što točno radi, npr. što radi kada odgovara na neki upit. Naredba kojom možemo uključiti *tracing* je sljedeća.

```
TRACING on;
```

Ukoliko želimo izvršiti više naredbi za manipuliranje podacima, možemo koristiti *batch*. Ovime ćemo smanjiti broj zahtjeva koji će čvorovi izmjenjivati jer nećemo slati podatke čvorovima prilikom izvršavanja svakog pojedinačnog upita. Ovi zahtjevi su atomarni u sustavu Cassandra. Što to zapravo znači u ovom kontekstu. To znači da ukoliko bilo koji *batch* uspije, svi *batch* zahtjevi moraju uspjeti.

```
cqlsh:keyspacename> BEGIN BATCH
INSERT INTO users (username, email) VALUES ('pperic',
'pperic@pperic.com');
INSERT INTO users (username, password) VALUES ('kkovic',
'somepassword');
APPLY BATCH;
```

Sustav naravno podržava indekse. Dodatne indekse možemo kreirati na sljedeći način.

```
cqlsh:keyspacename> CREATE INDEX emailindex ON users(email);
```

Indeks možemo izbrisati na sljedeći način.

```
cqlsh:keyspacename> DROP INDEX emailindex;
```

CQL jezik naravno ima još naredbi i mogućnosti, kao i sami sustav Cassandra. Ovo su bile neke od osnovnih, kako bi dobili osjećaj kako ovaj sustav funkcionira.

5.5. Elasticsearch

Elasticsearch je sustav za pretraživanje, iznimno brz i posjeduje REST HTTP API sučelje za pristupanje različitim operacijama u sustavu. Sustav je *open source* i dostupan je na različitim platformama. Sam sustav je implementiran u programskom jeziku Java i u pozadini koristi Apache Lucene tražilica *(engl. search engine).*

Slika 15. Logo Elasticsearch sustava

Jedna od alternativa ovom sustavu je Alache Solr. On je nešto stariji sustav koji služi istoj svrsi kao i sustavi Elasticsearch. Elasticsearch u usporedbi s Apache Solr pruža jednostavnije sučelje i zahtjeva manje konfiguracijskih podataka koji su potrebni da bi sustav normalno radio. Također još jedna od razlika je i da je Elasticsearch bolji izbor kada imamo puno operacija kojima možemo uzrokovati ažuriranje indeksa. Naime Apache Solr arhitektura je napravljena u vrijeme kada nije bilo predviđeno da jedan takav sustav prima mnogo operacija koje ažuriraju indeks, dok je Elasticsearch noviji sustav i autori sustava su bili svjesni ove činjenice i odradili su dobar posao.

Nakon pokretanja Elasticsearch poslužitelja prvo što trebamo napraviti je kreirati indeks i popuniti ga podacima koje ćemo kasnije pretraživati. Jedna od stvari koja može zbuniti je da se Elasticsearch čini kao prava baza podataka, jer posjeduje većinu karakteristika kao i normalne baze podataka. To je istina, no ovaj sustav primarno nije namijenjen da služi kao baza podataka, iako ga je moguće za to koristiti. Ideja ovog sustava je da bude u sinkronizaciji s nekom pravom bazom podataka i svi podaci koji u nju dolaze, da se repliciraju na Elasticsearch sustav, koji će podatke indeksirati i staviti na raspolaganje svima koji ih žele brzo pretraživati prema različitim kriterijima.

Kada dodajemo nove dokumente u indeks, ukoliko indeks ne postoji bit će kreiran za nas. Elasticsearch će automatski prepoznati tipove polja koje smo poslali u dokumentu. Naravno bolja opcija je da definiramo unaprijed tipove polja koja će sadržavati neki tip podataka u indeksu. Osim što ovime možemo definirati polja našeg tipa podataka, također možemo definirati koji su atributi pretražiti i koji atributi će biti indeksirani. Naravno ovo kao i sve ostale operacije radimo putem REST API sučelja.

Kao i prethodno opisani sustavi i ovaj posjeduje programska sučelja prema različitim programskim jezicima.

5.5.1. Instalacija

Pokazat ćemo instalacijski proces za Fedora Linux distribuciju. Kako bi instalirali sustav potrebno je dodati novi *yum* repozitorij sa sljedećim informacijama.

```
[elasticsearch-1.3]
name=Elasticsearch repository for 1.3.x packages
baseurl=http://packages.elasticsearch.org/elasticsearch/1.3/ce
ntos
gpgcheck=1
gpgkey=http://packages.elasticsearch.org/GPG-KEY-elasticsearch
enabled=1
```

Nakon ovoga jednom naredbom biti ćemo u mogućnosti instalirati ovaj sustav.

```
yum install elasticsearch
```

Nakon toga biti ćemo u mogućnosti pokrenuti Elasticsearch servis.

```
service elasticsearch start
```

5.5.2. Rad sa sustavom

Ovim sustavom se upravlja putem REST sučelja. To znači da ga možemo koristiti iz bilo kojeg programskog jezika koji podržava slanje REST naredbi poslužitelju.

Nakon što smo instalirali i pokrenuli Elasticsearch sustav, možemo provjeriti je li sustav normalno radi sljedećim GET zahtjevom.

```
curl -X GET http://localhost:9200/
```

Odgovori na zahtjeve su u JSON formatu, što je također idealan format zapisa za procesiranje od različitih programskih jezika.

Sada ćemo kreirati indeks *products* u koji ćemo spremati informacije o proizvodima.

```
curl -X PUT 'http://localhost:9200/products/' -d '{
    "settings" : {
        "number_of_shards" : 3,
        "number_of_replicas" : 2
    }
}'
```

Vidimo da smo u indeks dodali i neke opcije. Vidimo pojmove *shards* i *replicas*. Pogledajmo što točno ovi pojmovi znače.

- **Shard.** Kao što smo već rekli u pozadini se ovaj sustav temelji na tražilici Apache Lucene. Shard je jedna instanca Apache Lucene sustava. To je *worker* niske razine

i njime upravlja sustav Elasticsearch. Nakon ovoga možemo reći da je indeks logički prostor koji pokazuje na primarni i Replica Shard.

- **Replicas.** Svaki primarni *Shard* može imati 0 ili više replika. Replika je kopija primarnog *Shard-a*.

Ovaj sustav će za nas kreirati strukturu podataka u indeksu, tzv. Mapiranja (engl. mappings). U većini slučajeva ovo će biti zadovoljavajuće, no možemo i eksplicitno definirati mapiranja u sustavu.

```
curl -X PUT 'http://localhost:9200/products/product/_mapping'
-d '
{
    "product" : {
        "properties" : {
            "name" : {"type" : "string", "store" : true }
        }
    }
}'
```

Definiranje *mappings* svojstava indeksa je široka tema u ovom sustavu i nećemo ulaziti u detalje. Nakon što smo kreirali indeks možemo dodati neke dokumente u njega.

```
curl -XPUT 'http://localhost:9200/products/product/1' -d '{
    "name" : "Coca cola"
}'
```

Kada smo dodali dovoljno dokumenata, možemo isprobati primarnu svrhu ovog sustava. Pretraživanje.

```
curl -XGET
'http://localhost:9200/products/_search?pretty=true' -d '
{
    "query" : {
        "match" : {
            "name" : "Coca cola"
        }
    }
}'
```

U ovom upitu tražimo sve dokumente s vrijednosti *Coca cola* na atributu ime. Ovo je jedan od najjednostavnijih primjera pretraživanja u ovom sustavu. Sustav ima cijeli DSL specijaliziran

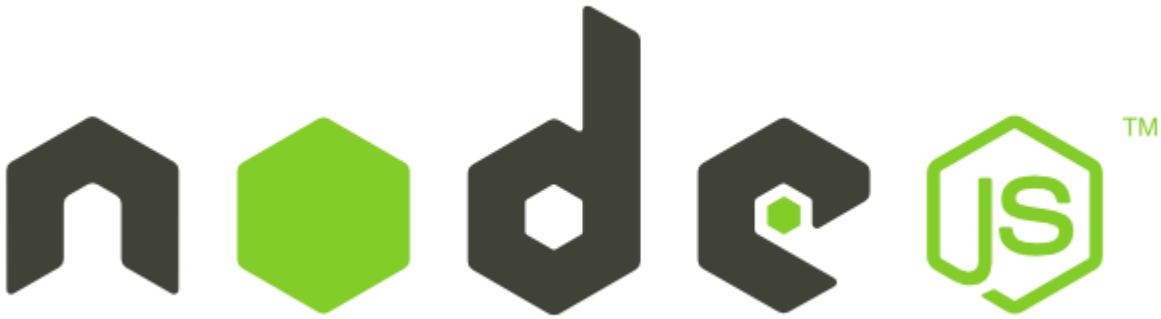
za različite vrste pretraživanja dokumenata. Ovo je također široka tema i u ovom radu nećemo ulaziti u detalje.

Zanimljiv koncept koji možemo naći u Elasticsearch-u je *river*. *River* je zapravo servis koji uzima ili šalje podatke Elasticsearch sustavu. Svaki *river* ima unikatno ime i tip. Za nas je zanimljiv MongoDb r*iver*, koji čita *oplog*[25] zapise i na osnovu podataka u toj datoteci pokreće akcije dodavanja, brisanja i ažuriranja podataka u određenom indeksu koji se nalazi u Elasticsearch sustavu.

5.6. Node.js

Kada se Javascript prvi puta pojavio bio je iznimno jednostavan programski jezik. Prvo je bio poznat pod imenom *Mocha*, zatim pod imenom *Livescript* i na kraju svima danas poznato ime Javascript. U početku programi napisati u Javascript jeziku bili su svega po par linija koda. Služili su za neke najosnovnije programske akcije unutar web preglednika.

No, malo po malo, Javascript jezik je postajao sve moćniji i moćniji jezik. Jedna od prekretnica u povijesti Javascripta sigurno je bila i pojava *jQuery* biblioteke, koja je na iznimno jednostavan način omogućavala različite akcije unutar web preglednika.



Slika 16. Logo Node.js sustava

Godina 2009 bila je sigurno još jedna prekretnica u razvoju Javascript jezika. Ryan Dahl je te godine na Javascript konferenciji predstavio projekt koji omogućava izvršavanje Javascript koda na poslužitelju. Arhitektura cijelog sustava bila je jako zanimljiva. Cijeli

[25] Oplog – Posebna kolekcija u MongoDb sustavu koja sadrži informacije o prošlim akcijama u bazi podataka.

sustav se temeljio na *non blocking*[26] filozofiji i cijeli Javascript kod se izvršavao u jednoj dretvi. Inicijalno je ovaj sustav bilo moguće pokrenuti samo na Linux operacijskom sustavu, no ubrzo s velikim brojem kontribucija ovaj projekt je evoluirao za samo 4 godine u jednu od najzanimljivijih poslužiteljskih tehnologija.

Arhitektura Node.js-a temelji se na Google V8 stroju, *libuv* biblioteci te *libeio* biblioteci. Naravno uz ove biblioteke postoje i mnoge druge, no ove tri su vjerojatno najvažnije značajke ove tehnologije.

Node.js dolazi i s NPM[27] *package manager* programom, koji nam omogućava da instaliramo i definiramo zavisnosti za naš program putem obične JSON datoteke.

Slika 17. Logo npm sustava

Primjer instaliranja *lodash* biblioteke putem NPM biblioteke je sljedeći.

```
npm install lodash --save
```

Ukoliko smo proslijedili i *--save* opciju, tada će NPM dodati ovu biblioteku u naš *package.json* kao zavisnost našeg sustava.

Sve ove činjenice privukle su programere da počnu konstantno razvijati nove module za Node.js. Jedno od najpopularnijih mjesta za objavu Node.js modula je Github[28].

Naveli smo mnoge prednosti ove platforme, no kao i sve druge platforme i ova ima neke nedostatke. Jedna od najvećih prednosti ove platforme je što omogućava izvršavanje Javascript jezika, što znači da ćemo naše programe pisati u tom programskom jeziku. Jezik je iznimno moćan i fleksibilan, no ta fleksibilnost ima i cijenu. Kao što znamo Javascript je dinamički programski jezik, što znači da ne posjeduje kompajler nego interpreter. Ukoliko radimo neke manje sustave korištenjem ove tehnologije ta činjenica ne predstavlja velik problem za nas, no ukoliko radimo velike sustave korištenjem ove tehnologije tada stvari postaju malo teže.

[26] Non blocking – Arhitektura sustava u kojem se dretva koja je napravila neki zahtjev ne zaustavlja dok odgovor na zahtjev nije dostupan

[27] NPM (Node package manager) – Program primarno namjenjen za instaliranje, ažuriranje i brisanje programa napisanih za Node.js platformu

[28] Github – Servis čija je primarna namjena da pruža usluge verzioniranja koda

Kod refaktoriranja koda kod programskih jezika koji su kompajlerski, prevoditelj će nam javiti sve pogreške koje su nastale našim akcijama. Kod Node.js-a to naravno nije slučaj. Kako bi bili u mogućnosti refaktorirati veće sustave moramo imati jako dobro napisane testove, koji su nam jedino osiguranje da stvari još uvijek rade. Osim refaktoriranja koda, fleksibilnost Node.js-a plaća se manjkom IDE[29] alata koji pružaju mogućnosti kao kod kompajlerskih programskih jezika. Jedna od stvari koje su postale potpuno normalne u svijetu npr. programskog jezika Java je *auto completion*[30]. Kod Javascript jezika još uvijek nemamo niti približno dobre alate kao što su npr IntelliJ za Javu, jer kod jezika kao što je Javascript, prilično je teško odrediti što ponuditi korisniku, jer se kontekst programa, tipovi varijabli i slično može promijeniti u svakom trenutku. Zbog ove činjenice ni već spomenuto refaktoriranje nije najsretnije podržano od strane IDE alata.

5.6.1. Primjer jednostavne aplikacije

U ovom poglavlju ćemo predstaviti jedan jednostavan Node.js program (Node.js, 2014).

```
var http = require('http');

var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World\n");
});

server.listen(3000);
console.log("Server running at http://127.0.0.1:3000/");
```

U prvom liniji vidimo *require* funkciju. To je jedna od glavnih funkcija koje su dostupne u Node.js-u. Tom funkcijom zahtijevamo neki vanjski modul i u mogućnosti smo mu pristupiti preko lokalne varijable, u ovom slučaju „http“.

Node.js se može definirati na više načina. Node.js je i HTTP poslužitelj i u ovom slučaju ga koristimo na taj način. Krelirali smo *HTTP* poslužitelj koji čeka zahtjeve na portu 3000 i na njih odgovara „Hello world“ porukom. Vidimo da smo u mogućnosti upravljati zaglavljima odgovora na HTTP zahtjev. Također smo u mogućnosti upravljati u drugim svojstvima odgovora, npr. kolačićima.

[29] IDE (Integrated development environment) – Okružje za razvoj računalnih programa koje se sastoji od više komponenti

[30] Auto completion – Predviđanje korisničkog unosa, i predlaganje mogućih opcija koje bi korisnik mogao napisati

5.7. Ember

Programeri koji se bave izradom web aplikacija su često u doticaju s programskim jezikom Javascript. Većinom je to putem biblioteka kao što su jQuery, Prototype, Mootools i sličnima. Ukoliko su zahtjevi aplikacije mali, tada pisanje Javascript koda ne zahtjeva veliko planiranje kako organizirati tih stotinjak linija. No ukoliko su zahtjevi aplikacije veći i želimo smjestiti dio poslovne logike na klijentsku stranu, tada moramo pažljivo razmisliti o strukturiranju našeg koda na klijentskoj strani aplikacije.

Ljudi već godinama razvijaju kompleksne aplikacije za web preglednik i godinama su skupili iskustva u tom području. Jedan od timova takvih ljudi je i tim koji stoji iza odličnog Javascript okvira (engl. framework) namijenjenog za izradu aplikacija za klijentsku stranu, a ime samog okvira je Ember i jedan je od *single page application*[31] okvira.

Cijeli okvir se temelji na konvencijama koje moramo pratiti ukoliko razvijamo aplikacije uz pomoć ovog okvira. Neki mogu smatrati ovo kao manjak fleksibilnosti, no nakon što naučimo koncepte na kojima se temelji, primjetiti ćemo da su ove konvencije zapravo najbolja strana ovog okvira. Konvencija nas forsira na upotrebu nekih dobrih praksi, odnosno praktičnih iskustava tima koji stoji iza ovog okvira.

Slika 18. Logo Ember okvira

Opišimo sada jedan uobičajen niz koraka koji nas čeka ukoliko želimo napraviti našu početnu stranicu.

Strukturu naše aplikacije je definirana putem skup putanja koje su dostupne u aplikaciji. Ukoliko definiramo putanju *index*, tada Ember očekuje da definiramo objekte

[31] Single Page Application. Moderne aplikacije gdje je cijela aplikacija na jednoj web stranici, odnosno cijela aplikacija se nikad ne osviježava, nego samo neki njezini dijelovi

IndexRoute, IndexController i *IndexView*. Već sada možemo vidjeti da okvir ima neke elemente *MVC*[32] uzorka dizajna.

Objekt *IndexRoute* je zadužen za učitavanje podataka koji će biti dostupni u ovom dijelu aplikacije, za postavljanje nekih opcija kontrolera, za parsiranje argumenata naše putanje i slično. Nakon toga imamo objekt *IndexController*. U njemu definiramo poslovnu logiku ukoliko takva postoji, definiramo podatke specifične za ovaj dio sustava, te definiramo događaje na koje ovaj dio sustava može reagirati.

Zadnji dio je IndexView. Kao što i pretpostavljamo, ovaj objekt je zadužen za direktnu komunikaciju s DOM stablom u web pregledniku. Tu možemo koristiti neke biblioteke kao što je jQuery, možemo dodati neke nove i slično. Također je ovaj dio sustava namijenjen za upravljanje specifičnim događajima koji dolaze iz DOM[33] stabla. Ovaj objekt ima pridružen predložak (engl. template), odnosno HTML reprezentaciju ovog dijela sustava. U tom HTML reprezentaciji možemo naći i neke neuobičajene konstrukte, kao što je *{{#if}}, {{#else}}, {{#each}}* i slično. Razlog tomu je što Ember koristi Handlebars *template engine*, koji ima HTML sintaksu proširenu dodatnim konstruktima kojima možemo definirati pravila putem kojih se geneira konačan HTML kod.

5.7.1. Primjer jednostavne aplikacije

Prvo što trebamo napraviti je kreirati instancu Ember aplikacije.

```
var App = Ember.Application.create({});
```

Prilikom kreiranja aplikacije moguće je proslijediti dodatne postavke, no mi ćemo u ovom slučaju koristiti početne.

Nakon toga potrebno je definirati putanje u našoj aplikaciji.

```
App.Router.map(function() {
     this.resource('users', { path: 'users' }, function() {});
});
```

Vidimo da smo definirali *users* putanju i kao što smo već objasnili Ember će očekivati da kreiramo i prikladne *controller, view i route* objekte.

```
App.IndexRoute = Ember.Route.extend({
    beforeModel: function() {
        this.transitionTo('users');
```

[32] MVC – Model View Controller uzorak dizajna koji se često koristi pri izradi web aplikacija

[33] DOM (Document Object Model) – DOM je višeplatformsko aplikacijsko programsko sučelje za valjane HTML i XML dokumente

```
    }
});
App.UsersRoute = Ember.Route.extend({
    model: function() {
        return [{
            id: 1,
            name: 'Ivan'
        }, {
            id: 2,
            name: 'Ante'
        }];
    }
});
```

Definirali smo dva *route* objekta. *Index* putanju koja će biti pozvana kada pristupimo početnoj stranici sustava i ta će nas putanja automatski prebaciti na drugu putanju *users* u kojoj smo definirali model koji će biti prikazan u predlošku aplikacije.

```
App.UsersController = Ember.ArrayController.extend({
    actions: {
        handleClick: function() {
            alert('click detected!');
        }
    }
});
```

Akcijama uglavnom upravljamo iz kontrolera. Ovdje smo definirali obradu događaja za akciju pod imenom *handeClick*. Ova akcija će biti pozvana iz predloška.

```
App.UsersView = Ember.View.extend({
    templateName: 'users'
});
```

Zadnji objekt koji ćemo definirati je pogled ove putanje i on nosi informaciju o imenu predloška ove putanje. Sam predložak izgleda ovako.

```
<script type="text/x-handlebars" data-template-name='users'>
{{#each user in content}} {{user.id}} - {{user.name}} <br/>
{{/each}}
<input type="button" value="Click me" {{action handleClick}}/>
```

```
</script>
```

Vidimo da čitamo vrijednosti iz modela putanje koji smo definirali u model *hook-u* u *UsersRoute* objektu. Također u ovom predlošku smo rekli da će se pritiskom na gumb pozvati *handleClick* funkcija *UsersController* objekta.

5.8. Ember data

Kao što ime nalaže, ova biblioteka je povezana i korištena od strane Ember okvira i napravljena je od strane istog tima i temelji se ponovo na nekim konvencijama koje su se pokazale kao dobre u praksi.

U gotovo svakoj aplikaciji, jako važan dio aplikacije zauzimaju podaci i upravljanje istima. U SPA[34] okviru kakav je Ember podaci dobivaju još veći naglasak i efikasno upravljanje istima može unaprijediti performanse aplikacije i unaprijediti samo korisničko iskustvo prilikom korištenja jedne takve aplikacije.

Putem ovog okvira možemo definirati objekte i relacije među njima što je jedna od prvih akcija koje radimo kada definiramo modele podataka u našoj aplikaciji. Možemo definirati uobičajene veze među objektima, kao što su jedan na prema više, jedan na jedan, više na više i to sve putem Javascript koda. Uz ovo biblioteka podržava priručnu memoriju (engl. cache) za podatke koje smo već dohvatili sa poslužitelj, tako da podatke ne moramo dohvaćati više puta. Uzimajući u obzir da se ova biblioteka koristi u kombinaciji sa SPA okvirom, to implicira da korištenjem ove biblioteke možemo drastično unaprijediti performanse pojedinih dijelova aplikacije, jer možemo izbjeći ponovno učitavanje podataka sa poslužitelja, a kao što znamo učitavanje podataka sa poslužitelja je jedan od poslova koji web pregledniku uzimaju najviše vremena.

U vrijeme pisanja ovog diplomskog rada ova biblioteka je još uvijek bila u 1.0 beta inačici, no pošto ju već mnoge firme koriste u produkcijskim okružjima i pokazala se kao stabilna i u ovoj inačici, korištena je i u ovom diplomskom radu, gdje se također pokazala odličnom.

5.9. Socket.io

Napredovanjem web tehnologija sve više poslovne logike se premješta na klijentsku stranu, te klijent postaje sve „deblji“. Već smo vidjeli neke od alata koji nam omogućuju

[34] SPA – Single Page Application. Moderne aplikacije gdje je cijela aplikacija na jednoj web stranici, odnosno cijela aplikacija se nikad ne osvježava, nego samo neki njezini dijelovi

razvoj takvih aplikacija (Ember okvir) i kod ovakve vrste aplikacija često imamo potrebu često osvježavati podatke.

Ukoliko se bavimo razvojem web aplikacija AJAX[35] tehnika nam je vrlo vjerojatno poznata, no postavlja se pitanje je li ona dovoljno dobra za ovakve tipove aplikacije u slučajevima kada imamo zahtjeve u stvarnom vremenu?

Uzmimo za primjer jedan sustav obavijesti kojeg su implementirali mnogi sustavi koje svakodnevno koristimo. Ukoliko želimo da ovaj sustav radi što bolje, kašnjenje prijenosa obavijesti na web preglednik mora biti što manje, kako bi korisničko iskustvo bilo bolje. Ako želimo ovakav sustav implementirati koristeći AJAX tehnologiju, morali bismo slati zahtjeve na poslužitelj nakon nekog malog vremenskog intervala, npr. 1 sekundu. Ovakav pristup jednostavno nije slakabilan, jer povećanjem broja korisnika takvog jednog sustava, performanse aplikacije će postajati slabije zbog preopterećenja poslužitelja. Ovakve probleme arhitekti često rješavaju horizontalnim skaliranjem sustava, no to je samo privremeno rješenje za ovaj problem.

U novije vrijeme pojavio se protokol *websockets*. Protokol omogućava komunikaciji između klijenta i poslužitelja u oba smjera putem trajne konekcije. Ovo drastično smanjuje opterećenje poslužitelja, jer klijenti dobijaju obavijesti samo kada one pristignu u jako kratkom vremenu.

Slika 19. Logo socket.io biblioteke

Nakon pojave ovog protokola počele su se pojavljivati mnoge biblioteke koje su pojednostavljivale rad s ovom tehnologijom i donosile zanimljive dodatne mogućnosti. Jedna od biblioteka koja je postala *de facto* standard u radu s *websocket-ima* u Javascript programskom jeziku je Socket-io.

Ova biblioteka omogućava komunikaciju u stvarnom vremenu između klijenta i poslužitelja putem modela temeljenog na događajima. Također, omogućava slanje poruka samo nekom skupu korisnika, odnosno onima koji su zainteresirani za tu vrstu informacija. Primjer toga su sobe u aplikacijama za razgovor. Biblioteka omogućava i autentikaciju i

[35] AJAX – Asinkroni Javascript i XML

autorizaciju korisnika koji pristupaju dostupnim resursima, te se odlično uklapa u poslužiteljsku tehnologiju kao što je Node.js.

5.9.1. Primjer jednostavne aplikacije

Implementirati ćemo jednostavnog klijenta i jednostavnog poslužitelja koji će razmijenjivati poruke. Poslužiteljski dio može izgledati ovako.

```
var socketio = require('socket.io');

var io = socketio.listen(3636);
io.sockets.on('connection', function (socket) {
    io.sockets.emit('server_message', 'message from server');

    socket.on('client_message', function (data) {
        // do something
    });
});
```

Kada se klijent uspješno spoji možemo mu slati poruke putem soket objekta. Također smo u mogućnosti primati poruke. Ovo možemo putem jednostavno API sučelja uz pomoć funkcija *on* i *emit*. Klijentska strana je sljedeća.

```
<script>
    var socket = io("http://localhost:3636");
    socket.on('server_message', function(msg) {
        socket.emit('client_message', 'Message from client');
        // do something with server message
    });
</script>
```

Kao što vidimo klijent definira lokaciju poslužitelja i port, te je nakon toga u mogućnosti izmjenjivati poruke sa poslužiteljem s istim API sučeljem kao što smo maloprije vidjeli (*on* i *emit*).

5.10. Grunt

Priprema aplikacija za produkcijsko okruženje uobičajeno podrazumijeva neki skup radnji koje moramo obaviti kako bi naša aplikacije radila što bolje u takvom okruženju. Također prilikom razvoja aplikacije često radimo iste poslove na svakom projektu.

Ukoliko se bavimo razvojem aplikacija temeljenih na Javascript programskom jeziku takve radnje su npr. spajanje svih Javascript datoteka u jednu, te kompresiranje datoteke, kompresiranje slika, dohvaćanje dodatnih biblioteka, pokretanje testova, kopiranje datoteka s jedne lokacije na drugu i slično.

Slika 20. Logo Grunt sustava

U novije vrijeme su se pojavili alati koji mogu automatizirati ove radnje. Alati su uglavnom specifični za programski jezik, a u Javascript svijetu trenutno su dva lidera. To su Grunt i Gulp. Gulp je noviji i brži jer se temelji na arhitekturi koja koristi tokove (engl. stream-ove), no Grunt je još uvijek popularniji, najvećim dijelom jer je stariji i ima više dostupnih dodataka.

Ovakvi alati pružaju mogućnost definiranja poslova koje možemo pokrenuti jednostavnim naredbama. Te poslove možemo kombinirati te dodatno konfigurirati. U projektu koji je realiziran za potrebe ovog diplomskog rada korišten je Grunt *build system*, te su razvijeni zadaci od strane autora diplomskog rada koji pomažu pri upravljanju testnim podacima.

6. Praktični dio

U okviru završnog rada biti će razvijena web aplikacija koja pokazuje na koji način možemo integrirati više baza podataka i iskoristiti heterogenost kako bismo poboljšali rad cijelog sustava. Svrha aplikacije je omogućiti korisnicima objavljivanje proizvoda, i kupovinu proizvoda putem ulaganja određene sume novca. Aplikacija se sastoji od više vrsta baza podataka, Node.js poslužitelja, te klijentske SPA aplikacije. Aplikacija se sastoji od standardnih funkcionalnosti kakve su prijava i registracija, te se sastoji od funkcionalnosti dodavanja novih proizvoda, kategorija, uređivanja, brisanja istih i slično. Za svaki proizvod korisnici aplikacije mogu ulagati novac, odnosno ulagati novce na neki proizvod. Aplikacija ima implementiranu autorizaciju, tako da razlikujemo nekoliko vrsta korisnika na sustavu.

To su:

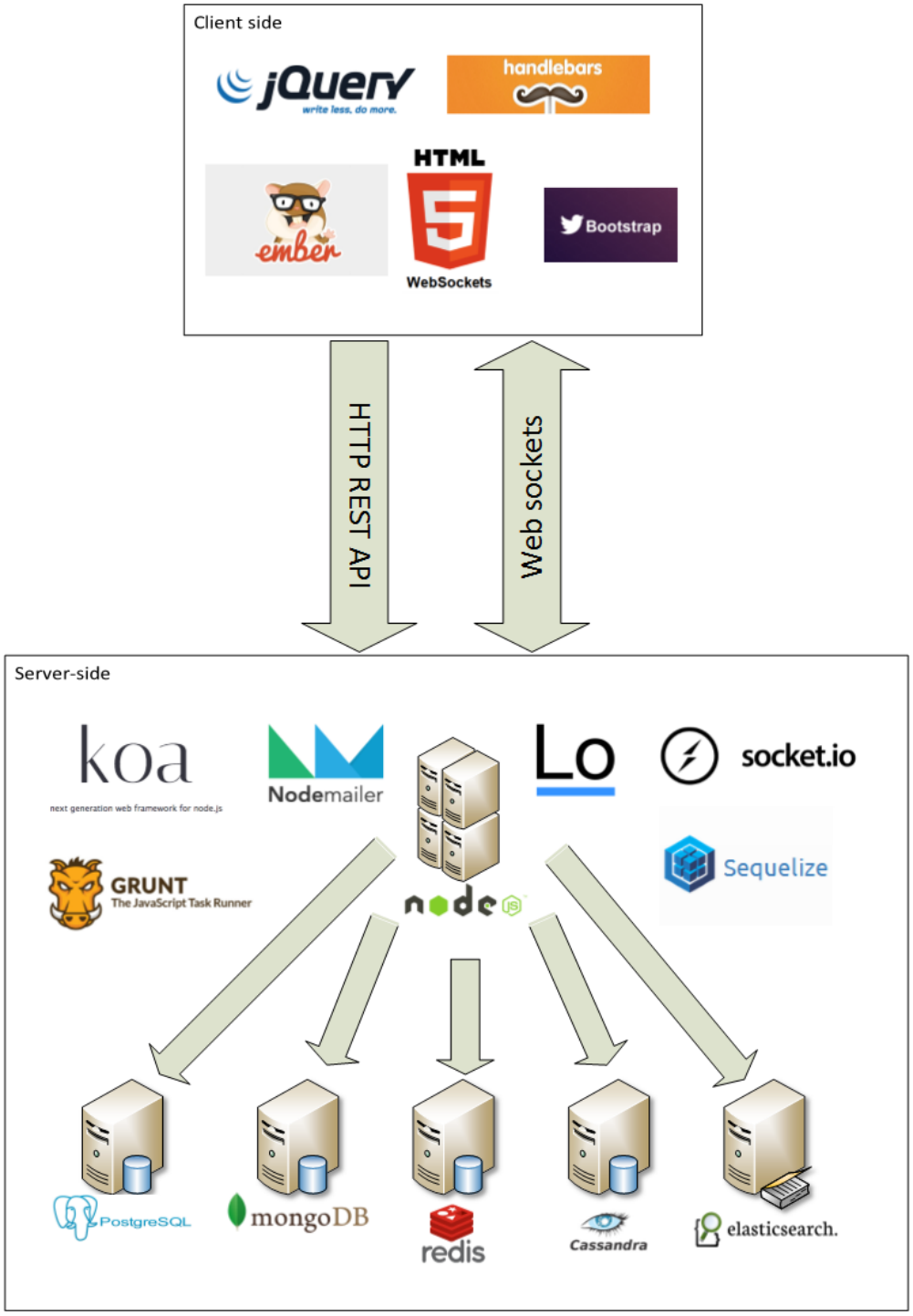
- Neregistrirani korisnik
- Registrirani korisnik
- Moderator
- Administrator

Neregistrirani korisnici imaju mogućnost pretraživanja proizvoda, te registriranja na sustav. Registrirani korisnici imaju dodatnu mogućnost, a to je objavljivanje i uređivanje vlastitih proizvoda. Registrirani korisnici također imaju mogućnost ulaganja novca i kupovanja virtualnog novca.

Moderatori imaju mogućnost uređivanja proizvoda drugih korisnika, te imaju mogućnost dodavanja novih kategorija u sustav. Administratori imaju sva navedena prava i dodatnu funkcionalnost pregledavanja akcija na sustavu, npr. log podataka u sustavu i sl.

6.1. Arhitektura sustava

U ovom poglavlju predstavit ćemo arhitekturu na kojoj se temelji sustav koji je razvijen u sklopu ovog diplomskog rada.



Slika 21. Arhitektura razvijenog sustava

Krenut ćemo od poslužiteljske strane sustava. Korisnici će ovaj sustav vidjeti kao REST API, što je omogućeno putem Node.js-a na kojem se temelji poslužiteljska strana sustava. Uzimajući u obzir činjenicu da se poslužiteljska strana temelji na Node.js-u, sve korištene

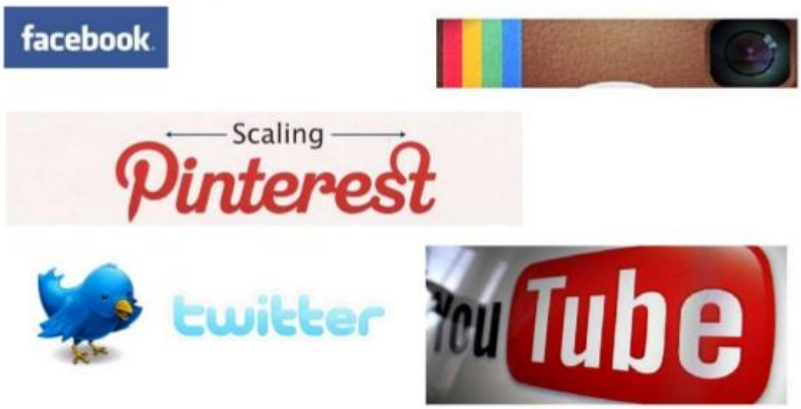
biblioteke su napisane u Javascript programskom jeziku i prilagođene su za Node.js platformu. Veći dio slike na poslužiteljskoj strani zauzima dio s bazama podataka. One su jako bitna komponenta ovog sustava i vjerojatno jedna od njegovih najvećih snaga ali i nedostataka. Poslužiteljska strana također podržava i komunikaciju putem *websocket-a* ukoliko su nam potrebni podaci u stvarnom vremenu.

Klijentska strana se temelji na Ember okviru. Također, klijentska strana koristi HTML5 *websocket* tehnologiju za komunikaciju u stvarnom vremenu sa poslužiteljem. Kao jezik za predloške na klijentskoj strani se koristi Handlebars.

Cijeli sustav, uključujući i klijentsku stranu se oslanja na Grunt *build tool*. Već smo ga prethodno detaljno opisali i u ovom projektu ima veliku ulogu. Jedna od stvari koje radi ovaj sustav je spajanje svih Javascript CSS dokumenata i njihova kompresija. Prilikom razvijanja sustava nakon svake promjene sustav pokreće spomenute akcije kako bi osvježio datoteke u skladu s novim promjenama. Za sučelje na klijentskoj strani zadužen je Bootstrap *front end* okvir.

6.2. Integracija baza podataka

Aplikacija koristi nekoliko vrsta baza podataka, te zbog toga imamo više manjih modela koji su smješteni u više različitih sustava za upravljanje bazama podataka. Krajnji cilj nam je integrirati pristup bazama podataka putem programske apstrakcije. Pitamo se možda je li je ovo samo eksperiment ili ga koriste već neke kompanije kako bi ostvarile svoje ciljeve.

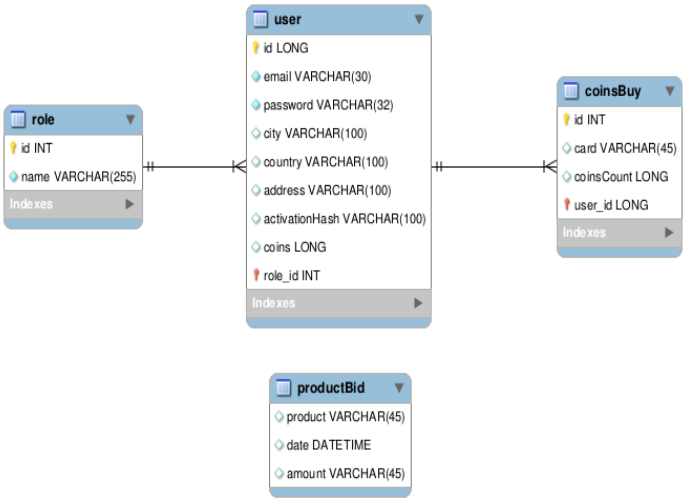


Slika 22. Neke od kompanija koje koriste heterogene baze podataka

Na slici su nama svima poznate kompanije koje koriste više baza podataka. Svaka od njih se odlikuje svojim prednostima i nedostacima, no cilj je iskoristiti što više prednosti sustava u područjima u kojima je sustav za to prikladan. Pogledajmo sada kako izgledaju modeli podataka na kojima se temelji implementirana aplikacija.

6.2.1. Modeli baza podataka

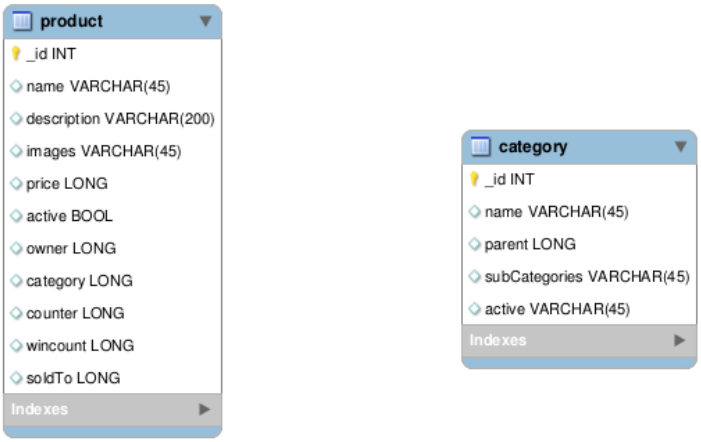
6.2.1.1. PostgreSQL



Slika 23. Korišteni relacijski model podataka

Model PostgreSQL baze podataka se sastoji od tablice za spremanje korisnika, tablice za spremanje korisničkih uloga, ulaganja novca na proizvode, te podataka o kupovini virtualnog novca. Vidimo da tablica za spremanje ulaganja novca na proizvode nije povezana niti s jednom drugom tablicom. Razlog je što je tablica s proizvodima zapravo spremljena u MongoDb bazi podataka. Pogledajmo sada model MongoDb baze podataka.

6.2.1.2. MongoDb



Slika 24. Korišteni dokumentni model podataka

Vidimo da se ovaj model sastoji od dvije kolekcije. To su kolekcija s proizvodima i kolekcija s kategorijama proizvoda. Možemo primjetiti da kolekcija produkt i kategorija nisu eksplicitno povezane, nego je veza između njih implementirana spremanjem reference na *_id* koji se nalazi u drugoj kolekciji. Iako se čini dosta slično kao kod relacijskih baza podataka, zapravo je dosta različitno, posebno na implementacijskoj razini.

6.2.1.3. Redis

U većini ozbiljnih web aplikacija imamo pojam korisničkih sesija. One su jako bitne kako bismo mogli održavati stanja između više neovisnih HTTP zahtjeva. Jedna od ključnih stvari u jednoj web aplikaciji može biti i način na koji upravljamo korisničkim sesijama.
Ukoliko imamo takvu aplikaciju, vrlo vjerojatno ćemo htjeti pri svakom novom zahtjevu dohvatiti podatke o korisničkoj sesiji iz nekog izvora. Podaci mogu biti smješteni u kolačiću koji se prenosi u korisnički preglednik, no ovo nije dobro rješenje, jer smo ograničeni količinom podataka koju možemo spremiti, a i otvaramo vrata sigurnosnim problemima ukoliko se u kolačiću nalaze osjetljivi podaci.

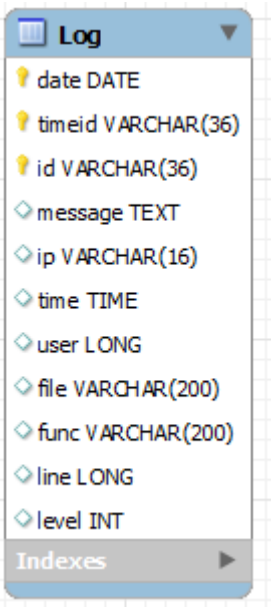
Rješenje za ova dva spomenuta problema može biti da spremimo identifikator u kolačić, a podatke o sesiji spremimo u bazu podataka. U ovom slučaju više nismo ograničeni što se tiče količine podataka koju možemo spremiti, a niti otvaramo vrata sigurnosnim propustima, jer spremamo podatke u bazu podataka koja se nalazi na poslužiteljskoj strani aplikacije i pod našom je kontrolom.

Slika 25. Korišteni key-value model podataka

Na slici vidimo i model podataka koji koristi razvijena aplikacija za spremanje korisničkih sesija.

6.2.1.4. Cassandra

Za spremanje velike količine log zapisa iskorišten je Cassandra sustav. Ideja je prikupiti podatke o aktivnostima korisnika, koji mogu poslije pomoći u poboljšanju rada samog sustava, ili npr. mogu pomoći prilikom otkrivanja neželjenih aktivnosti.



Slika 26. Korišteni Column-Familly model podataka

Na prethodnoj slici vidimo i grafičku reprezentaciju korištenog modela podataka.

6.2.2. Programska implementacija modela baza podataka

U ovom dijelu ćemo vidjeti na koji smo način programski implementirali neke od navedenih modela podataka. Osim za sustav Cassandra, modeli su implementirani u Javascript programskom jeziku i korišteni su ODM i ORM[36] alati, koji će za nas generirati naredbe kojima će se kreirati modeli u samim bazama podataka.

6.2.2.1. PostgreSQL

Za programsku implementaciju PostgreSQL modela baze podataka korištena je Sequelize Node.js ORM biblioteka. Uz pomoć ove biblioteke možemo putem Javascript programskog koda definirati modele naših tablica, što podrazumijeva veze između tablice, atribute, tipove, indekse, itd. Također ovakve biblioteke pružaju mogućnost direktnog izvršavanja upita na bazi podataka, pa ukoliko nismo našli željenu funkcionalnost u programskoj apstrakciji biblioteke, uvijek možemo napisati SQL kod koji će ova biblioteka izvršiti za nas.

Pogledajmo kako izgleda definicija tablice korisnika uz pomoć Sequelize biblioteke:

```
module.exports = function (sequelize, DataTypes) {
  var User = sequelize.define('User', {
    name: DataTypes.STRING,

    email: {
```

[36] ORM (Object Relational Mapping) – Računalni sustav za konverziju relacijskih podataka u objekte objektno orijentiranih programskih jezika

```
      type: DataTypes.STRING,
      validate: {
        isEmail: true
      }
    },
    password: DataTypes.STRING,
    active: {
      type: DataTypes.BOOLEAN,
      defaultValue: false
    },
    country: DataTypes.STRING,
    city: DataTypes.STRING,
    address: DataTypes.STRING,
    activationHash: DataTypes.STRING,
    coins: {
      type: DataTypes.INTEGER,
      defaultValue: 0
    }
  });
  return User;
};
```

Metoda *define* prima dva argumenta. Prvi je ime tablice koju će Sequelize kreirati za nas, a drugi je definicija tablice koja će biti kreirana. *DataTypes* objekt sadrži dostupne tipove koje možemo koristiti pri definiciji naših modela, a možemo također definirati i opcije pri definiciji atributa, npr. Možemo lagano implementirati validaciju podataka. U ovom slučaju implementirana je validacija za provjeru valjanosti email atributa.

```
module.exports = function (sequelize, DataTypes) {
  var Role = sequelize.define('Role', {
    name: {
      type: DataTypes.STRING,
      validate: {
        notNull: true
      },
      unique: true
    }
```

```
  });
  return Role;
};
```

Na ovoj slici vidimo definiciju za tablicu Role. Ništa previše novo, ali je definicija prikazana jer ćemo za ove dvije tablice definirati vezu između njih, putem Sequelize biblioteke.

```
models.User.belongsTo(models.Role);
```

Ova linije je dovoljna Sequelize biblioteci da kreira vanjske ključeve za tablice i definira vezu između njih. Biblioteka također nudi eksplicitno definiranje imena ključa i ostalih atributa koje možemo normalno pridružiti prilikom ručnog kreiranja tablice. Ovime smo definirali da svaki korisnik ima jednu ulogu i da uloga ima više korisnika. Mogli smo to definirati i uz pomoć *hasMany* funkcije, npr.:

```
models.User.hasMany(models.ProductBid);
```

6.2.2.2. MongoDb

Slično kao i PostgreSQL, MongoDb ima biblioteke koje nude dodatnu programsku apstrakciju putem Node.js-a. Jedna od najpopularnijih takvih biblioteka je Mongoose.

Već smo rekli da se MongoDb i PostgreSQL temelje na različitim konceptima, pa ovdje umjesto da definiramo tablice, ovdje definiramo fleksibilne kolekcije dokumenata. Zadaća Mongoose biblioteke je pružiti mogućnost programskog definiranja ovakvih kolekcija. Također, biblioteka nam omogućava uvođenje određenih ograničenja nad kolekcijama i to na razini biblioteke, jer na razini MongoDb baze podataka to ne bi imalo smisla, pošto je svaki dokument u kolekciji valjan.

Također, ova biblioteka pokušava uvesti pojam veza između kolekcija, pa tako možemo dobiti osjećaj da zapravo radimo s relacijskom bazom podataka.

Veze između dokumenata implementirane su uz pomoć referenci na *_id* atribut nekog drugog dokumenta. Ova biblioteka će interno postaviti dodatne upite za nas, te nam omogućiti pristup dokumentu koji nije direktno sadržan u dokumentu roditelj. Ovo se čini jako dobro, no nažalost uz previše korištenja ove funkcionalnosti možemo se previše udaljiti od ideja na kojima se temelji MongoDb baza podataka i nakon nekog vremena ćemo zapravo imati relacijski model u bazi podataka koja nije za to namijenjena.

Definirajmo sada kolekciju proizvoda uz pomoć Mongoose biblioteke:

```
var ProductSchema = new Schema({
  name: {
    type: String,
```

```
    required: true
  },
  description: String,
  images: [String],
  price: {
    type: Number,
    required: true
  },
  currency: {
    type: String
  },
  active: Boolean,
  owner: {
    type: Number,
    required: true
  },
  category: {
    type: Schema.Types.ObjectId,
    required: true,
    ref: 'category'
  },
  counter: {
    type: Number,
    required: true,
    default: 0
  },
  wincount: {
    type: Number,
    required: true
  },
  soldTo: {
    type: Number,
    default: null
  }
});
```

Prvo što trebamo napraviti je definirati shemu za kolekciju. Slično kao i kod Sequelize biblioteke i ovdje smo u mogućnosti definirati tipove atributa koje će sadržavati dokument. Vidimo da možemo definirati i neka ograničenja koja su karakteristična za relacijske baze podataka, npr. *required*[37].

Ako malo bolje pogledamo atribut *category* vidimo da je on tipa *Objectid*, a to je karakterističan MongoDb tip kojim predstavljamo nešto kao primarni ključ u relacijskim bazama podataka. Za ovaj atribut definirali smo da je referenca na kolekciju *category*, znači sljedeći put kada postavimo MongoDb upit putem Mongoose biblioteke, biblioteka će automatski postaviti dodatni upit kojime će uključiti dokument iz kolekcije *category* u konačan rezultat.

6.2.2.3. Redis

Kod implementacija Javascript reprezentacije Redis podataka nije korišten nikakav ORM alat. Razlog za to dijelom je nedostatak dobrih ORM alata za Redis bazu podataka, dok s druge strane implementacija jednog takvog mini sustava za potrebe na ovom projektu je prilično jednostavna, pa iz toga razloga u ovom projektu je iskorištena jednostavna prilagođena inačica za rad s Redis bazom podataka.

```
Session.get = function (sid) {
  return Base.get(SESSION_KEY + sid).then(function (result) {
     return result ? JSON.parse(result) : null;
  });
};
Session.set = function (sid, value) {
  return Base.set(SESSION_KEY + sid, JSON.stringify(value));
};
```

Ovdje vidimo da imamo dvije glavne metode. To su već poznate Redis *get* i *set*. Svaka od njih prima *sid*, odnosno identifikator korisničke sesije koji se sprema kao ključ u Redis bazu podataka. *Get* metoda vraća Javascript reprezentaciju Redis zapisa i to u JSON formatu, dok metoda *set* zapisuje Javascript objekt u Redis bazu podataka.

Na slici vidimo da koristimo funkcionalnosti iz *Base* objekta. Pogledajmo sada dio koda koji stoji iza funkcionalnosti tog objekta.

```
var client = require('redis').createClient();
```

[37] Required – Ograničenje dostupno u Mongoose biblioteci koje osigurava da će svi atributi označeni ovim ograničenjem biti popunjeni

```
Base.get = function (key) {
  logger.debug('getting value on key: ' + key);
  return Q.ninvoke(client, 'get', key);
};
Base.set = function (key, value) {
  logger.debug('setting ' + value + ' for key ' + key);
  return Q.ninvoke(client, 'set', key, value);
};
```

Za komunikaciju s Redis bazom podataka korištena je *node-redis* biblioteka. Gotovo svaki poziv metode u sustavu, pa tako i ovaj se zapisuje preko log sustava u bazu podataka. Nakon zapisivanja log zapisa vidimo da pozivamo *get* i *set* Redis metode.

6.2.2.4. Cassandra

Nažalost još uvijek Node.js ekosustav je prilično siromašan kada su u pitanju *Big Table* baze podataka. Iz tog razloga još uvijek najčešći način definiranja modela i postavljanja upita je direktno putem CQL jezika. Pogledajmo sada CQL skriptu koja kreira *Keyspace bidding*, i *Column familly* objekt u bazi podataka u koji ćemo spremati log zapise.

```
DROP KEYSPACE bidding;
CREATE KEYSPACE bidding WITH REPLICATION = { 'class' :
'SimpleStrategy', 'replication_factor' : 1 };


CREATE TABLE IF NOT EXISTS bidding.log (
  date varchar,
  timeid timeuuid,
  id uuid,
  message varchar,
  ip varchar,
  time timestamp,
  user bigint,
  file varchar,
  line int,
  func varchar,
  level int,
  PRIMARY KEY (date, timeid, id)
) WITH CLUSTERING order by (timeid DESC);
```

Vidimo da imamo složeni primarni ključ. Razlog tomu je što sustav Cassandra ne dozvoljava postavljanje upita koji mogu vratiti velike količine podataka, jer se na ovaj način vrlo lako može srušiti cijeli sustav. Sustav Cassandra očekuje da smo precizno definirali na kojem ključu želimo vrijednosti i ukoliko želimo filtrirati ili sortirati podatke prema nekim kriterijima, onda svi *column-i* (atributi u relacijskim bazama), po kojima želimo filtrirati podatke moraju biti dio primarnog ključa, ili možemo kreirati indeks nad njima.

Nove zapise u sustav dodajemo putem sljedeće CQL naredbe.

```
INSERT INTO log (date, timeid, id, message, ip, time, user,
file, line, func, level) VALUES (?, now(), uuid(), ?, ?, ?, ?,
?, ?, ?, ?)
```

Jednostavan primjer upita kojim možemo dohvatiti podatke je:

```
SELECT * FROM LOG WHERE date=? AND timeid>? LIMIT ?
```

6.2.3. Najbolje od oba svijeta

U ovom poglavlju ćemo reći koje smo to prednosti navedenih sustava iskoristili u ovom projektu.

Vidjeli smo da sustav koristi „oba svijeta“, a to su relacijske i NoSQL baze podataka. Kakve su prednosti ovoga konkretno za ovu aplikaciju?

Prisjetimo se nekih funkcionalnosti aplikacije. Aplikacija je u mogućnosti da sprema podatke o korisnicima i korisnici su u mogućnosti da kupuju virtualne novce. Obe informacije u iznimno bitne za pravilno funkcioniranje cijelog sustava i u ovom slučaju ACID svojstva nisu opcija, nego potreba. S obzirom na to ovdje NoSQL baze podataka i nisu baš najsretniji izbor i umjesto toga relacijske baze podataka se nameću kao bolji izbor u ovom slučaju. Konkretno vidjeli smo da je izabran PostgreSQL sustav koji će nam osigurati da smo sigurni da su podaci konzistentni, da će biti trajno zapisani kada dobijemo obavijest od strane sustava o tome. Također podaci o korisnicima u ovakvoj aplikaciji nisu primarna informacija i ne trebamo se previše brinuti zbog određenog gubitka performansi (u odnosu na NoSQL sustave) zbog odabira ovog sustava.

Glavna svrha ovog sustava je pohranjivanje i pretraživanje podataka o proizvodima u sustavu. Za razliku od podataka o korisnicima i njihovim računima, kod podataka o proizvodima mogli bi reći da smo spremni prihvatiti mali rizik od mogućih neželjenih situacija koje mogu nastati u nekim neočekivanim okolnostima. Važno je napomenuti da ovaj rizik vrlo vjerojatno nikad neće postati problem, no ipak rizik postoji. Čak i u slučaju da se problem dogodi, odnosno izgubimo podatak, to neće biti informacija koja može biti katastrofalna za korisnika sustava.

Sada kada smo se upoznali s mogućim nedostacima ovog pristupa, trebamo nešto reći i o brojnim prednostima odabira NoSQL sustava, odnosno MongoDb baze podataka. MongoDb baza podataka kao i drugi NoSQL sustavi su odlični kandidati za horizontalno skaliranje sustava, odnosno jeftino i jednostavno skaliranje sustava u slučaju povećanja broja korisnika i opterećenja sustava. Također ova baza podataka se odlikuje osjetno većim brzinama u odnosu na relacijske baze podataka, što je velika prednost kod aplikacije koja je implementirana, jer se njezina primarna funkcionalnost svodi na različite operacije s proizvodima. Uzimajući u obzir činjenicu da se MongoDb temelji na fleksibilnom modelu podataka, moći ćemo jednostavno dodavati nove elemente u dokumente, što kao znamo nije slučaj kod relacijskih sustava, jer ovakve operacije mogu biti jako spore i rizične.

Jedna od vrlo bitnih funkcionalnosti koje se tiču samih proizvoda je njihovo pretraživanje. Već smo rekli da MongoDb podržava indekse i da je sustav prilično brz, no ipak primarna namjena sustava nije pretraživanje podataka. Ukoliko imamo potrebu za *full text search* značajkom, u MongoDb sustavu možemo koristiti npr. regularne izraze, no njihovo izvršavanje je jako sporo, stoga MongoDb sustav možda i nije najbolja opcija ukoliko imamo potrebu za naprednijim načinima pretraživanja velike količine podataka. Već smo spomenuli da postoje sustavi čija je primarna namjena indeksiranje i pretraživanje podataka velikim brzinama. U ovom projektu je korišten jedan od najpopularnijih sustava trenutno, a to je Elasticsearch sustav za indeksiranje i pretraživanje dokumenata. No postavlja se pitanje na koji način pohranjivati podatke u oba sustava bez da previše opterećujemo programsku logiku. Rekli smo da Elasticsearch sustav podržava tzv. *river* podsustave, čija je namjena da manipuliraju podacima u Elasticsearch sustavu. Korištenjem *river-a* i MongoDb *oplog-a* u koji se spremaju sve operacije koje su izvršene u ovoj bazi podataka, uz pomoć biblioteke *elasticsearch-river-mongodb* nije potrebno da pišemo bilo kakvu programsku logiku. Jedina stvar koja je potrebna je da konfiguriramo *river* i prilikom unosa, brisanja ili ažuriranje podataka u MongoDb bazi, podaci će također biti ažurirani i u Elasticsearch indeksu.

```
curl -XPUT 'http://localhost:9200/_river/mongodb/_meta' -d '{
  "type": "mongodb",
  "mongodb": {
    "db": "databasename",
    "collection": "products"
  },
  "index": {
    "name": "products",
```

```
      "type": "product"
    }
  }'
```

Ovdje možemo vidjeti primjer konfiguracije kakvu očekuje MongoDb *river*. Nakon ovoga podaci će u oba sustava biti sinkronizirani.

No, vratimo se sada prednostima korištenja ovog sustava. Jedna od funkcionalnosti sustava je i *autocomplete*, i ta funkcionalnost mora biti iznimno brza ukoliko želimo da korisničko iskustvo bude na prihvatljivoj razini. Ova funkcionalnost treba pretraživati imena proizvoda i srećom Elasticsearch sustav ima podsustave koji direktno podržavaju značajke kao što su *autocomplete*. Osim ovoga sustav podržava i *full text search*, što je također zanimljivo za vrstu aplikacije kakva je razvijena. Elasticsearch je također u mogućnosti rangirati dokumente prema tome koliko su dobro zadovoljili zahtjeve pretraživanja. Sustav također podržava replikaciju podataka i *sharding*, odnosno „cijepanje“ podataka i spremanje istih na više čvorova.

Sljedeća bitna funkcionalnost u sustavu su već opisane korisničke sesije. Rekli smo da su jako bitan dio gotovo svake aplikacije, pa tako i ove i odabir prave tehnologije za ovu funkcionalnost može biti jedan od bitnih faktora koji će utjecati na performanse sustava.

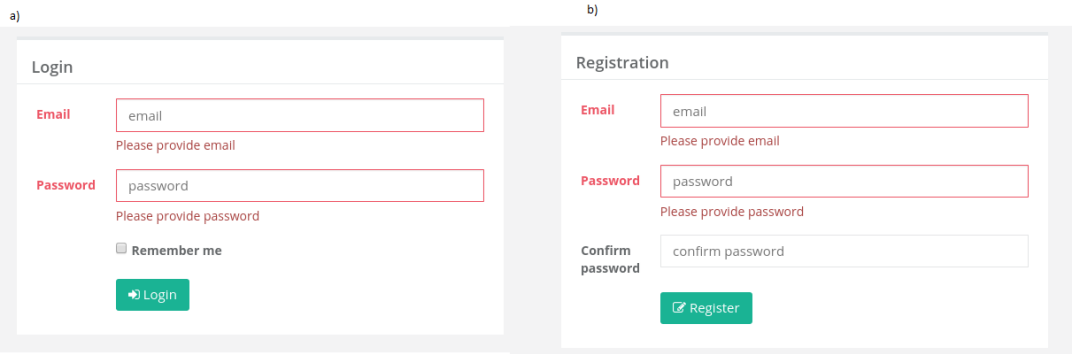
Redis baza podataka se odlikuje velikim brzinama kada podatke možemo prikazati u obliku ključ/vrijednost strukture, a upravo takva struktura je idealna za spremanja korisničkim sesija. Ključ će nam biti zapravo identifikator sesije koji ćemo spremiti u naš kolačić, a vrijednost će biti objekt koji je konstruiran prilikom prijave korisnika na sustav i predstavljati će korisničku sesiju. Dohvaćanje i modificiranje ovih podataka u Redis sustavu biti će iznimno brzo, no moramo biti svjesni jedne stvari: kada ažuriramo podatke o korisniku u PostgreSQL sustavu, moramo također ažurirati podatke i u Redis sustavu, što može donijeti dodatno opterećenje za programsku logiku, no upotrebom odgovarajuće programske apstrakcije možemo izbjeći ovaj problem. Ideja je da npr. kreiramo jedinstveno sučelje koje će interno upravljati podacima iz više izvora, dok će korisnik raditi sa sučeljem kao da upravlja sa samo jednim izvorom podataka.

Prozor kroz koji gledamo u našu aplikaciju su log zapisi i vrlo su bitni u produkcijskim okružjima u gotovo svim sustavima. Količina log zapisa, njihov format i ostale značajke mogu također bitno utjecati na performanse sustava. Prilikom dolaska jednog zahtjeva u sustav, aplikacija može generirati i stotine log zapisa, a kao što znamo kod velikih sustava u sekundi možemo imati i desetke tisuća zahtjeva. S razlogom vidimo da ovo može biti ozbiljan problem ukoliko naša baza podataka nema jako dobre performanse.

Već smo rekli dosta stvari o sustavu Cassandra i kod aplikacije koja je napravljena u sklopu ovog rada je korišten upravo taj sustav. Možemo si priuštiti jako mali rizik od gubitka podataka, no također zauzvrat dobijamo jako velike brzine unosa i pretraživanja podataka. Pošto su količine podataka koje će se pohranjivati u bazu iznimno velike, *big table* ideja na kojoj se baza podataka Cassandra temelji čini se kao odličan izbor za potrebe pohranjivanja log zapisa.

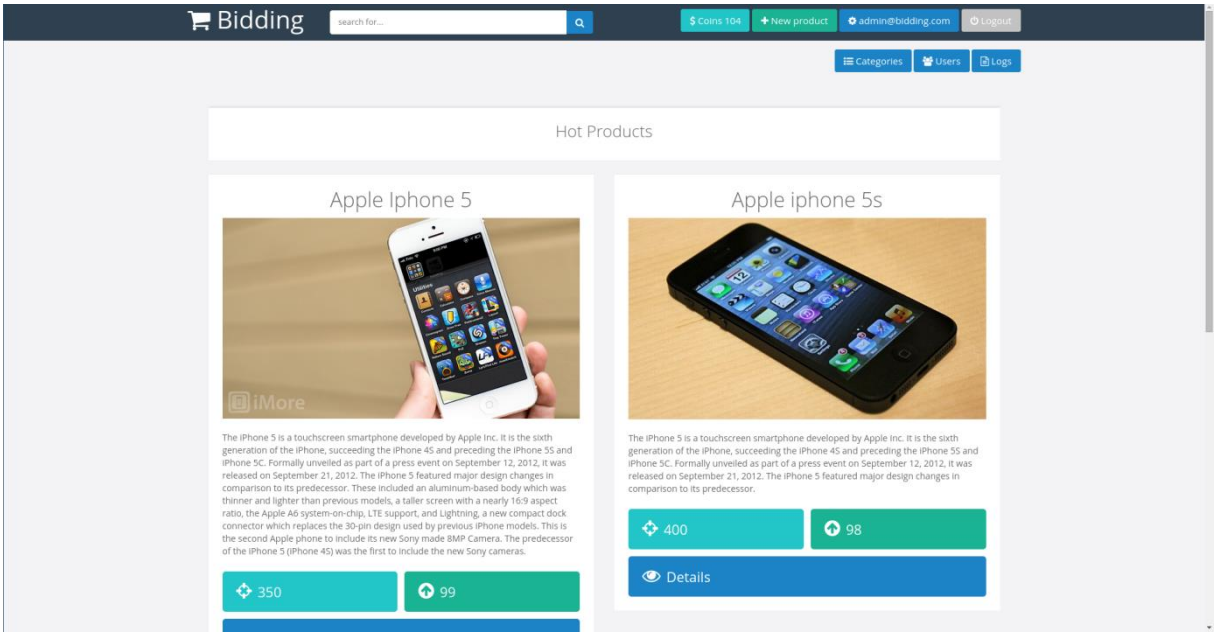
6.3. Web aplikacija

U ovom poglavlju ćemo vidjeti neke od formi i funkcionalnosti od kojih se sastoji razvijena klijentska strana ovog sustava.



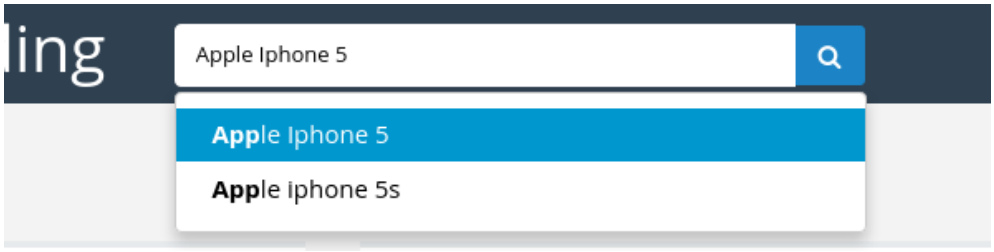
Slika 27. Forme za prijavu (a) i registraciju (b) na sustav

Na slikama vidimo forme za prijavu i registraciju na sustav. Kao što vidimo implementirana je validacija podataka u formama u stvarnom vremenu. Nakon što se registriramo na sustav, sustav će nam poslati aktivacijsku email poruku i nakon toga ćemo biti u mogućnosti prijaviti se u sustav. Početna stranica izgleda otprilike ovako:



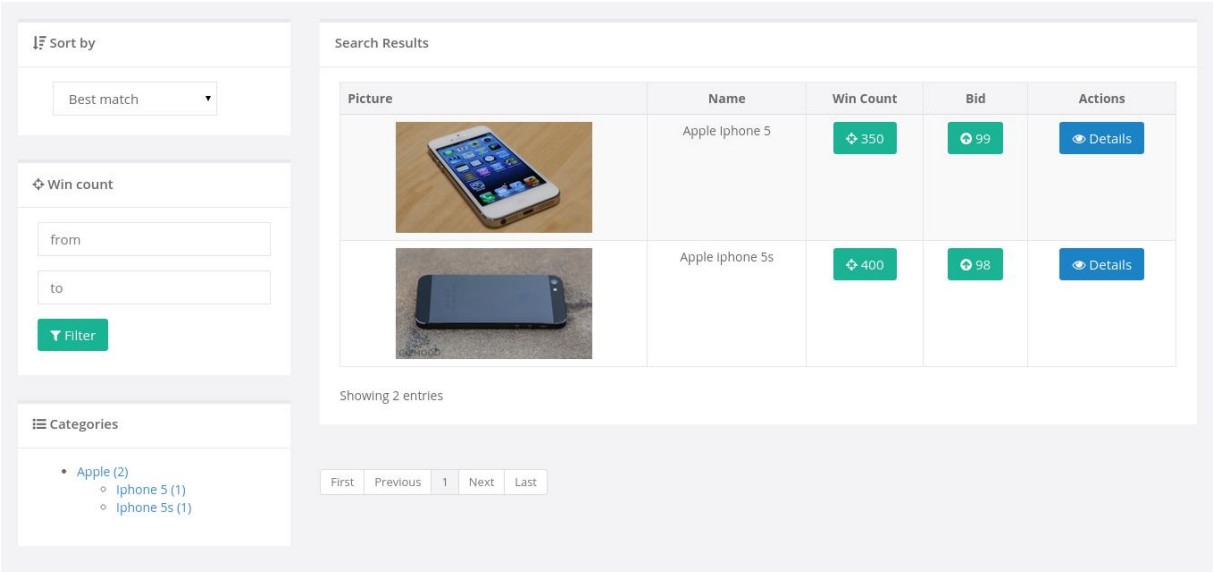
Slika 28. Početna stranica web aplikacije

Nakon biti ćemo u mogućnosti pretraživati proizvode.



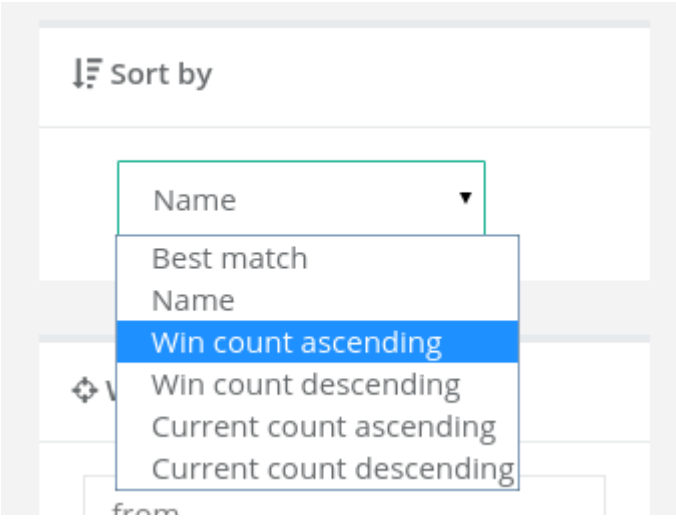
Slika 29. Pomoć pri pretraživanju proizvoda

Implementiran je i podsustav za predlaganje rezultata pri pretraživanju proizvoda, a na slici vidimo sustav u akciji. Sustav koristi Elasticsearch *autocomplete* mogućnosti.



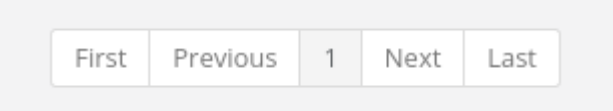
Slika 30. Rezultati pretraživanja

Ovdje vidimo i primjer stranice na kojoj možemo vidjeti rezultate pretraživanja. Osim pregleda samih proizvoda, imamo mogućnost koristiti i neke druge opcije.



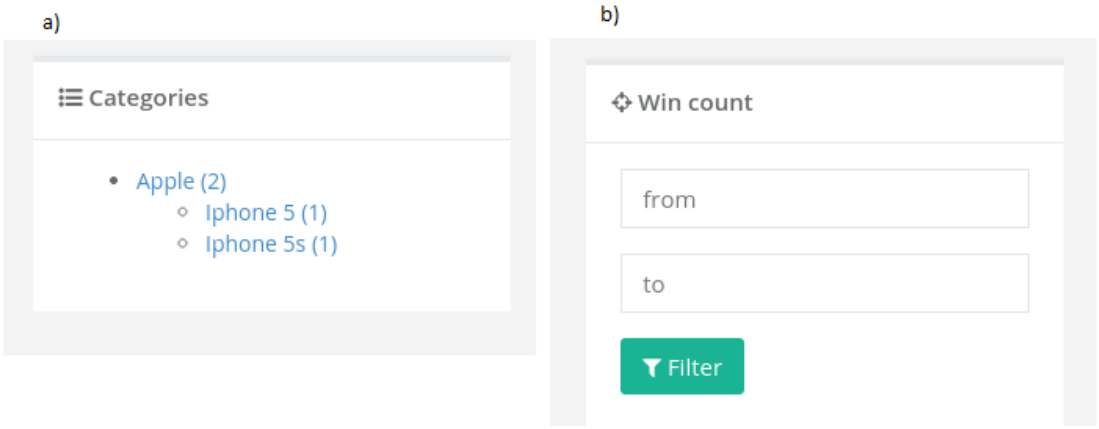
Slika 31. Sortiranje rezultata prema različitim kriterijima

Imamo mogućnost sortiranja proizvoda prema različitim kriterijima.



Slika 32. Straničenje rezultata

Ukoliko je količina pronađenih prizvoda velika, svi proizvodi neće biti prikazani, odnosno implementirano je straničenje. Straničenje je implementirano na poslužiteljskoj strani i koristi Elasticsearch mogućnosti straničenja.



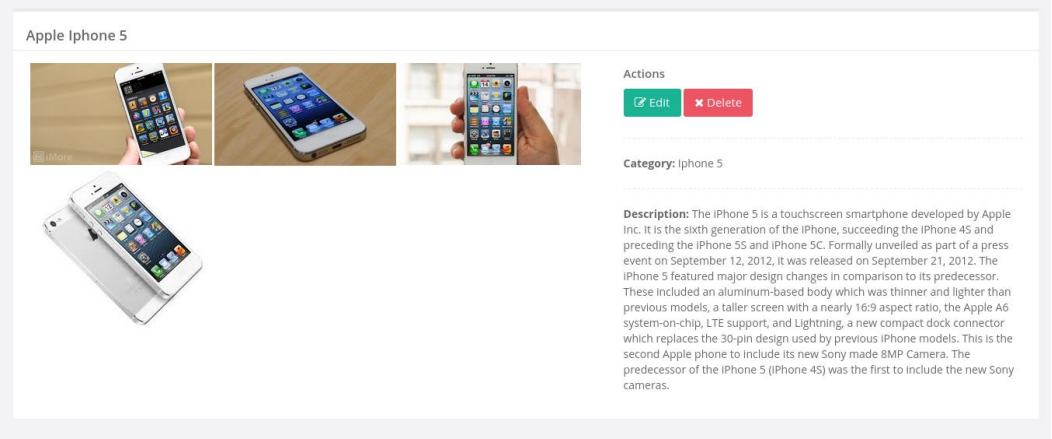
Slika 33. Filtriranje rezultata

Proizvodi su smješteni u neke kategorije i također smo u mogućnosti filtrirati rezultate prema kategorijama u koje oni pripadaju. Osim filtriranja rezultata prema kategorijama možemo filtrirati proizvode i prema virtualnoj cijeni proizvoda, odnosno *win count* broju.

Slika 34. Prikaz jednog proizvoda

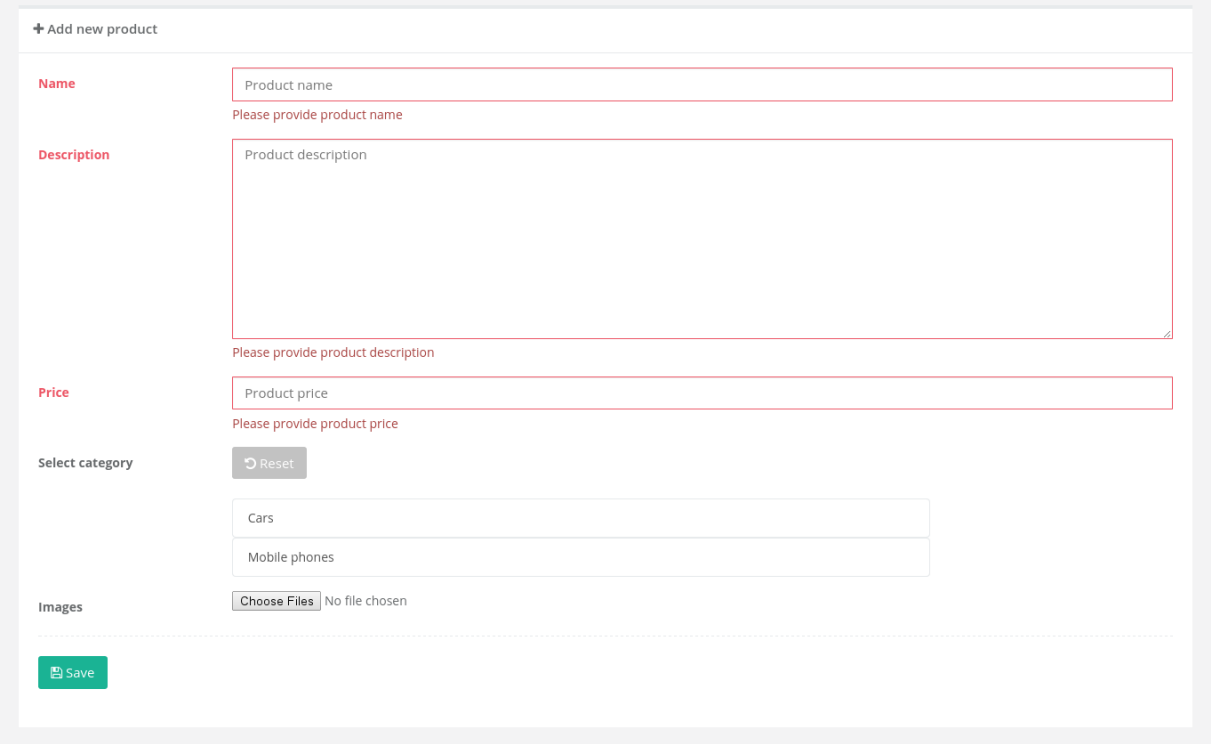
Svaki proizvod je prikazan u pravokutnom kontenjeru koji sadrži osnovne informacije o proizvodu. Gumb u koloni *Bid* šalje i prima informacije u stvarnom vremenu putem websocketa. Svaki klik na ovaj gumb smanjuje količinu virtualnog novca korisnika za određen iznos.

Također su implementirane provjere na klijentskoj i poslužiteljskoj strani o količini virtualnog novca koji su dostupni korisniku. Korisnik je također u mogućnosti kupiti virtualne novce kojima može ulagati novce za određeni proizvod.



Slika 35. Detalji proizvoda

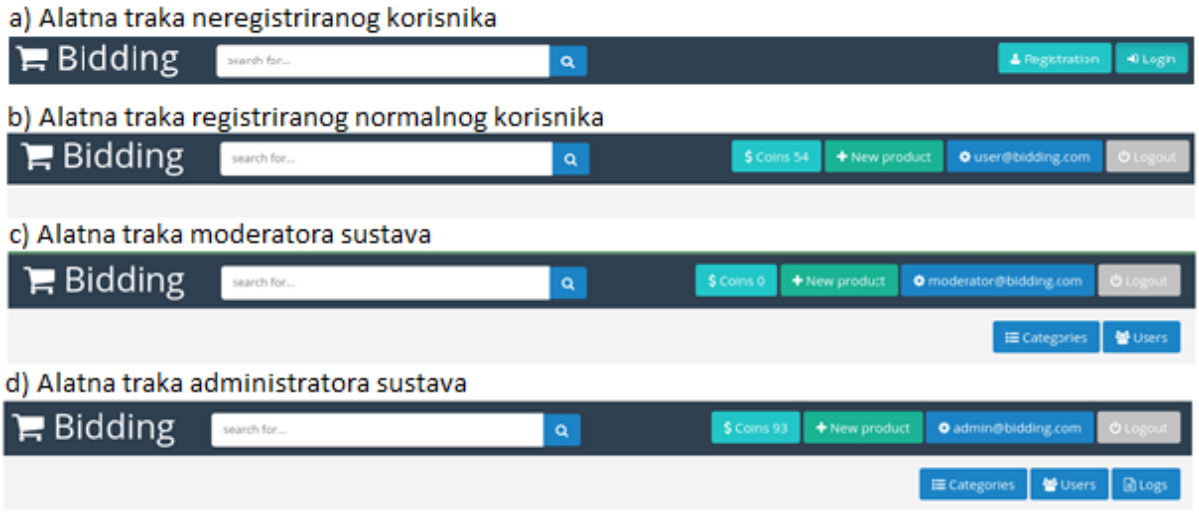
Vidimo da nam je dostupna stranica s detaljnijim opisom proizvoda i dodatnim opcijama koje su prikazane u slučaju da imamo dozvolu modificirati informacije o proizvodu.



Slika 36. Uređivanje proizvoda

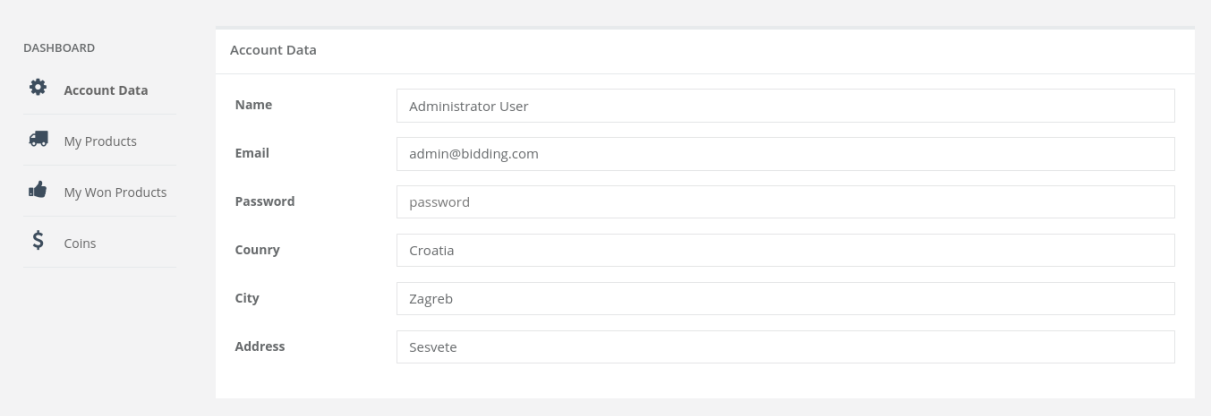
Ukoliko imamo dozvolu modificiranje proizvoda isti možemo obrisati ili mu izmijeniti neke informacije putem forme koja je prikazana na prethodnoj slici.

Kao što smo već rekli sustav ima podršku za autorizaciju korisnika. Jedan od primjera autorizacije na sustavu je i alatna traka sustava koja se mjenja ovisno o korisničkim pravima.



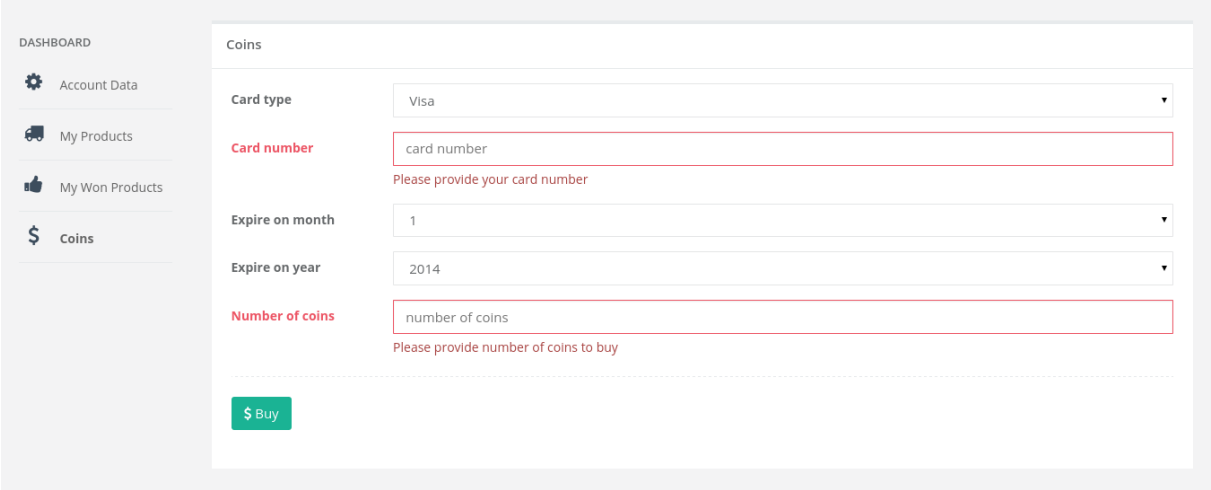
Slika 37. Alatne trake različitih tipova korisnika

Na prethodnim slikama vidimo alatne trake za četiri tipa korisnika koja su dostupna u ovom sustavu. Svaki registrirani korisnik ima mogućnost uređivanja i pregledavanja informacija o svome korisničkom profilu.



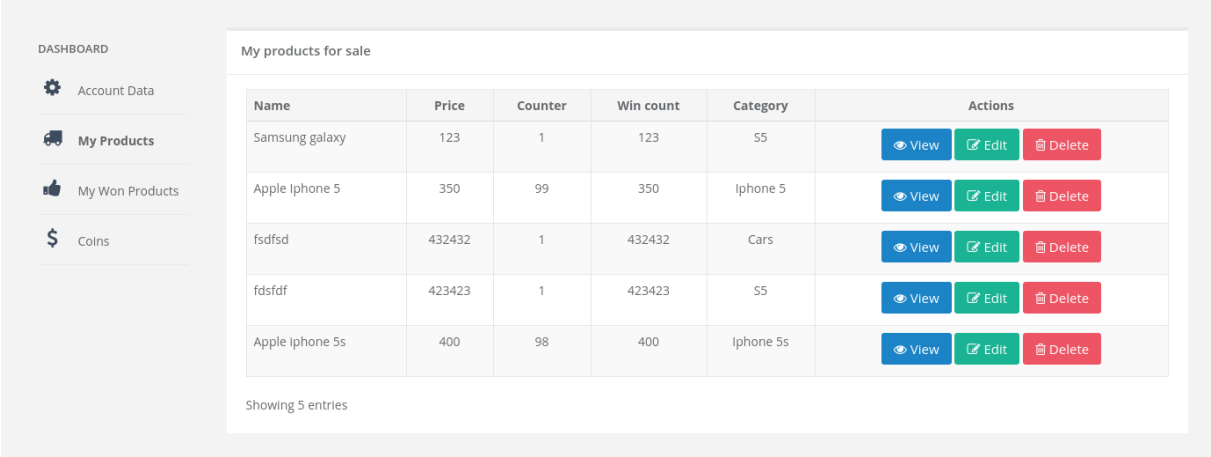
Slika 38. Korisnički račun

Kao što smo već rekli također je u mogućnosti kupovati virtualne novce.



Slika 39. Kupovina virtualnog novca

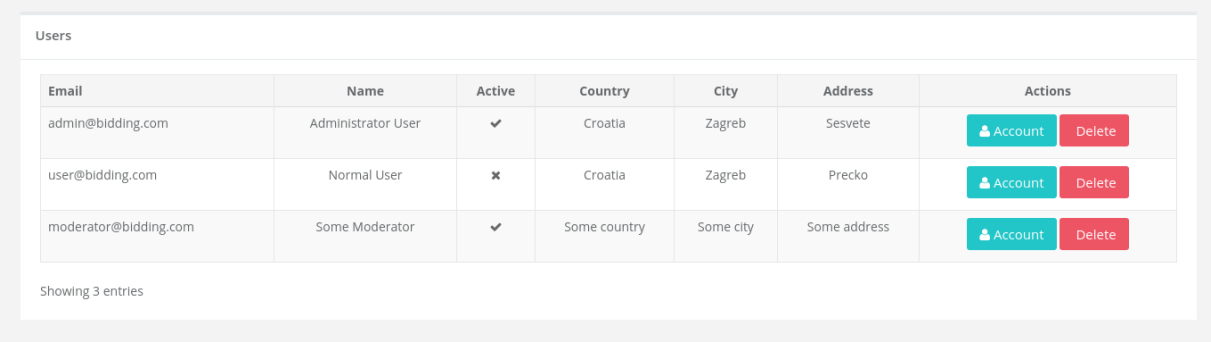
Vidimo da je za to potrebno pružiti uobičajene informacije o računu korisnika.



Slika 40. Pregledavanje liste kupljenih i objavljenih proizvoda

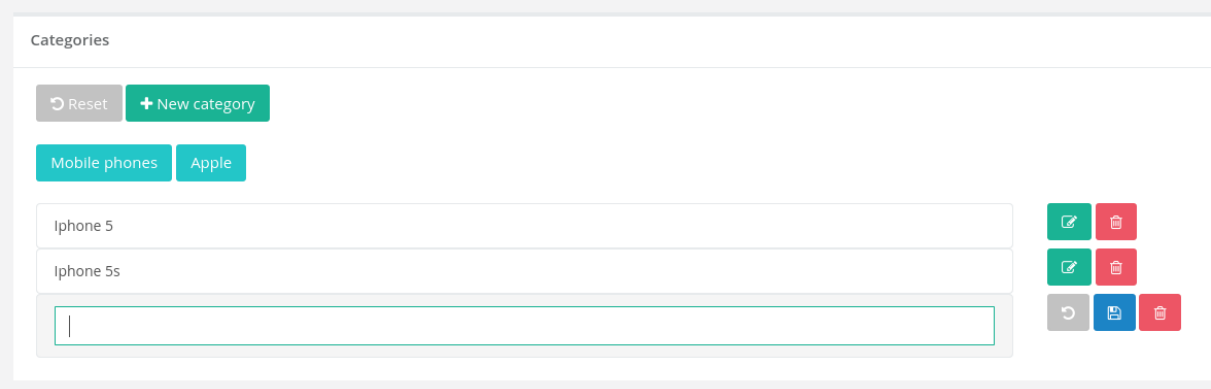
Korisnik je u mogućnosti pregledavati sve svoje proizvode. Prikazane su u listi preko koje može doći do različitih akcija nad tim porizvodima.

Ukoliko je prijavljeni korisnik moderator, u mogućnosti je uređivati i pregledavati proizvode drugih korisnika.



Slika 41. Pregledavanje korisnika u sustavu

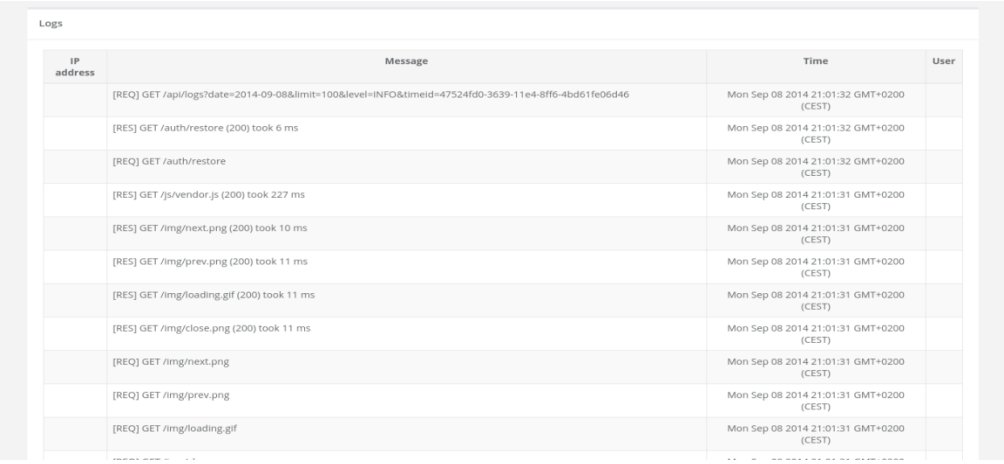
Na slici vidimo i listu korisnika preko koje je moderator u mogućnosti pristupiti profilima korisnika, te mijenjati njihove podatke.



Slika 42. Pregledavanja i uređivanje kategorija proizvoda

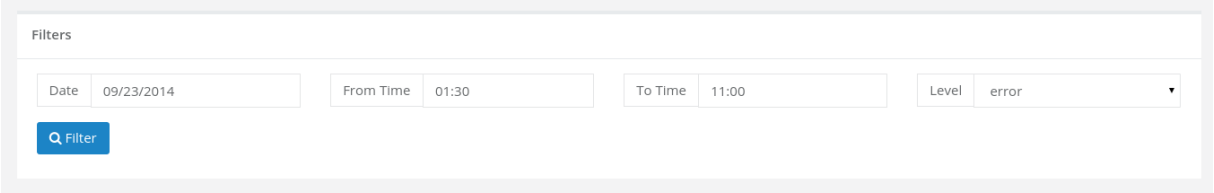
Ukoliko je korisnik moderator, u mogućnosti je upravljati dostupnim kategorijama proizvoda u sustavu.

Administrator sustava je jedini korisnik koji ima mogućnost pregledavanja log zapisa u sustavu.



Slika 43. Pregledavanje log zapisa u sustavu

Vidimo da je korisniku dostupan tablični prikaz log podataka koji dolaze iz baze podataka Cassandra.



Slika 44. Filtriranje log zapisa

Sustavi uglavnom generiraju velike količine log zapisa i jedno od poželjnih funkcionalnosti je mogućnost filtriranja istih po različitim kriterijima.

7. Zaključak

U ovom radu vidjeli smo prednosti i nedostatke relacijskih i NoSQL baza podataka. Kod NoSQL sustava pokazali smo više modela podataka na kojima se ovi sustavi temelje. Svaki model podataka ima neke svoje predstavnike, odnosno konkretne sustave za upravljanje bazama podataka, pa smo neke od njih iskoristili i pri realizaciji praktičnog dijela ovog diplomskog rada.

Vidjeli smo i neke ostale tehnologije koje se danas koriste pri razvoju modernih sustava. To su bili npr. *Node.js, Ember, Socket.io i* drugi. Sve su ovo manji dijelovi, odnosno Lego kockice koje trebamo poslagati na pravi način kako bi dobili jednu strukturu koja će biti stabilna u različitim uvjetima.

Razvijena je web aplikacija čija je namjena objavljivanje, pretraživanje i kupovanje različitih proizvoda. Svima je nama poznata platforma *eBay*. Sustav se može na prvu ruku činiti jednostavnim, no u pozadini se radi o iznimno kompleksnom sustavu, čija je zadaća da se nosi s tisućama različitih zahtjeva svake sekunde. Sustav kakav je razvijen u okviru ovog rada inspiriran je brojem korisnika koji ima npr. navedeni *eBay* i u skladu s time razvijena je arhitektura čija je ideja da bude skalabilna.

Kako bismo ostvarili visoku skalabilnost sustava vidjeli smo da smo integrirali i više baza podataka. Zbog pretpostavke o velikom broju korisnika na kojoj se temeljio razvijeni sustav, integriranje više različitih baza podataka bila je potreba kako bi sustav bio stabilan i s povećanjem korisnika. Vidjeli smo da smo uz NoSQL sustave iskoristili i jedan klasični relacijski sustav, odnosno PostgreSQL. Razlog tomu je kako smo rekli potreba za osiguranjem da nećemo imati određenih anomalija koje se tiču podataka o korisnicima, njihovim računima i slično. Kako bismo znali izabrati pravu tehnologiju, moramo znati njezine prednosti i nedostatke. Pokazali smo te prednosti i nedostatke za određene NoSQL sustave koji su iskorišteni za indeksiranje podataka, za spremanje log zapisa, za spremanje podataka o korisničkim sesijama, te za spremanje podataka o proizvodima i kategorijama u sustavu.

Uz pomoć poslužiteljske tehnologije Node.js bili smo u mogućnosti koristiti navedene sustave na jednostavan način i uvedena kompleksnost u odnosu na beneficije bila je prihvatljiva.

Naravno kompleksnost koju donose heterogene baze podataka u mnogo slučajeva nije prihvatljiva, te sustav neće imati nikakve, ili bolje rečeno, neće imati dovoljne beneficije u odnosu na uvedenu kompleksnost u sustavu. Uvedena kompleksnost se ne tiče samo

programske logike, nego i administracije takvih sustava. Vidjeli smo da je npr. ideja sustava Cassandra da radi na jednom distribuiranom sustavu, a kao što znamo arhitektura za ovakve sustave iziskuje dodatne financijske i vremenske resurse koje si mnoge organizacije nisu u mogućnosti priuštiti.

No, ipak postoje sustavi čije funkcioniranje ne bi bilo ovako dobro da nisu posegnuli za sustavima baza podataka koji se ne temelje isključivo na relacijskom modelu podataka. S obzirom na ovo i na druge navedene činjenice možemo zaključiti da izbor ovakve arhitekture može biti jako dobar i nužan izbor ili jako loš izbor koji će unijeti nepotrebnu kompleksnost u sustav. Ovo naravno ovisi o zahtjevima aplikacije i predviđanjima u kakvim će se okolinama naći aplikacija u budućnosti.

8. Literatura

1. Ali, M. G. (2009). *Object-Oriented Approach for Integration of Heterogeneous Databases in a Multidatabase System and Local Schemas Modifications Propagation.* International Journal of Computer Science and Information Security .
2. Banker, K. (2012). *MongoDb in action.* Shelter Island, NY: Manning Publications.
3. Brooklyn, C. (2014). *SQL Intro*. Preuzeto od http://theparticle.com/cs/bc/dbsys/sql.pdf
4. Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., & Widom, J. (bez datuma). *The TSIMMIS Pro ject:Integration of Heterogeneous Information Sources.* Stanford: Department of Computer Science, Stanford University.
5. Elasticsearch. (2014). *Službena dokumentacija sustava Elasticsearch*. Preuzeto od http://www.elasticsearch.org/guide/
6. Goodrich, M. T., & Tamassia, R. (2005). *Directed Graphs.* Preuzeto od http://ww0.java4.datastructures.net/handouts/Digraphs.pdf
7. Maleković, M. (2012). *FOI: Nastavni materijali kolegija Baze podataka 2*. Preuzeto od http://elfarchive.foi.hr/11_12/mod/resource/view.php?id=1287
8. MongoDb. (2014). *Službena dokumentacija sustava MongoDb*. Preuzeto od http://docs.mongodb.org/manual/
9. Neeraj, N. (2013). *Mastering Apache Cassandra.* Birmingham, UK: Packt Publishing.
10. Node.js. (2014). *Službena dokumentacija sustava Node.js*. Preuzeto od http://nodejs.org/api/
11. PostgreSQL. (2014). *Službena dokumentacija sustava PostgreSQL*. Preuzeto od http://www.postgresql.org/docs/manuals/
12. Rajeswari, V., & Varughese, D. D. (2009). *HETEROGENEOUS DATABASE INTEGRATION FOR WEB APPLICATIONS.* International Journal on Computer Science and Engineering.
13. Redis. (2014). *Službena dokumentacija sustava Redis*. Preuzeto od http://redis.io/documentation
14. Sadalage, P. J., & Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Crawfordsville, Indiana: Pearson Education.
15. Schatten, M., & Ivković, K. (2012). Regular Path Expression for Querying Semistructured Data -. *Central European Conference on Information and Intelligent Systems*.
16. Shaw, P. (2013). *Postgres Succinctly.* Morrisville, NC: Syncfusion Inc.
17. Simovici, D. (2011). *Introduction to Database Concepts*. Preuzeto od http://www.cs.umb.edu/cs630/hd1.pdf
18. Stephens, R. (2002). *Teach Yourself SQL in 21 Days.* Macmillan Computer Publishing.
19. Ullman, J. D. (1994). *Foundations of Computer Science: C Edition (Principles of Computer Science Series).* Lawrence C. Paulson.
20. Weber, S. (bez datuma). *NoSQL Databases*. Preuzeto od http://wiki.hsr.ch/Datenbanken/files/Weber_NoSQL_Paper.pdf