UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Mirjana Kostić

Razvoj veb aplikacija pomoću MEAN steka

Master rad

Beograd, 2018

Sadržaj

1 Uvod **3**

2 MEAN stek **4**

3 Pregled tehnologija MEAN steka **5**

3.1 JavaScript 5
3.2 Node.js 8
3.2.1 Node.js upravljač modulima i package.json 8
3.2.2 Pravljenje modula 11
3.2.3 Server Node.js i događaji 12
3.2.4 Upravljanje ulaznim i izlaznim podacima 13
3.2.5 HTTP/HTTPs 15
3.3 Modul express 19
3.3.1 Express generator 19
3.3.2 Rutiranje i REST API 20
3.3.3 Express zahtev i odgovor 25
3.3.4 Posrednički dodaci 25
3.4 MongoDB 28
3.4.1 Administriranje baze podataka MongoDB 30
3.4.2 Rad nad dokumentima 34
3.4.3 MongoDB u Node.js okruženju 40
3.5 Angular radni okvir 45
3.5.1 Arhitektura Angulara 45
3.5.2 Moduli 46
3.5.3 Komponente 47
3.5.4 Šabloni 49
3.5.5 Povezivanje podataka 49
3.5.6 Direktive 50
3.5.7 Servisi i umetanje zavisnosti 51
3.5.8 Obećanja i Posmatrači 52
3.5.9 Servis HTTP Client 53
3.5.10 Modul za rutiranje 54

4 Detalji implementacije aplikacije Travelerko **56**

4.1 Opis aplikacije Travelerko 56
4.2 Arhitektura aplikacije 58
4.3 Baza podataka 59

4.4 Serverska aplikacija 63
4.5 Klijentska aplikacija 74
4.5.1 Veb aplikacija 74
4.5.2 Mobilna aplikacija 79

5 Zaključak **81**

6 Literatura i korisni sajtovi **82**

7 Dodatak **83**

1 Uvod

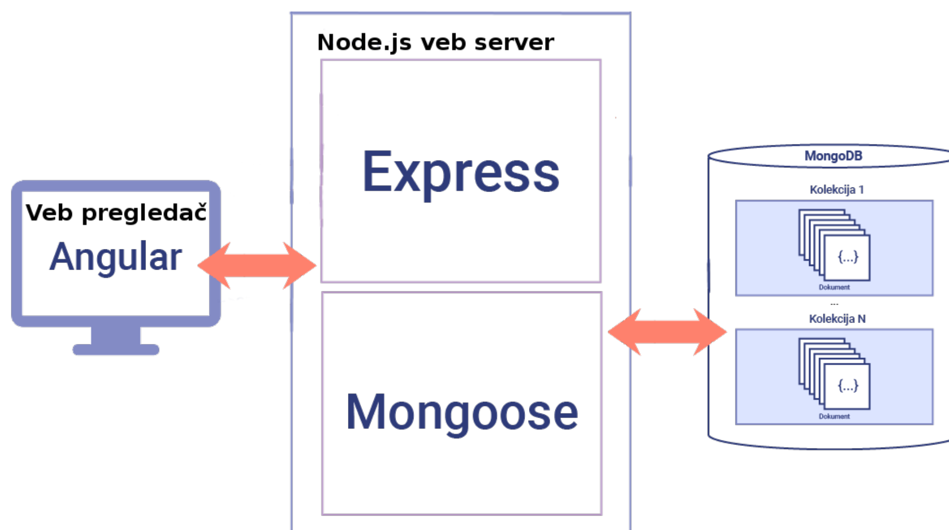
U poslednjih desetak godina razvoj i primena tehnologije doneo je različite pogodnosti i nametnuo potrebu korišćenja interneta i veb aplikacija u različitim domenima života. Znatno uvećano korišćenje interneta i internet zasnovanih aplikacija uticalo je, kao povratna sprega, i na proces razvoja aplikacija. Kao jedan od najbitnijih zadataka koje programer ima prilikom izrade veb aplikacija jeste odabir odgovarajućih tehnologija i alata za razvoj kako bi se ispunili glavni zahtevi veb aplikacija kao što su brzina, stabilnost, bezbednost, dinamičnost, i dizajn koji je prilagođen korisnicima. Jedan skup tehnologija koji se može koristiti za pravljenje takvih veb aplikacija je i *MEAN* (*MongoDB*[1], *express*[2], *Angular*[3], *Node.js*[4]) stek. Njegov naziv je akronim koji označava skup od četiri tehnologije čija je osnova *JavaScript*, a koje se razlikuju po nameni. Svaka od ovih tehnologija se može koristiti odvojeno ili u kombinaciji sa nekim drugim tehnologijama, ali zajedno one čine osnovu novog pravca u pravljenju veb aplikacija. *Node.js* ima široku primenu, ali se uglavnom koristi za implementaciju veb servera, uz pomoć modula *express*. *MongoDB* je *NoSQL* baza podataka, a *Angular* služi za implementaciju vizuelnog dela aplikacije.

U radu je dat prikaz tehnika razvoja veb aplikacija korišćenjem *MEAN* steka i sastoji se iz nekoliko poglavlja. U drugom poglavlju je opisan *MEAN* stek kao celina. U trećem poglavlju ta celina je razbijena na zasebne delove gde je svaki deo posvećen jednoj od tehnologija *MEAN* steka. Jedan deo trećeg poglavlja je posvećen samom *JavaScriptu* zato što on predstavlja osnovu tehnologija *MEAN* steka. Delovi trećeg poglavlja se nadovezuju logički jedan na drugi a ne prateći sam akronim. U četvrtom poglavlju je dat opis veb aplikacije koja će se napraviti korišćenjem tehnologija i principa opisanih u trećem poglavlju, uz opisanu implementaciju veb aplikacije. Pored pojašnjenja implementacije, uputstvo za pribavljanje i pokretanje cele aplikacije je dostavljeno kao dodatak, koji se nalazi na kraju rada.

[1] https://www.mongodb.com
[2] https://expressjs.com
[3] https://angular.io
[4] https://nodejs.org

2 MEAN stek

Rečeno je da u tehnologije *MEAN* steka spadaju *Node.js*, *express*, *MongoDB* i *Angular* ali se pored njih često koristi i neki od drajvera koji služi da omogući upotrebu baze podataka *MongoDB* u *Node.js* aplikaciji. Većina ovih tehnologija se može koristiti samostalno. Veb aplikacija se može napraviti samo korišćenjem *Node.js*-a, koji bi imao ulogu i servera i koji bi služio za prikaz *HTML* stranica. Međutim, *Node.js* u kontekstu *MEAN* steka predstavlja osnovnu platformu za razvoj koju koriste ostale tehnologije steka. U *Node.js* platformi se inicijalizuje server pomoću modula *express*, uz korišćenje drajvera *Mongoose* za povezivanje sa bazom podataka. Server se kreira tako da prima *HTTP* zahteve, obrađuje ih i vraća *HTTP* odgovor. *HTTP* zahteve šalje *Angular* aplikacija, u zavisnosti od akcije korisnika u njoj. Baza podataka *MongoDB* služi da čuva informacije koje se koriste u veb aplikaciji čime se obezbeđuje dinamičnost upotrebe. Kako su ovo zapravo četiri različite tehnologije organizovane u jednu celinu, neophodno je prvo upoznati se sa svakom tehnologijom zasebno.



Slika 1: *Tehnologije MEAN steka*

3 Pregled tehnologija MEAN steka

3.1 JavaScript

JavaScript je skript jezik, čiji je kompajler ugrađen u veb pregledače, kako bi omogućio programerima da dinamički menjaju sadržaj veb strana bez podrške veb servera. On se u potpunosti izvršava na računaru klijenta.

JavaScript je razvio Brendan Ajk (*eng. Brendan Eich*) u maju 1995. godine za potrebe pregledača Netskejp Navigator (*eng. Netscape Navigator*). Kako je popularnost jezika rasla tako je sadržaj veb aplikacija postajao dinamičniji. Evolucija *JavaScripta* je ujedno i evolucija veb stranica u veb aplikacije. Kako se *JavaScript* razvijao, tako su se razvijali alati i biblioteke koji su ga koristili. Veb pregledači su takodje morali da prate razvoj i da se prilagođavaju novim zahtevima kako bi obezbedili što bolje iskustvo korisnicama. Tako su se razvili i *JavaScript* mehanizmi (*eng. JavaScript Engine*) a mehanizam na koji posebno treba obratiti pažnju je V8 mehanizam koji je urađen u pregledač Gugl Hrom (*eng. Google Chrome V8 Engine*).

V8 mehanizam je prvi put objavljen 2008. godine. To je *JavaScript JIT* (*eng. Just In Time*) kompajler i nastao je kako bi poboljšao performanse *JavaScripta* unutar pregledača. Iako V8 mehanizam koristi više niti tokom rada (za kompajliranje, prikupljanje otpadaka, ...), *JavaScript* koristi samo jednu nit i samim tim *JavaScript* se izvršava liniju po liniju. Dva jako bitna dela V8 mehanizma čine hip na kome se smeštaju tekuće promenljive i stek koji prati stanje programa koji se trenutno izvršava. Međutim V8 menahizam ne radi sam. Pored njega bitni delovi su i veb API (u kome su implementirane funkcionalnosti koje se često koriste poput AJAX-a, DOM-a ...), povratni red (*eng. callback queue*) i petlja za događaje (*eng. event loop*).

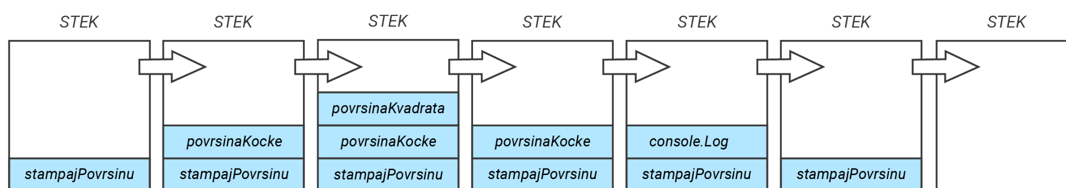
U primeru 1 poziva se funkcija `stampajPovrsinu()` kojoj se prosleđuje argument 5 kao dužina stranice kvadrata čiju površinu funkcija računa. Inicijalno stek je prazan. Kako se pročita poziv funkcije `stampajPovrsinu()` tako se on smešta na stek. Ta funkcija kasnije poziva funkciju `povrsinaKocke()` koja se isto smešta na stek. Funkcija `povrsinaKocke()` u sebi poziva funkciju `povrsinaKvadrata()` koja se takođe smešta na stek. Funkcija `povrsinaKvadrata()` vraća svoj rezultat funkciji `povrsinaKocke()` i uklanja se sa steka jer se smatra završenom. Isto se i funkcija `povrsinaKocke()` uklanja sa steka kada vrati svoj rezultat funkciji `stampajPovrsinu()`. Nakon dodeljivanja povratne vrednosti funkcije `povrsinaKocke()` promenljivoj povrsina, ta vrednost se ispisuje pomoću funkcije `console.log()` koja se postavlja na vrh steka, i nakon završetna rada, odnosno ispisa, skida sa steka. Kada se i ta funkcija završi, može se reći da se završila i funkcija `stampajPovrsinu()` koja se skida

sa steka, čime stek ostaje prazan.

```
function povrsinaKvadrata(a) {return a*a;}
function povrsinaKocke(a) {return 6*povrsinaKvadrata(a);}
function stampajPovrsinu(a) { var povrsina=povrsinaKocke(a);
console.log(povrsina);
}
stampajPovrsinu(5);
```

Primer 1: *Primer funkcije u JavaScriptu*

Dakle kako se neka funkcija pozove, ona se stavlja na stek i kako se njeno izvršavanje završi, ona se sa steka sklanja. Na slici 2 vidi se da se funkcija `stampajPovrsinu()` sa steka uklanja tek kada se završi izvršavanje svih pomoćnih funkcija koje su pozvane tokom izvršavanja.



Slika 2: *Stanje steka tokom izvršavanja funkcije* stampajPovrsinu()

U primeru 2 se koristi asinhroni poziv pomoću funkcije `setTimeout()`. Ako se prati prethodni primer, prva pomisao je da će se u konzoli ispisati prvo reč "zdravo", pa nakon 5 sekundi reč "svima" i nakon toga "u komisiji".

```
console.log("zdravo");
setTimeout(function(){console.log("svima");},5000);
console.log("u komisiji");
```

Primer 2: *Primer pozivanja asinhrone funkcije*

Slika 3 predstavlja prikaz konzole veb pregledača nakon izvršavanja primera 2. Iako se *JavaScript* izvršava striktno sinhrono, na slici 3 se vidi da nije uvek tako. Prethodno je rečeno da mehanizam V8 ne radi sam a u primeru 2 se vidi kakva je uloga veb API-ja, povratnog reda i petlje događaja.

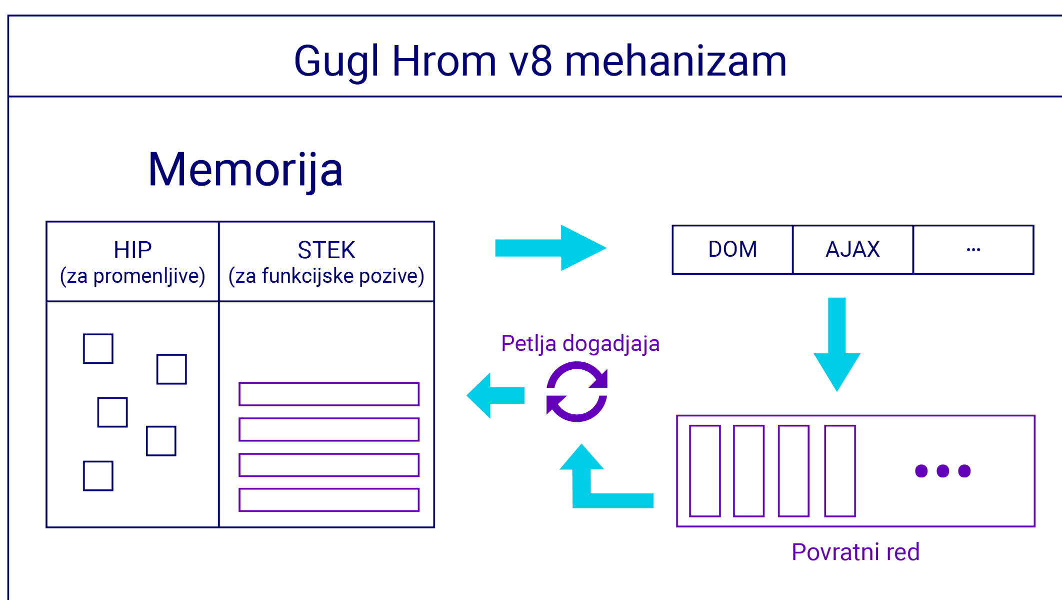
`setTimeout()` je funkcija koja se nalazi unutar veb API-ja koji obezbeđuje veb pregledač i nije sastavni deo naredbi *JavaScripta*. To je tajmer i ima zadatak da za prosleđeni vremenski period (u primeru 2 je podešen na 5 sekundi) izvrši prosleđenu funkciju (u primeru 2 je to anonimna funkcija koja pomoću `console.log()` funkcije

ispisuje reč “svima”). Izvršavanje je normalno do dela kada se `setTimeout()` nađe na steku. Tada, veb API preuzima `setTimeout()` i u tom trenutku stek smatra da je ta funkcija završena, istu skida sa steka i nastavlja dalje sa radom. Za to vreme, veb API obrađuje `setTimeout()` funkciju i nakon izvršenih 5 sekundi čekanja njenu povratnu funkciju prebacuje u povratni red. Petlja događaja ima zadatak da, ukoliko je stek prazan, proveri da li postoji neka funkcija u povratnom redu i ukoliko postoji, prebaci je nazad na stek kako bi se izvršila.

```
zdravo
u komisiji
svima
>
```

Slika 3: *Izgled konzole nakon izvršavanja primera koda 2*

Isti rezultat bi se verovatno dobio i kada bi se u primeru 2 umesto 5 sekundi prosledilo 0 sekundi je će veb API odmah obraditi `setTimeout()` funkciju i poslati je u povratni red, ali će petlja događaja svakako da čeka da se stek oslobodi kako bi povratnu funkciju iz povratnog reda prebacila na stek, tako da vreme prosleđeno tajmeru je zapravo minimalno vreme čekanja pre nego što se izvrši prosleđena funkcija. Isto se dešava i sa AJAX zahtevima bez kojih je danas teško zamisliti izradu veb aplikacija.



Slika 4: *Arhitektura V8 mehanizma*

3.2 Node.js

Node.js je platforma koja je nastala 2009. godine. Razvio ga je Rajan Dal (*eng. Ryan Dahl*). Ideja je bila da se premoste razlike između tada postojećih tehnologija za izradu klijentskih i serverskih delova veb aplikacija. Kako svi veb pregledači raspoznaju samo *HTML*, *CSS* i *JavaScript*, a ni jedna od tih tehnologija ne obezbeđuje pristup bazama podataka kao ni serverski deo aplikacije, bilo je neophodno za sve obrade na klijentskoj strani koristiti *JavaScript* a za serversku stranu ASP ili *PHP* ili nesto slično. To znači da su konstatno potrebne dve sintakse, u okviru istog dela programa. Mogućnost za nastanak platforme *Node.js* je ostvarena tek kada se pojavio V8 mehanizam koji je zahvaljujući svojoj politici otvorenog koda postao jezgro platfome *Node.js*. Oko V8 mehanizma su "obmotane" dodatne funkcionalnosti i tako je napravljeno serversko okruženje koje odgovara i klijentskom okruženju u pregledaču, odnosno koristi sve postojeće *JavaScript* tehnologije. Ono što čini *Node.js* jako zanimljivim je njegova modularnost. Njegovu osnovu čini skup modula koji se uključuju po potrebi, a pored osnovnih modula postoji zajednica programera koja svakog dana pravi nove module za različite namene. To je izuzetno značajno i daje veliku prednost ovoj tehnologiji u odnosu na slične, jer ukoliko postoji potreba da se proširi *Node.js* aplikacija nekom novom funkcionalnošću, velika je verovatnoća da je neko već napravio i objavio paket za baš tu funkcionalnost. Velika prednost pisanja koda u *JavaScriptu* je to što je on već ugrađen u pregledače tako da se kod se može koristiti i u serverskoj i u klijentskoj aplikaciji. Međutim, sve što ima prednosti ima i svoje mane. *Node.js* je relativno mlad, i iako ima veliku zajednicu korisnika, on nije dovoljno testiran i korišćen. Pored toga, *Node.js* zahteva malo drugačiji pristup pisanju koda i generalno načinu razmišljanja jer kako bi se maksimalno iskoristile njegove mogućnosti i smanjila verovatnoća za grešku tj. za neočekivano ponašanje aplikacije potrebno je organizovati kod tako da odgovara asinhronom načinu rada.

3.2.1 Node.js upravljač modulima i package.json

Prethodno je rečeno da je *Node.js* modularan, odnosno lako proširiv raznim modulima/paketima. *Node.js* ima zajednicu koja objavljuje module i koja na raspolaganju ima privatni i javni registar modula. Registar modula je zapravo jedna velika "biblioteka" modula *Node.js* i javno je dostupna. Na stranici registra mogu se pretraživati moduli ali isto tako ona služi kako bi se moduli i objavili. Kako bi registar modula mogao da se koristi prilikom pravljenja aplikacije neophodno je da se poseduje alat koji se zove *Node.js* upravljač modulima (*eng. node package manager*). Ovaj alat nije potrebno zasebno instalirati jer se on automatski instalira prilikom instalacije platforme *Node.js*. On, pored drugih korisnih načina upotrebe,

predstavlja vezu izmedju aplikacije koja se pravi i registra modula. Pomoću njega se sa lakoćom instaliraju i uklanjaju moduli, služi i za objavljivanje napravljenog modula u registar modula i instaliranje neobjavljenih modula, odnosno instaliranje modula koji se ne nalazi u registru. Komande pomoću kojih se *Node.js* upravljač modulima koristi uglavnom imaju oblik `npm operacija -opcije ime_modula`, a neke od operacija koje se često koriste su:

- **search** - pretražuje module u registru
- **install** - služi za instalaciju modula. Može se koristiti i skraćenica **i**
- **uninstall** - služi za brisanje modula iz aplikacije
- **view** - prikazuje informacije o prosleđenom modulu

Svi moduli koji se instaliraju pomoću *Node.js* upravljača modulima se instaliraju u direktorijumu `node_modules`.

package.json *package.json* je tekstualni fajl u JSON formatu i on je obavezan za svaki modul, uključujući i koren aplikacije koja je i sama jedan veliki modul. Pomoću njega se bliže definišu aplikacija i moduli unutar aplikacije. Obavezna svojstva koje mora da sadrži su:

- ime (*eng. name*) - jedinstveni naziv modula pomoću kog moze da se registruje
- verzija (*eng. version*) - verzija modula

Ostali atributi su:

- autor (*eng. author*) - svojstvo čuva naziv autora modula
- opis (*eng. description*) - svojstvo čuva opis modula koji je koristan ukoliko želimo da modul objavimo
- zavisnosti (*eng. dependencies*) - ovo svojstvo čuva listu svih instaliranih modula i njihovih podmodula koje koriste. Lista sadrži ime modula i njegovu verziju, kao i ostale module od kojih je on zavisan. Prava primena ovog svojstva je da, ukoliko se iz nekog razloga izmešta aplikacija ili se pravi iz početka, kako bi se svi moduli instalirali pomoću jedne komande, dovoljno je da se prepiše *package.json* fajl i pokrene `npm install` komanda. Ovom komandom će se instalirati svi moduli navedeni pod ovom sekcijom.

- zavisnosti u razvoju (*eng. devDependencies*) - ovo svojstvo se koristi kada je aplikacija u razvojnom (*eng. development*) okruženju, tj. ako globalna promenljiva *NODE_ENV* ima vrednost *development* i neki modul se instalirao korišćenjem *-D* zastavice. Prilikom pokretanja `npm install` komande instaliraće se svi moduli koji su ovde navedeni, u suprotnom, ukoliko je aplikacija u produkcionom (*eng. production*) okruženju, ni jedan od ovih modula neće biti instaliran
- glavni skript (*eng. main*) - ovim svojstvom se navodi glavni skript koji se pokreće, predefinisano *index.js*

U primeru 3 se može videti kako *package.json* fajl može da izgleda.

Instaliranje novih modula Novi modul se iz registra ili neobjavljen instalira pomoću komande `npm install`. Modul iz registra se instalira tako što se prilikom pozivanja `npm install` komande prosleđuje njegovo jedinstveno ime iz registra, dok se neobjavljeni modul instalira tako što se `npm install` komandi prosledi putanja do modula. Ukoliko se nakon `install` izostavi ime modula ili putanja do istog, instaliraće se svi moduli koji su navedeni pod svojstvom zavisnosti (i u nekim slučajevima zavisnosti u razvoju) fajla *package.json*.

Da bi se istalirala određena verzija modula, uz ime modula pridružuje se karakter @ i željena verzija koja se sastoji iz tri broja odvojena tačkom na primer: `npm install express4@4.2.0`. Ukoliko se navede samo prvi broj verzije na primer: `npm install express@4`, biće instalirana verzija sa najvećim brojevima iza broja 4. Poslednja verzija modula se umesto zadavanjem broja verzije može instalirati i pomoću ključne reči `latest`.

Pretpostavimo da se prilikom instalacije modula *express* instalirao i modul *qs* koga modul *express* koristi. Međutim, prilikom pravljenja aplikacije javila se potreba za korišćenjem novije verzije modula *qs* koja ima funkcionalnosti koje verzija instalirana uz modul *express* nema. U tom slučaju se može instalirati potrebna verzija modula *qs* bez ikakve brige da će doći do prepisivanja stare verzije modula i da će se izazvati greška jer će *Node.js* upravljač modulima sam razrešiti konflikt različitih verzija. Verzija modula *qs* koji je naknadno instaliran će biti u `node_modules` direktorijumu aplikacije, dok će verzija koju koristi *express* biti unutar *express* `node_modules` direktorijuma. Prilikom instalacije modula mogu se postaviti i različite zastavice. Ukoliko je neki modul potreban samo prilikom razvoja aplikacije za njegovu instalaciju može se koristiti zastavica `-D`. Ovo je korisno za instaliranje raznih debagera i paketa za praćenje stanja aplikacije. Zastavica `--production` ima zadatak da prilikom instalacije zaobiđe module navedene pod svojstvom zavisnosti u razvoju fajla

package.json. Ukoliko se postavi zastavica `-g` to znači da će se modul instalirati globalno. Globalno instalirani moduli se kasnije mogu pozivati pomoću komandne linije dok se lokalno instalirani moduli (podrazumevana opcija prilikom instalacije) mogu pozivati samo u okviru aplikacije u kojoj su instalirani. Iako se moduli *express* i *mocha* često koriste, preporuka je da se oni uvek instaliraju lokalno, kako bi se uvek koristila najsvežija verzija. Gobalno instalirani moduli se instaliraju samo jednom i po potrebi se ažuriraju.

Iz svega navedenog vidi se da *Node.js* upravljač modulima nije samo veza između centralnog registra modula i aplikacije, već je i alat pomoću koga se organizuju i usklađuju verzije svih modula unutar aplikacije.

3.2.2 Pravljenje modula

Moduli čine bitan deo svake *Node.js* aplikacije jer je i sama *Node.js* aplikacija jedan veliki modul. Zbog toga je i bitno razumeti i znati kako se moduli prave. Pravljenje modula će biti prikazano na primeru aplikacije koja će prosleđenu nisku napisati u obrnutom redosledu, odnosno s leva na desno. Ovaj modul se može nazvati *okret*. Prvo će se napraviti prazan direktorijum kome će se dati ime *okret*, a zatim će se u njemu napraviti fajl *package.json*. Jedan od načina da se kreira fajl *package.json* je pomoću *Node.js* upravljača modulima komandom `npm init` ili `npm init -y`. Zastavica `-y` označava da se prilikom kreiranja fajla koriste sve predefinisane opcije. Na kraju se dobija *package.json* koji se može a i ne mora menjati.

```
{
 "name": "okret",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
 },
 "keywords": [],
 "author": "",
 "license": "ISC"
}
```

Primer 3: *package.json fajl kreiran komandom* npm init -y

Primer 4 pokazuje izgled skripta *index.js* koji se nalazi u korenu direktorijuma *okret*. U liniji `module.exports=okreniredosled` se zapravo izvozi funkcija `okreniredosled()`

kako bi kasnije mogla da se koristi u aplikaciji. To je zapravo suština modula. On izvozi funkcije koje će se kasnije uključiti u aplikaciju. Osim funkcija mogu da se izvezu niske, *JSON* objekti, promenljive. Da bi modul mogao da se instalira on mora biti u određenom formatu, tj. mora da se upakuje u `.tgz` paket. I za ovo se može iskoristiti *Node.js* upravljač modulima i njegova komanda `npm pack`. Ovako upakovan modul je spreman ili da se objavi ili da se neobjavljen instalira u *Node.js* aplikaciju. Nakon što se modul instalira, on se u aplikaciji u kojoj je instaliran uključuje, odnosno uvozi pomoću funkcije `require("okret")` bez obzira na to da li je modul prethodno objavljen ili ne, jer ukoliko modul nije javno dostupan u registru, prilikom njegove instalacije u *package.json* fajlu će se sačuvati putanja do modula.

```
function okreniredosled(recenica){
    return recenica.split("").reverse().join("");
}
module.exports=okreniredosled;
```

Primer 4: *index.js*

3.2.3 Server Node.js i događaji

Odrada zahteva na tradicionalnim veb serverima ukratko izgleda približno ovako: server primi zahtev, dodeli ga slobodnoj niti, nit ga obrađuje i na kraju vraća odgovor i oslobađa se. Ako postoji više zahteva, postoji i više niti. Server *Node.js* to radi malo drugačije. Iako je sama aplikacija višenitna, *Node.js* za obradu zahteva koristi samo jednu, glavnu nit, koja sadrži pomoćni mehanizam nazvan petlja događaja i strukturu koja se zove red događaja. To je veoma slično načinu rada V8 mehanizma. Obrada zahteva bi izgledala ovako: *Node.js* prima zahteve i stavlja ih zajedno sa povratnim funkcijama u red događaja. Glavna nit pokreće petlju događaja koja uzima jedan zahtev iz reda događaja, obrađuje ga i povratnu funkciju stavlja ponovo u red događaja i tako u krug. *Node.js* aplikacija se zatvara tek kada se obrade svi zahtevi i izvrše svi pozivi iz petlje događaja. Međutim, princip rada je drugačiji kada se izvršavaju sinhrone, tj. blokirajuće operacije. Kada petlja događaja preuzme neki zahtev za izvršavanje sinhrone operacije, ona će taj zahtev predati različitoj niti iz skladišta niti koja se nalazi i radi u pozadini. Kada se završi obrada toka zahteva, nit će iz pozadine smestiti povratnu funkciju u nit iz koje je prvobitno pokrenuta i u red događaja nad kojim petlja događaja radi u aktivnoj niti, a nit koja je radila obradu sinhronog zahteva će se osloboditi i vratiti u skladište niti. Povratna funkcija ovakvog zahteva će ili poslati neki odgovor, ili pokrenuti neki drugi događaj ili će se jednostavno završiti. Zbog ovakvog načina rada *Node.js* ima bolje performanse u

odnosu na ostale veb servere. Upravo zbog ovoga, on je skalabilan i “neblokirajući”. Međutim, ovakav model rada zahteva malo drugačiji način razmišljanja a samim tim i drugačije pisanje i organizaciju koda u odnosu na standardne veb servere. Svaki “posao” treba da se organizuje tako da ima niz povratnih funkcija koje će se pozivati, kako bi se održala skalabilnost i performanse. To znači da će se povratne funkcije nadovezivati jedna na drugu tako da rezultat jedne funkcije predstavlja objekat nad kojim se izvršava druga funkcija, a sve nadovezano izgleda kao jedna naredba tj. `deo1(argument1).deo2().deo3()` ili `deo3(deo2(deo1(argument1)))`. Ako bi se navele posebne naredbe red za redom, moglo bi se desiti da se ne zavrsi izvršenje neke naredbe iz niza naredbi a da bude pokrenuta sledeća koja zavisi od izvršenja neke od predhodnih.

3.2.4 Upravljanje ulaznim i izlaznim podacima

JSON (*eng. JavaScript Object Notation*) je jedan od najkorišćenijih načina zapisa podataka koji se koristi u *JavaSkriptu*, a samim tim je neizostavno i njegovo korišćenje u celoj *Node.js* aplikaciji. Veoma je lak za čitanje jer je njegova sintaksa veoma razumljiva ljudima. Za rad sa podacima u *JSON* formatu u *JavaSkriptu* koristi se ugrađeni objekat koji se takođe naziva `JSON`. Među najbitnijim funkcijama koje ovaj objekat obavlja su pretvaranje *JavaSkript* objekta u nisku i pretvaranje *JSON* niske nazad u objekat. Ovo se koristi kako bi se podaci lakše slali na primer iz klijentske aplikacije ka serverskoj ili uskladištili u bazi podataka.

Kako bi se *JavaScript* objekat pretvorio u *JSON* nisku koristi se funkcija `JSON .stringify(obj)` gde je prosleđeni parametar `obj` tipa *JavaScript* objekat. U suprotnom smeru, da bi se *JSON* niska pretvorila nazad u objekat koristi se funkcija `JSON.parse(tekst)` gde je prosleđeni argument `tekst` tipa string ali u *JSON* formatu.

Modul *buffer* je jedan od glavnih modula platforme *Node.js* i služi za kreiranje, čitanje i pisanje binarnih podataka, kao i za upravljanje istim. Binarni podaci su korisni na primer za slanje kompresovanih podataka što znači da se njihovim korišćenjem zauzima manje prostora u memoriji nego prilikom korišćenja teksta. Međutim prilikom konverzije teksta u bafer i bafera u tekst morao bi da se navede metod kodiranja koji se koristi (utf8, utf16, base64,...). Objekti tipa *Buffer* se skladište u strukturi koja je slična nizu, ali u delu memorije koji se nalazi van uobičajene V8 dinamičke memorije i prilikom kreiranja objekta *Buffer* mora da se odredi veličina memorije koju će objekat *Buffer* zauzeti. Nakon što se kreira objekat *Buffer*, nad njim se mogu izvršavati razne operacije pisanja u bafer i čitanja iz bafera. Za pisanje u bafer mogu se koristiti metode `write(tekst, [indeks], [duzina], [kodiranje])`, `fill(vrednost, [od], [do])`

ili `objekat[indeks]= vrednost`. Čitanje iz bafera se obavlja isto pomoću nekoliko metoda, na primer `toString([kodiranje], [od], [do])` ili `objekat[indeks]`.

Modul *stream* sadrži objekte koji rade sa tokovima podataka. Tokovi podataka su strukture memorije koje služe za čitanje ili pisanje ili u nekim situacijama za čitanje i pisanje. Svrha tokova je prenos podataka sa jedne lokacije na drugu i obično se koriste za *HTTP* podatke i datoteke. Modul *stream* sadrži nekoliko objekata koji implementiraju različite vrste tokova *Readble*, *Writable*, *Duplex* i *Transform*. Tokovi za čitanje (*eng. Readble*) se koriste prilikom čitanja podataka koji pristignu u aplikaciju sa nekog drugog izvora. Najpoznatija upotreba je prilikom čitanja *HTTP* odgovora na klijentu, *HTTP* zahteva na serveru i čitanje fajla pomoću sistema za upravljanje datotekama (*eng. file system - fs*). Konkretno, za čitanje podataka se može koristiti metoda `read([velicina_u_bajtovima])` koja vraća pročitani tok kao *String*, *Buffer* ili *null* ukoliko je tok prazan. Ako se postavi veličina u bajtovima, biće pročitan isečak te veličine, odnosno dužine. Kako su *Node.js* aplikacije vođene događajima, tok za čitanje emituje sledeće događaje:

- `readble` - emituje se da bi obavestio da se podaci mogu čitati iz toka
- `data` - emituje se u toku čitanja podataka tj. prosleđivanju isečka podatka rukovaocu događaja
- `end` - emituje se kada tok nema više podatke
- `close` - emituje se kada se zatvori osnovni izvor toka
- `error` - emituje se ukoliko je došlo do greške prilikom prijema podataka za čitanje

Tokovi za upisivanje (*eng. Writable*) omogućavaju upis podataka na način na koji ga određeni delovi aplikacije mogu pročitati. Najpoznatija primena toka za upisivanje je za slanje dodatnih podataka u *HTTP* zahtevima koje klijent šalje serveru, *HTTP* odgovorima koje server šalje klijentu i upisivanje u fajl pomoću sistema za upravljanje datotekama. Za upisivanje podataka u tok koristi se metod `write(isecak, [kodiranje], [povratna funkcija])` koja upisuje isečak u tok. Ukoliko je navedena povratna funkcija ona će se izvršiti tek kada se potroše podaci za upis. Tok za upisivanje emituje nekoliko vrsta događaja:

- `drain` - emituje se kada se potroše svi podaci za upis metodom `write()`
- `finish` - emituje ga metoda `end(isecak,[kodiranje],[povratna funkcija])` i označava kraj upisa u tok. Nakon ovog događaja podaci za upis se više ne prihvataju.

3.2.5 HTTP/HTTPs

Moduli *http* i *https* su jedni od najznačajnijih modula koji čine jezgro platforme *Node.js*. Ovi moduli imaju sve što je potrebno za brzu i osnovnu implementaciju veb servera, što je zapravo jedna od glavnih mogućnosti platforme. U praksi se za implemetaciju veb servera koristi modul *express* koji omogućava više funkcionalnosti od samih modula *http* i *https*, ali modul *express* se bazira na modulima *http* i *https*. Jedni od najbitnijih objekata koji se nalaze u okviru modula *http* i *https* su *ClientRequest*, *ServerResponse*, *IncomingMessage* i *Server*. U narednom delu će biti prikazano kako ovi objekti komuniciraju međusobno i kako se pomoću njih može implementirati veb server.

Tokom kreiranja neke funkcionalnosti u veb aplikaciji, često se dešava da je potrebna neka informacija sa nekog drugog servisa ili drugog dela aplikacije. U tom slučaju, šalje se zahtev tom servisu, po potrebi mu se pošalju i dodatni parametri, a zatim se čeka da taj servis vrati neki odgovor koji se može koristiti po potrebi. Za slanje zahteva koristi se metod `http.request(opcije, povratna_funkcija)`. Prilikom pozivanja ovog metoda, kreira se objekat *ClientRequest*. Parametar *opcije* je tipa *JavaScript* objekat i sadrži sve informacije o zahtevu koji se šalje. Neke od opcija koje se mogu postaviti su:

- `host/hostname` - adresa servera kome se šalje zahtev
- `port` - port servera
- `method` - metoda zahteva koji se šalje. Može biti GET, POST, CONNECT, OPTIONS, gde je GET podrazumevana vrednost
- `path` - putanja koja vodi do traženog servisa kojoj se mogu proslediti i parametri u formatu upita (*eng. query string*)
- `headers` - *JavaScript* objekat pomoću koga se postavlja zaglavlje zahteva

Parametar `povratna_funkcija` je funkcija koja se poziva nakon što je poslat zahtev i ona prima odgovor sa servera. Njoj se prosleđuje parametar koji predstavlja objekat *IncommingMessage* koji u ovom slučaju ima ulogu odgovora sa servera. Objekat *ClientRequest* implementira tok za upisivanje pa samim tim može koristiti sve metode i događaje toka za upisivanje. Pomoću metode `write(isecak)` može se upisati `isecak` koji je objekat tipa *String* ili *Buffer* u zahtev. U tom slučaju u zaglavlju zahteva bi trebalo da se doda opcija `'Transfer-encoding:chunked'`. Metod `end([podatak])` takođe može da upise podatak u telo zahteva tj. u tok za upisivanje,

ali se od `write()` razlikuje jer će odmah isprazniti tok i završiti zahtev. Metoda `abort()` prekida trenutni zahtev.

Kako bi se implementrirao server, koji će obraditi neki zahtev, on prvo mora da se kreira a zatim da se postave osluškivači na određenom portu, koji će da osluškuju zahteve poslate ka tom serveru. Server se kreira pomoću metode `http.createServer([osluskivac])` koja vraća objekat *Server*. Parametar `osluskivac` predstavlja funkciju koja se poziva kada se emituje događaj `request`. Događaj `request` se emituje uvek kada server primi zahtev od klijenta. Funkcija `osluskivac` prima dva argumenta objekat *IncommingMessage*, koji predstavlja zahtev koji je pristigao i objekat *ServerResponse* koji služi za slanje odgovora nazad klijentu. Postavljanje osluškivača na određenom portu, na kreiranom serveru, se radi pomoću metode `listen(port, [hostname], [backlog], [povratna_funkcija])`. Parametar `port` označava port na kome se osluškuju zahtevi, a parametar `povratna_funkcija` označava funkciju koja će se pokrenuti kada server počne da osluškuje na određenom portu. Kako bi se zaustavilo osluškivanje upotrebljava se metoda `close()` nad objektom *Server*. Tokom objašnjavanja rada klijenta i servera, mogla je da se stvori slika kako su još povezani objekti *IncomingMessage* i *ServerResponse*. Objekat *IncomingMessage* se kreira i prilikom slanja zahteva na klijentu i prilikom prihvatanja zahteva na serveru. Na strani klijenta, on predstavlja odgovor servera, a na strani servera predstavlja zahtev klijenta. Za čitanje podataka implemetira se tok za čitanje. Objekat *ServerResponse* se kreira samo na serveru, prilikom prihvatanja zahteva od strane klijenta i služi kako bi se vratio klijentu odgovor. Objekat *ServerResponse* implementira tok za upisivanje. Komunikacija između servera i klijenta će još jednom biti opisana u primerima 5 i 6. Prvo će se kreirati server u zasebnom skriptu. Server će osluškivati zahteve i vraćati neki odgovor. Kreirani server će se pokrenuti i ostaviti da radi u pozadini kako bi osluškivao zahteve. U posebnom skriptu će se kreirati klijentski zahtev ka serveru i obrađivati vraćeni odgovor od servera.

Server je napravljen tako da osluškuje zahteve na adresi `localhost` na portu `3000` i kada neki zahtev pristigne, on kao odgovor šalje nisku pomoću koje obaveštava klijenta koji je poslao zahtev da je objekat pristigao i kada je pristigao. Klijent u primeru 6 šalje zahtev i čeka odgovor. Kako je odgovor tipa tok za upisivanje, na njega se postavlja osluškivač događaja `data` jer kada se taj događaj pokrene, to znači da je u tok upisan jedan deo podataka. Kada se taj tok zatvori emituje se događaj `end` što znači da sa tog toka više ne mogu da se čitaju podaci jer je on zatvoren već se pristigli podaci mogu obrađivati po potrebi. U primeru 6 je *JSON* niska pretvorena nazad u objekat i u konzoli je ispisana prosleđena poruka.

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

http.createServer((request,response)=>{
   console.log("Zahtev je primljen na serveru");
   response.writeHead(200,{"Content-Type":"application/json"});
   var poruka=JSON.stringify({poruka:"Zahtev je primljen",vreme:new
       Date().toString()});
   response.end(poruka);
}).listen(3000,"localhost",()=>{console.log('Server osluskuje na
   portu 3000');});
```

Primer 5: *Serverski skript koji čita* GET *zahteve*

```
var http=require("http");
const opcije={
   hostname:"localhost", //ova je podrazumevana opcija
   port: 3000 // port na kome smo postavili osluskivac na serveru
};
var request = http.request(opcije,(response)=>{
   var odgovor="";
       odgovor+=isecak.toString();
   });
   response.on('end',()=>{
       const odgovorObj=JSON.parse(odgovor);
       console.log("Odgovor sa servera je '"+odgovorObj.poruka+"'")
           ;
       console.log("---");
   });
   response.on('error',(greskica)=>{console.log("greska prilikom
      primanja zahteva");});
}).end();
```

Primer 6: *Klijentski skript koji šalje zahteve metodom* GET

Slično je i sa `post` zahtevom koji je prikazan u primerima 7 i 8, samo što će se podaci koji se šalju od klijenta ka serveru poslati u telu zahteva, odnosno pomoću metode `write()`, a na serveru će se, pre nego što se pošalje odgovor, pročitati telo zahteva.

```
const http=require('http');
http.createServer((request,response)=>{
   var primljenaPoruka="";
   request.on('data',(isecak)=>{
       primljenaPoruka+=isecak;
   });
   request.on('end',()=>{
       var porukaObj=JSON.parse(primljenaPoruka);
       response.writeHead(200,{'Content-Type':'text/plain'});
       response.end(porukaObj.ime+" kaze svima \""+porukaObj.poruka
          +"\"");
   });
}).listen(3000,()=>{console.log('Server osluskuje na portu 3000')
    ;});
```

Primer 7: *Serverski skript koji čita* POST *zahteve*

```
const http=require('http');
http.request({port:3000,method:"POST"},(response)=>{
   var odgovor="";
   response.on('data',(isecak)=>{odgovor+=isecak;});
   response.on('end',()=>{console.log(odgovor);})
}).end(JSON.stringify({ime:"Mirjana",poruka:"Cao"}));
```

Primer 8: *Klijentski skript koji šalje zahteve metodom* POST

U primerima 5 i 7 serverski skript neprestano osluškuje zahteve i prestaje sa radom tek kada je nasilno prekinut, na primer pomoću tastera `CTRL+C`. Izvršavanje klijentskog skripta traje dok se ne pročita odgovor sa servera.

Modul *https* je jako sličan modulu *http* i od njega se razlikuje tako što ima dodatan bezbednosni sloj. Slanje zahteva sa klijenta i obrada istog na serveru je jako slična kao uz korišćenje modula *http* s tim što zahteva dodatna podešavanja koja se tiču autorizacije tog dodatnog bezbednosnog sloja odnosno *SSL* sertifikata. Glavne opcije koje se moraju podesiti su :

- `key` - označava privatni ključ koji se koristi za autorizaciju *SSL* sertifikata
- `cert` - javni ključ koji će se koristiti
- `agent` - ova opcija treba da se isključi ukoliko se koristi *https* modul jer je njena predefinisana vrednost `"Connection:keep-alive"` koja se šalje u zaglavlju zahteva

3.3 Modul express

U ovom poglavlju će biti prikazani načini na koji se moduli *http* i *https* mogu proširiti pomoću modula *express* koji je ujedno i najpopularniji modul koji se koristi u *Node.js* aplikacijama. *Express* omogućava ubrzanje razvoja i koristi se slično kao i moduli *http* i *https*, s tim što ima dodatne mogućnosti. Kako *express* nije glavni modul, on se mora instalirati lokalno, u projektu uz pomoć komande *Node.js* upravljača modulima. Kako bi nakon instalacije mogao da se koristi, potrebno ga je i uključiti u aplikaciju pomoću `require()` funkcije. Kako bi dalje mogle da se koriste funkcionalnosti modula *express* potrebno je instancirati i objekat *Express*. U primeru 9 je prikazano kreiranje servera koji osluškuje zahteve na portu 3000. Ovo je dovoljno za pokretanje servera, međutim, svaki poslati zahtev bi rezultirao greškom. To je zato što je potrebno da se dodaju još neke "stvari" koje bi obrađivale zahteve koje veb pregledač šalje.

```
const express=require('express');
var app=express();
app.listen(3000);
```

Primer 9: *Kreiranje servera pomoću express modula*

3.3.1 Express generator

Kreiranje servera se može ubrzati uz pomoć *express generatora* koji služi za pravljenje "kostura" koji se kasnije može nadograditi i prepraviti tako da odgovara potrebama aplikacije. Kako bi se koristio *express generator*, on se pre svega mora instalirati globalno, uz pomoć komande `npm install -g express-generator`. Ranije je rečeno da globalno instalirani moduli mogu da se koriste samostalno, pa se tako i *express generator* može koristiti nezavisno od aplikacije, iz konzole pomoću ključne reči `express`. *Node.js* aplikacija koja koristi modul *express* se uz pomoć *express generatora* može napraviti onako kako je prikazano na primeru 10.

```
express generatortest
cd generatortest && npm install
npm start
```

Primer 10: *Pozivanje komandi express generatora*

U primeru 10 je prikazano pravljenje aplikacije koja se zove *generatortest*. `express` komanda je napravila direktorijum *generatortest* i u njemu odgovarajuću strukturu

direktorijuma i fajlova. Među tom strukturom se nalazi i fajl *package.json*. Zato se i pokreće `npm install` komanda, kako bi se instalirali svi moduli koji su navedeni u *package.json* fajlu. Ako se otvori tj. pročita *package.json* fajl, može se primetiti linija `"scripts": { "start": "node ./bin/www"}` koja govori da ukoliko se u tom direktorijumu pozove funkcija `npm start`, pokrenuće se skript */bin/www.js* koji će konfigurisati server i uvesti skript *app.js* koji sadrži svu serversku logiku koju je *express generator* napravio, a koji se kasnije može menjati po potrebi. Direktorijum *public* služi za skladištenje statičkih dokumenata, poput slika, dodatnih *JavaScript* fajlova i stilova, tj. *CSS* fajlova koji se koriste u aplikaciji. Direktorijum *routes*, može se iskoristiti za implementaciju *REST API*-ja, koji će biti opisan kasnije. Direktorijum *view* se koristi za skladištenje *HTML* šablona, odnosno templejta (*eng. template*). Ovakva organizacija fajlova i direktorijuma se pokazala veoma praktičnom i korisnom, ali je svakako na programeru da odluči šta je sve potrebno aplikaciji i na koji način će ona biti organizovana.

3.3.2 Rutiranje i REST API

Na početku poglavlja o modulu *express* rečeno je da je za pokretanje servera dovoljno samo da se postavi osluškivač na nekom portu, međutim, uglavnom je potrebna preciznija obrada zahteva koji pristižu. U ovom odeljku će biti prikazani načini na koje mogu da se definišu rute (*eng. routes*), kako se rutama prosleđuju dodatni parametri i kako oni mogu da se obrade. Na kraju će sve biti "sumirano" pomoću primera mini *REST API*-ja. Za početak, svi zahtevi koji će se obrađivati u primerima će biti *GET* zahtevi. Ruta se definiše tako što se nad objektom *Express* pozivaju određene funkcije koje imaju sledeći oblik: `app.metoda(putanja, [posrednicki dodatak], povratna funkcija)` gde je `app` referenca na objekat *Express*, `metoda` je jedna od metoda *REST API*-ja koja će u budućim primerima biti *GET*, `putanja` označava *URL* ili deo *URL*-a kojim definisana ruta upravlja a `povratna funkcija` je funkcija koja prima dva parametra, `zahtev` i `odgovor` i ponaša se slično kao i povratna funkcija u objektu *Server* modula *http*. Kako bi ovo bilo jasnije za početak će se ignorisati posrednički dodatak. Primer 9 se može proširiti na način koji je prikazan u primeru 11. U praksi, često se dešava da postoji potreba da se rutama proslede dodatne informacije, na primer identifikator nekog objekta čije informacije treba da se pretraže u bazi. Postoji nekoliko načina da se proslede dodatne informacije. Najjednostavniji način prosleđivanja informacija je pomoću *HTTP* upita koji ima oblik `?kljuc=vrednost` i koji se navodi nakon *URL*-a u veb pregledaču. Međutim, kada se u serveru definiše ruta ona se definiše samo nad osnovnim *URL*-om kao što je prikazano u primeru 12. Upiti se automatski

obrađuju i smeštaju kao objekat u svojstvo `query` objekta *Request*.

```
const express=require('express');
var app=express();
app.get('/',(request,response)=>{response.send("index strana");});
app.get('/korisnici',(request,response)=>{response.send("korisnicka
    strana");});
app.listen(3000);
```

Primer 11: *Definisanje rute*

```
const express=require('express');
var app=express();
app.get('/',(request,response)=>{
    response.send("Index stranica");
});
app.get('/korisnici',(request,response)=>{
    response.send("Korisnici stranica <br> "+JSON.stringify(request
        .query));
});
app.listen(3000);
```

Primer 12: *Obrada poslatih podataka putem upita*

Informacije se mogu proslediti i definisanjem parametara. U tom slučaju u putanji rute se tačno navode koji i koliko parametra može da se očekuje što je prikazano u primeru 13.

```
app.get('/korisnici/:korisnikID',(request,response)=>{
    response.send("Korisnici stranica <br> Korisnik ID: "+request.
        params.korisnikID);
});
app.get('/detalji/:korisnikID/racun/:racunID/:vrsta',(request,
   response)=>{
    let send="Detalji stranica"+
    "<br> Korisnik ID: "+request.params.korisnikID.korisnikID+
    "<br> Racun id:"+request.params.racunID+
    "<br> Vrsta: "+request.params.vrsta;
    response.send(send);
});
```

Primer 13: *Obrada poslatih podataka definisanjem parametra*

Preuzimanje podataka pomoću parametara je jako praktično jer se na taj način može definisati koji podaci se očekuju i u kom formatu, tako da je mogućnost zloupotrebe

smanjena. Varijanta definisanja parametra bi bila i upotreba regularnih izraza prilikom navođenja putanja prilikom definisanja rute koja je prikazana u primeru 14.

```
app.get(/^\/korisnici\/(\w+)\/racun\/(\w+)\/(.+)?\$/,(request,
    response)=>{
     let send="Korisnici stranica"+
     "<br> Korisnik ID: "+request.params[0]+
     "<br> Racun id:"+request.params[1]+
     "<br> Vrsta: "+request.params[2];
     response.send(send);
});
```

Primer 14: *Obrada poslatih podataka definisanjem regularnih izraza*

Naravno, ukoliko se ruta definiše pomoću *POST* metode, dodatne informacije se uvek mogu proslediti u telu zahteva.

Za implementaciju *REST API*-ja, objekat *Express* obezbeđuje metode za svaki tip zahteva koji može da stigne od strane klijenta, i nad istim *URL*-om, tj. putanjom, mogu se definisati različite rute sa različitim metodama. *Express* podržava sledeće metode:

- `app.delete(putanja, povratna_funkcija)` - prima *DELETE* zahteve, koji se najčešće koriste kako bi naglasili potrebu za brisanje određenog objekata
- `app.get(putanja, povratna_funkcija )` - prima *GET* zahteve
- `app.put(putanja, povratna_funkcija )` - prima *PUT* zahteve koji se najčešće koriste kako bi ukazali potrebu za ažuriranjem objekta
- `app.patch(putanja, povratna_funkcija )` - uglavnom se koristi za ukazivanje potrebe za delimičnim ažuriranjem nekog objekta
- `app.post(putanja, povratna_funkcija )` - prima *POST* zahteve i uglavnom se koristi kako bi se ukazala potreba za kreiranjem novog objekta
- `app.all(putanja, povratna_funkcija )` - prima zahteve bilo kog tipa
- `app.options(putanja, povratna_funkcija)` - prima *OPTIONS* zahtev koji uglavnom šalje veb pregledač i uglavnom se koristi za identifikaciju dozvoljenih metoda

Kroz primere 15 i 16 će biti prikazana upotreba ruta kroz aplikaciju koja će prikazivati, kreirati, menjati i brisati informacije o korisnicima. Kako još uvek nije

pisano o bazama podataka i mogućnostima čuvanja podataka u bazi, za sada će se podaci čuvati lokalno, što znači da će se vratiti na staro prilikom osvežavanja stranice. Podaci o korisnicima će se proslediti pomoću posredničkih funkcija, a kod za prosleđivanje će biti sledeći: `app.use((request, response, next)=> { request .korisnici = korisnici; next();});`. Takođe će se koristiti još jedna posrednička funkcija iz dodatnog modula *body-parser* koji će obraditi telo svih zahteva, iako će se telo slati samo uz POST i PUT zahteve. Kod koji će se koristiti izgleda ovako: `app .use(bodyparser.urlencoded({ extended: true })); app.use(bodyparser.json());` i njegov zadatak je da obradi telo zahteva i smesti ga u *JSON* formatu u svojstvo body.

U direktorijumu `routes` će biti smeštena dva fajla. `index.js` i `korisnici.js`. Fajl `korisnici.js` će sadržati logiku za obradu podataka, odnosno u njoj će biti implementirane povratne funkcije koje će obrađivati svaki tip zahteva, a te funkcije će se na kraju izvesti iz fajla kako bi mogle da se koriste prilikom implementacije ruta.

```
module.exports = {
getKorisnici(request, response) {
    response.send(request.korisnici);
},
getKorisnik(request, response) {
   response.send(request.korisnici[request.params.korisnikid]);
},
postKorisnik(request, response) {
   let noviid = request.korisnici.length;
   request.korisnici.push(request.body);
   response.send("Novi korisnik ima id:"+noviid+"<br>")
},
putKorisnici(request, response) {
   request.korisnici[request.params.korisnikid] = Object.assign(
      request.korisnici[request.params.korisnikid], request.body);
    response.send(request.korisnici[request.params.korisnikid]);
},
deleteKorisnik(request, response) {
   request.korisnici.splice(request.params.korisnikid, 1);
   response.send("Obrisan");
}
};
```

Primer 15: *korisnici.js*

Fajl `index.js` u ovom slučaju nije potreban pošto postoji samo jedan skript koji obrađuje zahteve nad istim objektom. On služi da objedini sve skriptove, kao što su `korisnici.js` i opet ih izveze, radi lakšeg rukovanja i trenutno on izgleda ovako:

`module.exports ={ korisnici: require('./korisnici.js')}`. U korenu projekta treba da postoji drugi fajl `index.js` u kome će biti implementirane rute koje će koristiti logiku iz skripta `korisnici.js`.

```
const express = require('express');
const routes = require("./routes/index.js");
let korisnici = [
    {ime: 'Petar',prezime:'Petrovic', godine:26},
    {ime: 'Jana',prezime:'Janjic', godine:27},
];
let app = express();
app.use((request, response, next) => { request.korisnici =
    korisnici; next();});
app.use(bodyparser.urlencoded({ extended: true }));
app.use(bodyparser.json());
app.get('/korisnici', routes.korisnici.getKorisnici);
app.get('/korisnici/:korisnikid', routes.korisnici.getKorisnik);
app.post('/korisnici', routes.korisnici.postKorisnik);
app.put('/korisnici/:korisnikid', routes.korisnici.putKorisnici);
app.delete('/korisnici/:korisnikid', routes.korisnici.
    deleteKorisnik);
app.listen(3000);
```

Primer 16: *Glavna skripta index.js*

Za svaku metodu zahteva je pozvana samo jedna funkcija iz skripta `korisnici.js` koja obrađuje zahteve. Na ovaj način je veoma lako implementiran *REST* i samim tim je ostavljena mogućnost za dalje nadogradnje aplikacije, tako da se ne menja postojeća logika i postavljena struktura. Takođe, server napravljen na ovaj način ostavlja mogućnost da se različite klijentske aplikacije, koje mogu biti u različitim tehlonogijama, povezuju na isti server putem *HTTP(S)* zahteva, i za iste zahteve i prosleđene parametre dobiju iste odgovore, što je veoma povoljno ako se prave aplikacije za različite platforme, od veb i desktop aplikacija do mobilnih aplikacija koje danas preovladavaju. Ovakav pristup je veoma zahvalan jer na taj način sve te klijentske aplikacije dele jednu serversku koja je fizički a i logički odvojena od klijentske aplikacije čime se umanjuje programerski posao. Pored toga što *REST API* treba da bude lepo organizovan i kvalitetno napisan, on treba da bude i dobro testiran a najbolji način da se testira ovakav *API* je pomoću programa koji se zove *POSTMAN*.

3.3.3 Express zahtev i odgovor

Objekti *Request* i *Response* modula *express* se malo razlikuju od standardnih objekata *ClientRequest* i *ServerResponse* modula *http*. Oni su proširena verzija ovih objekata pa imaju i dodatne mogućnosti, odnosno dodatne metode i svojstva. Objekat *Response* služi za kreiranje i slanje *HTTP* odgovora. Uz njegovu pomoć, postavljaju se statusi i zaglavlja i vraća se nekakav odgovor nazad klijentu. Zaglavlja se postavljaju pomoću metode `set(opcija, vrednost)`, a učitavaju pomoću metode `get(opcija)`. Statusi se postavljaju pomoću metode `status(broj)` gde je broj trocifreni broj definisan u `HTTP` specifikaciji na sledeći način:

- `2XX` - označavaju uspeh
- `3XX` - označavaju redirekciju ili informacije o keširanju
- `4XX` - označavaju greške na strani klijenta, gde je najpoznatija je upotreba kodova 401,403 i 404
- `5XX` - označava greške koje su se desile na strani servera

Postoji nekolko načina da se pošalje odgovor i svi ti načini se razliku po tome kakav se odgovor šalje. Metoda `send(telo)` šalje telo kao odgovor i u zavisnosti od toga kojeg je tipa telo, postavlja vrednost opcije `Content-Type` u zaglavlju. Telo može biti tipa *Buffer*, može biti *String*, objekat ili niz. Za slanje *JSON* odgovora, koristi se metoda `json([telo])`.

3.3.4 Posrednički dodaci

Posredničke funkcije su funkcije koje se izvršavaju od trenutka kada je primljen zahtev neke rute do njegove obrade. Posredničke funkcije moraju da prime tri argumenta: objekat *Request*, objekat *Response* i funkciju `next()` koja poziva narednu funkciju u nizu. Za posredničke dodatke se može reći da su to skupovi posredničkih funkcija koje samostalno mogu da obavljaju neku radnju, na primer da učitaju telo zahteva i da ga u odgovarajućem formatu smeste na odgovarajuće mesto. Veliki broj posredničkih dodataka se nalazio u sklopu modula *express*, kao njegova komponenta, međutim, u poslednjim verzijama, ove komponente su izbačene iz modula i premeštene u registar modula, tako da se mogu po potrebi instalirati i uvesti u aplikaciju. Najčešća je upotreba dodataka koji obrađuju parametre, kolačiće i sesije. Neki od najkorišćenijih posredničkih dodataka su *body-parser* koji "razbija" podatke poslate pomoću tela *POST* zahteva i skladišti ih u svojstvo `request.body`, *passport* koji pomaže prilikom autentifikacije, *morgan* koji

služi za logovanje zahteva i koji se uglavnom koristi u procesu razvoja aplikacije. Osnovni način, kako se neki posrednički dodatak ili funkcija postavlja globalno, nad aplikacijom je pomoću funkcije `use([putanja,] posrednicka_funkcija)`, `putanja` označava deo *URL* nad kojim se primenjuje definisana `posrednicka_funkcija`. Ukoliko se definiše više različitih posredničkih funkcija nad istom putanjom, one će se izvršavati u redosledu u kojem su definisane. Takođe, posrednička funkcija se može definisati i nad samo jednom rutom, u tom slučaju, ona se navodi između putanje, i povratne funkcije rute.

```
const app=require('express')();
const bodyparser = require('body-parser');
const custom=(request,response,next) => {request.log='Ovo je
   korisnicki definisana fuckija'; next();};
const povratna=(request,response)=>{
   let poruka="";
   if(JSON.stringify(request.body)!="{}") poruka+=JSON.stringify(
      request.body)+"<br>";
   if(JSON.stringify(request.query)!="{}") poruka+=JSON.stringify(
      request.query)+"<br>";
   if(request.log) poruka+=request.log+"<br>";
   response.send(poruka);
}
app.use(bodyparser.urlencoded({ extended: true }));
app.use(bodyparser.json());
app.get('/',povratna);
app.get('/korisnici',custom,povratna);
app.listen(3000);
```

Primer 17: *Posredničke funkcije*

Posredničke funkcije mogu se definisati i nad parametrima koji se šalju putem zahteva pomoću metode `app.param(ime_parametra,posrednicka_funkcija)`. U ovom slučaju, posrednička funkcija prima četvrti parametar koji sadrži vrednost parametra.

Nakon svega ovoga, može se reći da se pisanje glavnog serverskog skripta sastoji iz nekoliko koraka:

1. uvoženja potrebih modula
2. instanciranja aplikacije
3. konfiguracije servera

4. konfiguracije posredničkih dodataka
5. definisanje ruta
6. obrade grešaka
7. pokretanje servera

Kada se sve ovako “sabere” stvarno deluje jednostavno za korišćenje a ostavlja mnogo mogućnosti.

3.4 MongoDB

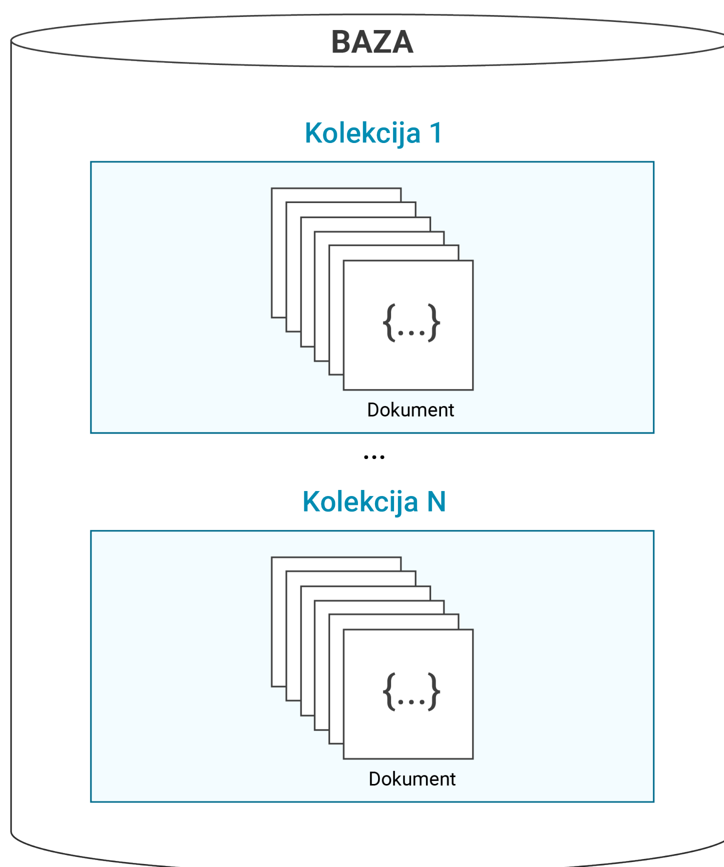
MongoDB je *NoSQL* baza podataka. *NoSQL* je zapravo akronim nastao od izraza *Not Only SQL* i označava skup baza podataka koje ne prate principe relacionih baza podataka. Među *NoSQL* bazama postoji podela prema tipu, odnosu načinu na koji se čuvaju podaci u bazi, gde *MongoDB* spada u baze koje se zasnivaju na modelu dokumenta. To bi značilo da se objekti podataka posmatraju i skladište kao dokumenti. Dokumenti imaju strukturu sličnu *JSON* formatu, a skup dokumenata, koji imaju istu namenu, naziva se kolekcija. Kolekcija se može posmatrati kao tabela u relacionom modelu baze podataka, samo što dokumenti u okviru kolekcije, mogu ali i ne moraju da imaju iste atribute. Svakom dokumentu se dodeljuje jedinstveni identifikator koji se smesta u polje `_id`. Izgled jednog dokumenta je prikazan u primeru 18.

```
 {
   _id: ObjectId("5af9b6bd6f5ca92429f6a4c5"),
   string_polje:"Neki string",
   broj_polje:1234,
   objekat_polje:{svojstvo1:"vrednost1", "svojstvo2":2, podobjekat
       :{novi_objekat:true}},
   niz_polje_sa_brojevima:[1,2,3,4],
   niz_polje_sa_stringovima:["string1","string2"],
   niz_objekata:[{jedan:"jedan","dva":2},{tri:{cetiri:[1,2,3,4],
       pet:false}}],
   niz_mesovit:["neki tekts", 1234, [1,2,"tri",4], {objekat:"jeste
       ", gde:"u nizu"}],
   null_polje:null
}
```

Primer 18: *MongoDB dokument*

Polja ne smeju da se nazivaju *null*, da sadrže tačku (zbog adresiranja u ugnježdenim objektima) a ni znak $ (zbog njegove upotrebe u komandama i operacijama). Kao što je prikazano u primeru 18 vrednosti polja u dokumentu mogu biti različitih tipova. Neki od njih su *String*, *Double*, *Date*, *Object*, *Array*, *Boolean*, *Null*. Za *MongoDB* je specifično da ne postoji spajanje tabela, u ovom slučaju kolekcija, kao u relacionim bazama podataka, ali se zato kao vrednost nekog polja može smestiti drugi objekat ili niz objekata. Međutim, ovaj način ugnježdavanja objekata je pogodan samo za relacije jedan na jedan među tim objektima. Prvi razlog je što dokumenti imaju ograničenje u pogledu memorijskog prostora i mogu biti maksimalne veličine 16MB. Drugi razlog je ukoliko postoji

mnogo dokumenata sa ugnježdenim objektima i pritom ti ugnježdeni objekti predstavljaju iste entitete, jedna promena na tom objektu će prouzrokovati iste promene u mnogo dokumenata. Način da se ovo izbegne je da se umesto celog objekta čuva referenca na taj objekat u vidu jedinstvenog identifikatora međutim, kako je malopre rečeno da ne postoji spajanje tabela, u ovom slučaju će često biti potrebno izvršiti više operacija pretrage. U principu, prilikom dizajniranja baze treba "prevagati" da li je bolje ugnježditi objekte ili je možda optimalnije izvršiti više operacija pretrage tako da podaci uvek budu dostupni i dostavljeni u najmanjem mogućem vremenu.



Slika 5: *Grafički prikaz MongoDB baze podataka*

MongoDB ima mogućnost kreiranja i takozvanih ograničenih kolekcija (*eng. capped*). To su kolekcije koje imaju fiksnu veličinu, i kada se kolekcija popuni, da bi se upisao novi dokument u kolekciju, pre toga će se najstariji dokument automatski ukloniti iz kolekcije.

3.4.1 Administriranje baze podataka MongoDB

Kada se priča o administraciji baze podataka, jedna od prvih stvari koja je potrebna da se uradi kako bi se baza pre svega zaštitila jeste da se definišu korisnici koji će moći da se povežu sa bazom podataka kao i da se odredi njihov nivo pristupa. Informacije o korisnicima jedne baze se skladište u kolekciji `system.users` i svaki *Users* dokument ima polja:

- `pwd`- šifra kojom se korisnik loguje na bazu
- `user` - korisničko ime
- `roles`- niz koji skladišti objekte koji opisuju privilegije koje korisnik poseduje nad bazom podataka
- `customData` - opciono se koristi za sladištenje dodatnih informacija o korisniku u formatu dokumenta
- `authenticationRestrictions` - opciono se navodi lista *IP* adresa sa kojih korisnik može da se poveže na server

Lista definisanih korisnika nad bazom podataka se može dobiti komandom `show users` ali i metodom `db.system.users.find()` koja pretražuje kolekciju i vraća sve dokumente u njoj, tako da je vraćena vrednost zapravo objekt kursora, koji će biti detaljnije objašnjen kasnije. Kako bi se kreirao korisnik koristi se metoda `createUser( korisnik_dokument [,nivo_upisivanja])` gde je `korisnik_dokument` dokument tipa *Users* a `nivo_upisivanja` je takođe dokument koji opisuje koji nivo upisivanja se primenjuje. U primeru 19 prikazano je kreiranje korisnika i šta se dobija kao povratna vrednost.

Kada se kreira korisnik koji ima privilegije nad nekom drugom bazom te privilegije će se čuvati samo na nivou *admin* baze, odnosno neće postojati u `system.users` kolekciji te druge baze, ali *MongoDB* omogućava da se prilikom prijavljivanja na njegov server odabere baza koja će se koristiti za autentifikaciju. Neke od privilegija mogu biti ugrađene, a neke privilegije može i korisnik sam definisati. Ugrađene privilegije su :

- `read` - korisnik može samo da isvršava operacije čitanja iz baze podataka, tj. pretrage nad nesistemskim kolekcijama. Od sistemskih kolekcija, može da čita jedino `system.indexes`, `system.js` i `system.namespaces`.
- `readWrite` - korisnik ima sve read privilegije uz mogućnost menjanja podataka u nesistemskim kolekcijama. Ovde mogu da se izvršavaju akcije za pretragu, ažuriranje, brisanje, kao i pravljenje i brisanje kolekcija i indeksa nad poljima u kolekciji

- `dbAdmin` - korisniku se dodeljuju administratorske privilegije nad bazom podataka što podrazumeva kreiranje novih kolekcija i indeksa, brisanje istih, kao i generalno kreiranje i brisanje baze podataka. Pored ovoga postoji mogućnost pozivanja metoda za opporavak baze podataka, kao i metoda koje daju statistiku o bazi.

- `userAdmin` - korisniku se dodeljuje mogućnost kreiranja, menjanja i brisanja korisnika i privilegija

- `dbOwner`- kombinacija `readWrite`, `dbAdmin` i `userAdmin` privilegija

- `clusterManager` - korisnik ima pristup akcijama koje se između ostalog koriste za particionisanje i replikaciju

- `clusterMonitor` - korisnik ima samo mogućnost nadgledanja podataka o particionisanju i replikaciji, tj. može da izvršava operacije čitanja iz baza `config` i `locals`

- `hostMonitoring` - korisniku se daje mogućnost nadgledanja i upravljanja serverima

- `clusterAdmin` - kombinacija `clusterManager`, `clusterMonitor`

- `backup` - korisniku se daju minimalne privilegije nad operacijama koje se koriste za pravljenje rezervne kopije podataka

- `restore` - korisniku se daju privilegije za obnavljanje podataka na osnovu napravljene kopije

Pored ovih privilegija, korisnicima u `admin` bazi podataka su omogućene i privilegije `readAnyDatabase`, `readWriteAnyDatabase`, `dbAdminAnyDatabase`, `userAdminAnyDatabase`. Ove privilegije se od privilegija koje se ne završavaju sa `AnyDatabase` razlikuju tako što se korisniku pomoću njih dodaju privilegije koje važe na sve kolekcije u svim bazama podataka na serveru. Ovo je pogodno, na primer, za kreiranje samo jednog korisnika koji će imati mogućnost upravljanja nad ostalim korisnicima na serveru pomoću privilegije `userAdminAnyDatabase`, ili jednog korisnika koji će imati privilegiju `dbAdminAnyDatabase`. Korisnici se brišu pomoću metode `dropUser("ime_korisnika")`. Kako bi se korisnici prijavljivali prilikom povezivanja na bazu podataka prilikom pokretanja servera treba da se koristi opcija `--auth`, dok se iz mongo konzole prijavljivanje radi pomoću metode `db.auth(user,pwd)`, kada se trenutni upravljač prebaci na željenu bazu.

```
db.createUser({user:"testuser", pwd:"testpwd",roles: ["dbAdmin", {
    role:"readWrite", db:"othertest"}], customData: {poruka: "
    Pravljenje korisnika koji ima dbAdmin privilegiju nad trenutnom
    bazom test a readWrite privilegiju nad bazom othertest"}});
Successfully added user: {
  "user" : "testuser",
  "roles" : [
    "dbAdmin",
    {
      "role" : "readWrite",
      "db" : "othertest"
    }
  ],
  "customData" : {
    "poruka" : "Pravljenje korisnika koji ima dbAdmin privilegiju
        nad trenutnom bazom test a readWrite privilegiju nad bazom
        othertest"
  }
}
```

Primer 19: *Povratna vrednost funkcije createUser()*

Baza na serveru se kreira pomoću komande `use ime_baze` ili komande `db.getSiblingDB(ime_baze)`. Ovim komandama se ujedno menja i trenutni upravljač nad željenom bazom, međutim, ukoliko baza još nema ni jednu kolekciju, ona se neće prikazati na listi baza kada se pozove komanda `show dbs`. Baza se briše pomoću metode `db.dropDatabase()` koja se poziva kada se trenutni upravljač nalazi na bazi koja se briše. Kada se baza obriše, trenutni upravljač baze podataka će i dalje ostati na upravo obrisanoj bazi podataka. Za kopiranje baze podataka koristi se metoda `db.copyDatabase(izbaze, ubazu [, izhosta, korisnickoime, sifra,mehanizam])`.

Kolekcija se kreira tako što se nad trenutnim upravljačem baze podataka pozove metoda `db.createColection(ime_kolekcije [,opcije])`. Kako je kolekcija, recimo, ekvivalent tabeli u relacionoj bazi podataka, a ako ste upoznati sa relacionim bazama podataka, znate da je prilikom kreiranja tabele neophodne da se navedu polja koja će tabela da sadrži kao i njihov tip. Prilikom kreiranja kolekcija to nije slučaj jer se u kolekcije mogu smeštati dokumenti koji ne moraju imati iste atribute. Ovo je ujedno i prednost i mana kolekcija jer sa jedne strane ne mora da se vodi računa kakva je struktura dokumenata koji se smeštaju u kolekciju a sa druge strane može da se desi da ne postoji nikakva uniformnost u okviru kolekcije. Opcije koje se zadaju su u formatu *JavaScript* objekta i mogu biti `capped` sa vrednostima *true* ili *false* koja je indikator da li je kolekcija ograničena ili ne. Ukoliko je kolekcija

ograničena, njena veličina se može podesiti pomoću opcije `size` koja prima vrednost broja bajtova ili opcije `max` kojom se određuje maksimalan broj dokumenata koji se može smestiti u kolekciju. Ako se opcija `autoIndexID` postavi na vrednost *true*, nad poljem `_id` će se postavljati indeks, ali indeksiranje se može konfigurisati i pomoću opcije `indexOptionDefaults`. Nad kolekcijom se mogu definisati i pravila provere ispravnosti podataka pomoću opcije `validator`. Objekat kolekcije može se dobiti pozivom metode `db.getCollection(ime_kolekcije)` a kolekcija se briše tako što se nad objektom kolekcije pozove metoda `drop()`.

Nad kolekcijama polje `_id` je podrazumevano indeksirano polje jer je ono jedinstveno na nivou kolekcije. Ovaj predefinisani indeks se ne može obrisati ali se zato mogu postavljati novi indeksi nad drugim poljima koji se mogu i brisati. Postoji nekoliko različitih tipova indeksa koje korisnik može da definiše: indeks nad jednim poljem, indeks nad kombinacijom polja (dva ili više polja, redosled navođenja polja je bitan), indeks nad poljem čija je vrednost niz koji se takođe indeksira po elementima, prostorni indeks koji se zasniva na koordinatama, tekstualni indeksi i heširani indeksi koji se koriste prilikom particionisanja. Takođe, indeksima se prilikom kreiranja mogu definisati određena svojstva, npr. svojstvo `sparse` može da se postavi na vrednost *true* ukoliko postoji potreba da se u indeks upisuju samo dokumenti koji imaju polje nad kojim je definisan indeks, makar to polje imalo vrednost *null*, ako se postavi svojstvo `unique` na vrednost *true*, onemogućiće se pojavljivanje više od jedne iste vrednosti polja nad kojim se definiše indeks, svojstvo `partial` će se pobrinuti da se indeksiraju samo dokumenti koji zadovoljavaju uslove postavljenje `partial` svojstvom. Indeksi se kreiraju pomoću metode `db.collection .createIndex( definicija_indeksa, svojstva_indeksa)`. Lista indeksa se može videti pozivanjem metode `db.ime_kolekcije.getIndexes()`. Određeni indeks se briše pomoću metode `db.ime_kolekcije.dropIndex("ime_indeks")` dok se svi indeksi nad kolekcijom brišu pomoću metode `db.ime_kolekcije.dropIndexes()`.

Replikacija je skup akcija kojim se definiše više servera tako da svi serveri imaju iste podatke. Skup replika sadrži minimun tri servera koji se razlikuju po ulogama koje su im dodeljene. Jedan od tih servera je glavni odnosno primarni server. On služi i za upisivanje i za čitanje podataka i skup replika može imati samo jedan primarni server. Drugi tip servera je sekundarni. Sve operacije pisanja, ažuriranja i brisanja koje se izvršavaju na primarnom serveru se posebnim operacijama distribuiraju na sekundarni server tako da je sekundarni server zapravo kopija primarnog servera. Kako bi se operacije čitanja ubrzale može postojati više sekundarnih servera sa kojih će se čitati podaci. Treći tip servera je arbitar i on služi kao posrednik između primarnog i sekundarnog. On je tu da odredi koji sekundarni server će

postati primarni ukoliko zaključi da je došlo do greške na trenutnom primarnom serveru.

Particionisanje *MongoDB* podrazumeva da se jedna kolekcija "razbije" na više skupova dokumenata od kojih je svaki skup na zasebnoj instanci *MongoDB*. Ovom metodom organizacije podataka se dobija takozvano horizontalno skaliranje podataka, koje u zavisnosti od organizacije dokumenata po klasterima može da doprinese poboljšanju performansi ili suprotno, da umanji performanse. *MongoDB* iznajmljuje veb platformu pod imenom *Atlas* u kojoj se može skladištiti baza podataka gde su replike i particije već podešene a mogu se dodatno podešavati po principu klika na dugme.

3.4.2 Rad nad dokumentima

U ovom poglavlju će biti prikazane osnovne operacije za rad nad dokumentima u kolekciji. Biće prikazano kako se dokumenti upisuju u kolekciju, kako se ažuriraju i kako se brišu. Pored tih osnovnih operacija, biće prikazani i načini za čitanje podataka iz kolekcije i filtriranja odnosno pisanje upita, od jednostavnih, do malo komplikovanijih upotrebom dodatnih operatora i agregacijom podataka. Međutim, pre svega toga potrebno je objasniti tri jako bitna pojma a to su Nivo pisanja (*eng. write concern*), nivo čitanja (*eng. read concern*) i preferirano čitanje (*eng. read preferance*).

Nivo pisanja služi da se opiše nivo na kojem instanca *MongoDB* (samostalna, u replici, u particiji) obaveštava o stanju operacija upisivanja dokumenata. To je potvrda od servera da je upisivanje izvršeno. Viši nivo znači da će prvo sačekati upisivanje sloga na disk, pa će se poslati odgovor da je upisivanje izvšeno, međutim što je nivo veći, to je izvršavanje sporije. Nivo upisivanja se definiše na sledeći način `{ w: vrednost_nivoa, j: true_false, wtimeout: br_milisek }`. Opciji `w` mogu se dodeliti nekoliko različitih vrednosti. Ako je vrednost *1* to znači da će server prvo da sačeka potvrdu o upisivanju pre slanja odgovora dok vrednost *0* znači da će poslati odgovor pre nego što sačeka potvrdu. Vrednost *majority* znači da se očekuje potvrda pisanja sa većine servera u okviru replike. Ukoliko je kao vrednost naveden skup tagova replika, to znači da se očekuje potvrda pisanja sa replika koje imaju neki od tih tagova. Ukoliko je opcija `j` postavljena na *true* operacija pisanja će se sačuvati u logu, a opcija `wtimeout` očekuje broj milisekundi koliko će se čekati odgovor i ukoliko to vreme istekne, poslaće se poruka o grešci.

Nivo čitanja se pre svega odnosi na to koji podaci će biti vraćeni nakon izvršavanja nekog upita. Na primer, ukoliko je nivo čitanja *local*, dokumenti koji će se vrati predstavljaće najsvežije podatke u trenutku izvršavanja upita, bez obiza

na to da li njihovo postojanje ili nepostojanje potvrđeno na većini replika u skupu replika. Dok, ukoliko je nivo čitanja postavljen na *majority*, svi vraćeni podaci moraju postojati na većini replika u skupu replika.

Preferirano čitanje je malo drugačije i njime se određuje sa kojih replika u skupu replika se vrše operacije čitanja. Preferirani nivo može imati nekoliko opcija

- `primary` - operacije čitanja se izvršavaju isključivo nad primarnom replikom
- `primaryPreferred` - operacije čitanja se po mogućstvu izvršavaju nad primarnom replikom a ukoliko je ona zauzeta može se izvršiti i nad sekundarnom
- `secondary` - sve operacije čitanja se izvršavaju nad sekundarnim replikama
- `secondaryPreferred` - operacije čitanja se po mogućstvu izvršavaju nad sekundarnim replikama, međutim ukoliko su sve sekundarne replike nedostupne, čitanje će se izvršiti nad primarnom replikom
- `nearest` - operacije čitanja se izvršavaju na replici koja najbrže odgovori zahtevu čitanja

Unošenje jednog i samo jednog dokumenta u kolekciju se obavlja pozivanjem metode `db.ime_kolekcije.insertOne(dokument_u_json_objekat_formatu)`. Kako bi se u kolekciji sačuvalo više dokumenata od jednom može se koristiti metoda `db.ime_kolekcije.insertMany([niz_dokumenata])`. Alternativa ovim dvema metodama je metoda `db.ime_kolekcije.insert()` koja može da primi i jedan dokument a i niz dokumenata. Alternativno, dokument se može sačuvati i pomoću metode `db.ime_kolekcije.save(dokument)` koja se može koristiti kako za ažuriranje, tako i za unošenje dokumenata u kolekciju. Prilikom unošenja dokumenata u kolekciju, ukoliko ona ne postoji, biće automatski kreirana, takođe, svaki dokument će prilikom unosa automatski dobiti polje `_id` koje će imati ulogu primarnog ključa.

Kako bi se izlistali svi dokumenti neke kolekcije koristi se metoda `db.ime_kolekcije.find()`. Metodi `find()` se može proslediti parametar koji se naziva upit. Upit koji se prosleđuje je u *JSON* formatu i ima pravila pomoću kojih se gradi. Generalno, ideja je da ključ objekta bude polje nad kojim se izvršava upit, a vrednost objekta "tip" upita, tj. uslov koji treba da se ispuni nad tim poljem. U upitu se može definisati više polja nad kojima se filtriraju podaci. Filtriranje pomoću upita jednakosti nad jednim poljem bi se definisalo kao `polje:vrednost`. Na primer, metoda `db.kolekcija.find({polje:vrednost})` u relacionoj bazi podataka bila bi jednaka upitu `SELECT * FROM kolekecija WHERE polje=vrednost`. Pri definisanju vrednosti može se koristiti i neki od operatora koji se koriste u izgradnji upita.

Tada bi upit izgledao kao `polje:{operator1:vrednost1 [,operator2:vrednost2...]}`. Ukoliko se navede više klauzula, one su spojene operatorom *AND* što bi u primeru `{polje1:vrednost1, polje2:vrednost2}` ekvivalent u relacionoj bazi podataka bio `WHERE polje1=vrednost1 AND polje2=vrednost2`. Operatori počinju znakom $, dok navođenje vrednosti zavisi od samog operatora i od tipa podataka koji se nalaze pod poljem. Neki od operatora za filtriranje su:

- $eq - operator jednakosti (upit `polje:{$eq:vrednost}` daje iste rezultate kao i upit `polje:vrednost`)
- $ne - operator razlike, vraća sva dokumenta čija se vrednost nenpoklapa sa zadatom vrednošću
- $gt - upit `polje:{$gt:uslov}` vraća dokumente čija je vrednost polja veća od vrednosti zadate parametrom *uslov*
- $gte - kombinacija $gt i $eq, tj. ekvivalent operatoru `>=`
- $lt - upit `polje:{$lt:uslov}` vraća dokumente čija je vrednost polja manja od vrednosti zadate parametnom *uslov*
- $lte - kombinacija $lt i $eq operatora, tj. ekvivalent operatoru `<=`
- $in - vraća vrednosti koje se podudaraju sa jednom vrednošću iz niza koji se navodi u operatoru. Upit `polje:{$in:[vrednost1, vrednost2]}` vraća sve dokumente čiji ključ polje ima vrednosti vrednost1 ili vrednost2
- $nin - suprotno od operatora $in, vraća sve dokumente koji imaju vrednost koja se ne podudara sa nekom vrednošću iz prosleđenog niza
- $and - spaja klauzule pomoću operatora *AND*
- $or - spaja klauzule pomoću *OR* operatora
- $not - koristi se za invertovanje operatora. Na primer, kombinacija $not i $in operatora je jednaka korišćenju operatora $nin
- $exists - može imati vrednost *true* ili *false*. Upit `polje:{$exists:true}` vraća sve dokumente koji poseduju polje čije je ime *polje*
- $all - vraća dokumente koji sadrže sve elemente navedene u nizu
- $elemMatch - kao parametar prima dokument. Koristi u slučaju da u dokumentu postoji niz koji sadrži druge dokumente i kada treba da se filtriraju rezultati po svim poljima tog dokumenta unutar niza.

U primeru 20 prikazana je kolekcija koja se zove *filmovi*.

```
{naslov:"The Shawshank Redemption", "godina":1994, "reditelj": "
   Frank Darabont", ocena: 9.3, glumci:[ "Tim Robbins", "Morgan
   Freeman", "Bob Gunton"], zanr:["krimi","drama"]},
{naslov:"The Godfather", "godina":1972, "reditelj": "Francis Ford
   Coppola", ocena: 9, glumci:[ "Marlon Brando", "Al Pacino", "
   James Caan"], zanr:["krimi","drama"]},
{naslov:"American History X", "godina":1998, "reditelj": "Tony Kaye
   ", ocena: 8.5, glumci:[ "Edward Norton", "Edward Furlong", "
   Beverly D'Angelo"], zanr:["krimi","drama"]},
{naslov:"Coco", "godina":2017, "reditelj":[ "Lee Unkrich", "Adrian
   Molina"], ocena: 8.5, glumci:[ "Anthony Gonzalez", "Gael Garcia
   Bernal", "Benjamin Bratt"], zanr:["animacija", "avantura", "
   komedija"]},
{naslov:"Relatos salvajes", "godina":2014, "reditelj": "Damian
   Szifron", ocena: 8.1, glumci:[ "Dario Grandinetti", "Maria
   Marull", "Monica Villa"], zanr:["komedija","drama","triler"]},
{naslov:"Harry Potter and the Order of the Phoenix", "godina":2007,
    "reditelj": "David Yates", ocena: 7.5, glumci:[ "Daniel
   Radcliffe", "Emma Watson", "Rupert Grint"], zanr:["avantura","
   fantazija"]},
{naslov:"Who Framed Roger Rabbit", "godina":1987, "reditelj": "
   Robert Zemeckis", ocena: 7.7, zanr:["animacija", "avantura", "
   komedija"]}.
```

Primer 20: *Kolekcija "filmovi"*

Kako bi se dobili svi dokumenti u kolekciji koristi se metoda `db.filmovi.find()`. Za dobijanje jednog dokumenta koristi se metoda `db.filmovi.findOne()`, kao rezultat će se prikazati prvi dokument koji je upisan u kolekciju. Za prikazivanje svih filmova koji su snimljeni posle 2000. godine pravi se upit na sledeći način: `db.filmovi.find({godina:{$gt:2000}})` a za uključivanje u rezultat i filmova koji su snimljeni i 2000. godine koristio bi se operator `$gte`, tj. metoda `db.filmovi.find({godina:{$gte:2000}})`. Filmovi čija je ocena između *8* i *9* dobijaju se pomoću metode `db.filmovi.find({ocena:{$gt:8,$lt:9}})`. Za prikaz dokumenata koji ne sadrži polje *glumci* koristi se metoda `db.filmovi.find({glumci:{$exists:false}})` a za prikaz svih dokumenata u kolekciji koji imaju žanr *komedija* koristi se metoda `db.filmovi.find({zanr:{$in:["komedija"]}})`.

Za ažuriranje dokumenata se takođe koriste specijalni operatori, ali zadatak specijalnih operatora za ažuriranje je da bliže objasne kakvo ažuriranje treba da se izvrši. Na primer, ukoliko vrednost polja godina treba da se uveća za jedan, operator koji se

koristi u upitu je `$inc` i to bi izgledalo ovako `$inc: {godina:1}`, dok bi umanjenje polja godina za jedan izgledalo ovako: `$inc: {godina:-1}`. Svaki operator za ažuriranje se koristi na sledeći način `$operator:{polje1:vrednost1 [,polje2:vrednost2]}`, a pored operatora `$inc` koji je malopre naveden postoje i sledeći:

- $set - služi za zamenu trenutne vrednosti polja novom zadatom vrednošću
- $rename - menja naziv polja
- $currentdate - postavlja vrednost polja na trenutni datum
- $mul - množi trenutnu vrednost polja zadatim koeficijentom
- $min - ažurira vrednost polja na zadatu vrednost samo ako mu je zadata vrednost manja od trenutne
- $max - ažurira trenutnu vrednost polja na zadatu vrednost samo kako je zadata vrednost veća od trenutne
- $unset- uklanja polje iz dokumenta
- $setOnInsert - ukoliko akcija ažuriranja za posledicu ima unošenje novog dokumenta u kolekciju, ovim operatorom se definišu predefinisane vrednosti svih polja dokumenta.
- $addToSet - dodaje vrednost u niz samo ako on ne postoji u trenutnom nizu
- $push - dodaje element u niz
- $pull - uklanja sve elemente in niza koji zadovoljavaju zadati uslov
- $pop - ukoliko mu se prosledi vrednost 1 onda uklanja poslednji element iz niza, a ukoliko mu se prosledi vrednost -1 onda uklanja prvi element iz niza
- $pullAll - uklanja sve vrednosti koje se poklapaju sa zadatim vrednostima iz niza

Za ažuriranje dokumenata nije dovoljna sama upotreba operatora za ažuriranje već ti operatori postoje i kao pomoćno sredstvo. Ažuriranje može da se izvrši pomoću nekoliko funkcija, `save()`, `update()`, `updateOne()`, `updateMany()`, `findOneAndUpdate()`. `update()` funkcija ima sledeću formu: `db.ime_kolekcije.update(upit, operatori, opcije_objekat)`. Ukoliko je opcija `upsert` postavljena na vrednost *true* i ukoliko ni jedan dokument ne odgovara traženom upitu, novi dokument će biti dodat u kolekciju . Ovde se može iskoristiti operator `$setOnInsert` da bi se zadale vrednosti polja

prilikom unosa. Ukoliko je opcija `multi` uključena sa vrednošću *true*, ažuriraće se svi dokumenti koji odgovaraju zadatom upitu, a ukoliko je opcija `multi` postavljena na *false*, ažuriraće se samo jedan dokument i to prvi. Opcija `writeConcern` definiše nivo upisivanja. Ukoliko treba da se na primer ažurira film sa naslovom "Who Framed Roger Rabbit" i to tako da se doda polje *glumci* kao niz koji sadrži niske i da se promeni polje *godina* na vrednost *1988*, metoda za ovakvu izmenu bi izgledala: `db.filmovi.update( {naslov:"Who Framed Roger Rabbit"}, {$set: {godina:1988}, $push: {glumci: {$each: ["Bob Hoskins", "Christopher Lloyd"]}}},{multi:false})`. Brisanje dokumenata se radi pomoću funkcije `db.ime_kolekcije.remove([upit])`. Ukoliko se izostavi upit, obrisaće se svi dokumenti iz kolekcije a ukoliko se navede, obrisaće se samo oni dokumenti koji zadovoljavaju dati upit.

Agregacija u *MongoDB*-u se može izvesti na tri načina. Jedan od tih načina je takozvana prosta agregacija i sastoji se od samo dve metode `count()` i `distinct()` koje se mogu izvršiti nad upitima. Prva vraća broj dokumenata koji zadovoljavaju zadati upit nad jednom kolekcijom a druga u zavisnosti od naziva polja prosleđenog u argumentu vraća jedinstvene vrednosti. Za ostale agregacije, na primer mininum ili maksimum nekog polja potrebno je koristiti ili agregacioni radni okvir ili `map-reduce` funkcije. Agregacioni okvir radi tako što se nad određenom kolekcijom pozove metoda `aggregate()` kojoj se prosleđuju dodatni parametri odnosno operatori. Metoda `aggregate()` radi tako što u više koraka izdvaja određene dokumente a zatim formatira izlaz u vidu dokumenata tako da prikažu željene informacije. Na primer, da bi se nad kolekcijom *filmovi* prikazala prosečna ocena svih dokumenata koristila bi se metoda `db.filmovi.aggregate([{$group: { _id:null , prosek: {$avg:"$ocena"}}} ])`. U ovom slučaju agregacioni okvir prvo nad svim dokumentima u kolekciji prolazi kroz korak grupacije po navedenom identifikatoru, odnosno izdvaja sve moguće grupe polja navedenih u `_id` opciji koja je obavezna. Kako je u ovom primeru navedeno da `_id` ima vrednost `null`, svi dokumenti će se grupisati u jedan dokument. Dalje, dokument će sadržati polje koje je u primeru nazvano *prosek* u kojem će se izračunati prosečna ocena svih dokumenata. Izlaz iz komande će izgledati ovako: `{ "_id": null, "prosek": 8.216666666666667}` Za prikaz prosečne ocene svih filmova koji su snimljeni nakon 2000. godine agregaciona metoda bi izgledala ovako: `db.filmovi.aggregate([{$match:{godina:{$gt:2000}}},{ $group: { _id:null, prosek: { $avg: "$ocena"} } } ])`. U ovom slučaju bi bi se prvo izveo korak naveden opcijom `$match` odnosno, izdvojili bi se prvo svi dokumenti koji odgovaraju navedenim kriterijumima, zatim bi se nad njima izveo korak grupacije kao u prethodnom primeru. Ukoliko bi se dokumenti grupisali po nekom polju, odnosno ukoliko bi rezultat vratio više dokumenata nego u prethodnom primeru, može

se primeniti i opcija, odnosno korak `$sort`, koja sortira dokumente nakon grupacije ili `$limit` koja prikazuje određeni broj dokumenta u rezultatu. Celu agregaciju treba zamisliti kao niz koraka gde svaki sledeći naveden korak kao ulaz prima izlaz iz prethodnog koraka, tako da postoji razlika u redosledu prilikom navođenja operatora. Ukoliko bi se, na primer, korak `$sort` naveo ispred koraka `$group`, prvo bi se dokumenti sortirali po određenom kriterijumu a zatim bi se grupisali.

3.4.3 MongoDB u Node.js okruženju

Kako bi bazom podataka *MongoDB* moglo da se upravlja iz *Node.js* aplikacije neophodno se da u aplikaciji instalira drajver pomoću koga se *Node.js* aplikacija povezuje na bzu podataka *MongoDB*. Jedan od drajvera koji postoji je *Mongoose*. *Mongoose* je *ODM* (*eng. object-document-mapper*) i služi da premosti razlike između baze podataka i jezika koji se koristi za programiranje, tj. da omogući korišćenje baze unutar aplikacije. *Mongoose* je pogodan jer ima dodatne funkcionalnosti koje pomažu pri radu sa bazom podataka. Te funkcionalnosti fizički ne postoje na erveru *MongoDB*, ali su programski implementirane kako bi se olakšala upotreba, na primer definisanje šema ili pravljenje upita nad dve kolekcije istovremeno, slično spajanju tabela u relacionom modelu. Da bi se to ostvarilo, *Mongoose* se služi klasama *Schema, Model, Document, Query* i *Aggregate.* Klasa *Schema* služi za definisanje šeme, odnosno strukture neke kolekcije. Klasa *Model* služi za definisanje logike nad dokumentima i predstavlja sve dokumente u kolekciji, a definiše se na osnovu zadate šeme. Iako je rečeno da je za *MongoDB* specifično to što kolekcije ne zahtevaju da se prethodno definiše struktura, ovo je jako korisno za proveru ispravnosti objekata, odnosno da se u dokument neke kolekcije ne upišu polja koja nisu potrebna, kao i za formatiranje objekata prilikom upisa. Klasa *Document* predstavlja pojedinačan dokument dok klase *Query* i *Aggregate* služe kao "čuvari" upita koji bi se kasnije izvršili nad objektom klase *Model* što olakšava ponovno korišćenje i prilagođavanje upita po potrebi. Kako bi se aplikacija povezala sa bazom podataka koristi se metoda *connect(konekcionaniska, opcije [,povratnafunkcija])* nad osnovnim objektom *Mongoose*. Niska koja se koristi za povezivanje sa bazom podataka je ovakvog formata: `"mongodb://[username:password@]host1[:port1][,host2 [:port2],...[,hostN[:portN]]][/[database][?options]]"`. U slučaju da se aplikacija povezuje na lokalnu bazu koja se zove *test*, bez korisničkog imena i šifre, niska za povezivanje bi bila `"mongodb://localhost/test"`. Dodatne opcije prilikom povezivanja sa bazom su u *JSON* formatu gde je ključ indikator opcije a vrednost predstavlja vrednost te opcije. Opcije mogu biti:

- `readPreferance` - ovom opcijom se opisuje preferirano čitanje nad skupom re-

plika

- `ssl` - opcija treba da se postavi na vrednost *true* ukoliko se za povezivanje za bazom koristi *SSL* protokol
- `auto_reconnect` - ukoliko j vrednost postavljena na *true*, klijent, u ovom slučaju aplikacija će pokušati ponovo da uspostavi konekciju sa bazom ukoliko dođe do greške
- `readConcern` - opcija kojom se odeđuje nivo čitanja
- `W` - opcija kojom se određuje nivo upisivanja.

Veza sa bazom se raskida pomoću metode *disconnect()*. Prilikom pozivanja metode `connect()` kreira se objekat *Connection*.

Kako bi *Mongoose* mogao da se koristi, kao i da se koriste sve njegove mogućnosti mora prvo da se definiše šema nad nekom kolekcijom. Definicija šeme je slična definiciji atributa prilikom kreiranja tabele u relacionim bazama podataka. To je na neki način "preslikavanje" objekta šeme sa stvarnom kolekcijom u bazi. Šema se kreira tako što se instancira objekat klase *Shema* glavnog objekta *Mongoose* a zatim se toj klasi prosleđuje objekat koji opisuje tu šemu pri čemu se mogu postaviti i dodatne opcije. Objekat definicije šeme je formata `polje:tip` gde *tip* može biti alfanumerička vrednost tipa tog polja ili drugi objekat koji pored osnovnog svojstva `type` može da ima i dodatna svojstva koja opisuju predefinisanu vrednost, ili obaveznost polja. Tip polja može biti *String, Number, Date, Buffer, Boolean, Mixed, ObjectId, Array i Map*. Tip *Mixed* označava da polje može da sadrži bilo koji tip dok tip *ObjectId* očekuje kao vrednost *ObjectID*. Primer 21 pokazuje kako može da izgleda objekat kojim se definiše šema.

Pored podrazumevane vrednosti (*eng. default*) može se proslediti svojstvo `required` koje označava da je prilikom unosa novog dokumenta neophodno da dokument poseduje polje nad kojem je uključeno svojstvo `required`, svojstvo `lowercase` automatski transformiše nisku u identičnu nisku samo sastavljen od malih slova dok svojstva `set` i `get` definišu funkcije kojima se postavljaju i prikazuju vrednosti polja nad kojima su definisane. Nad nekim poljem pomoću ključne reči `ref` može se dodeliti referenca ka drugoj šemi. Opcije koje se mogu postaviti prilikom kreiranja objekta *Schema* su `collection` koja označava ime kolekcije nad kojom se definiše šema, `capped` kojom se navodi da li je kolekcija ograničena ili ne, `_id` koja za vrednost *true* automatski dodeljuje polje `_id` dokumentima a ukoliko je opcija `strict` postavljena na *true* to znači da polja koja nisu definisana u šemi,

a prosleđena su za čuvanje neće biti sačuvana. Nad objektom definisane šeme dalje se mogu kreirati dodatne metode koje se čuvaju u svojstvu *methods* objekta šeme.

```
{
tip_string:   String,
tip_number:    Number,
tip_date_sa_predefinisanom_vrednoscu: { type: Date, default: Date.
   now },
tip_boolean:  Boolean,
     tip_mixed:   Schema.Types.Mixed,
  _tip_object_id: Schema.Types.ObjectId,
    tip_decimal: Schema.Types.Decimal128,
    tip_array: [],
niz_brojeva:[Number],
niz_mixed:[Schema.Types.Mixed],
ugnjezdeni_objekat:{
    svojstvo_string_sa_predf_vred:{type: String, default:"
       Predefinisana"},
    svojstvo_array:[]
                   }
}
```

Primer 21: *Objekat kojim se definiše šema*

U primeru 22 se može videti kako se kreira šema za kolekciju *filmovi* i kako se nad tom šemom kreira metod koji vraća nisku u formatu "naslov (godina)"

```
var mongoose=require('mongoose');
var filmoviSchema=new mongoose.Schema({
   naslov:{type:String, required:true},
   godina:Number,
   reditelj: mongoose.SchemaTypes.Mixed, //niska ili niz niski
   ocena: {type:mongoose.SchemaTypes.Decimal128, min:0.0, max
      :10.0},
   glumci:[String],
   zanr:[String]
},{collection:"filmovi"});
filmoviSchema.methods.formatiranIspis=function(){
   return this.naslov+" ("+this.godina+")";
}
```

Primer 22: *Kreiranje šeme*

Ključna reč `this` u metodi `formatiranIspis()` je pokazivač na objekat *Document* nad kojim se ova metoda može pozvati. U primeru 23 je prikazano kreiranje modela

na osnovu definisane šeme pomoću klase *Model* glavnog objekta *Mongoose* kojoj se prosleđuje naziv i šema na osnovu se definiše model.

```
var Filmovi=mongoose.model("Filmovi" ,filmoviSchema);
```

Primer 23: *Kreiranje modela*

Nad modelom se mogu definisati upiti i na taj način bi se dobili objekti klase *Document*. Model se može koristiti za unos novih dokumenata u kolekciju, ažuriranje i brisanje. Upiti se definišu na sličan način koji je opisan u poglavlju "Rad nad dokumentima", samo što bi se metodi `find()` prosledila i povratna funkcija koja kao parametre prima grešku i rezultate. Međutim, lakše i fleksibilnije je praviti upite korišćenjem objekta *Query*. Da bi se dobio objekat *Query*, dovoljno je da se nad kreiranim modelom pozove metoda `find()` bez prosleđivanja povratne funkcije. Ova metoda će se samo sačuvati u objektu *Query*. Da bi se ona izvršila, odnosno da bi se izvršio upit na modelom, neophodno je da se nad objektom pozove metoda `exec()` koja kao argument prima povratnu funkciju pomoću koje se može utvrditi uspešnost izvršavanja upita. Objekat *Query* ostavlja još mnogo mogućnosti pre pozivanja metode `exec()`. Može se na primer dodatno filtrirati pomoću metode `where()` uz korišćenje metoda koje imaju ulogu operatora na primer `gt()`, `lt()`, `ne()`, `in()`, `or()`, `and()`, `exists()`, `mod()`, `elemMatch()`. Može se uraditi projekcija pomoću metode `select()` ili sortiranje pomoću metode `sort()`. Jedna od posebno zanimljivih metoda koja se može pozvati nad objektom *Query* je metoda `populate()` koja, ako je u šemi navedena referenca ka nekoj drugoj šemi, izvršava nešto slično spajanju tabela gde se u rezultatu, na mestu tog polja nalazi dokument iz referirajuće kolekcije odnosno šeme. Nakon što se upit izvrši, rezultat upita će biti prosleđen povratnoj funkciji metode `exec()` i taj rezultat će biti objekat klase *Document*. Međutim objekat *Document* može biti više od samo rezultata. On može služiti za unos novih dokumenata u kolekciju, za ažuriranje i brisanje. Za instanciranje novog objekta klase *Document* potrebno je da se pozove konstruktor željenog modela što je prikazano u primeru 24 gde se kreiranje dokumenta izvršava na dva načina.

```
var novi_film=new Filmovi();
novi_film.naslov="Novi film";

var novi_novi_film=new Filmovi({naslov:"Novi novi film"});
```

Primer 24: *Novi dokumenti*

U primeru 24 je instanca objekta *Document* dodeljena dvema promenljivama

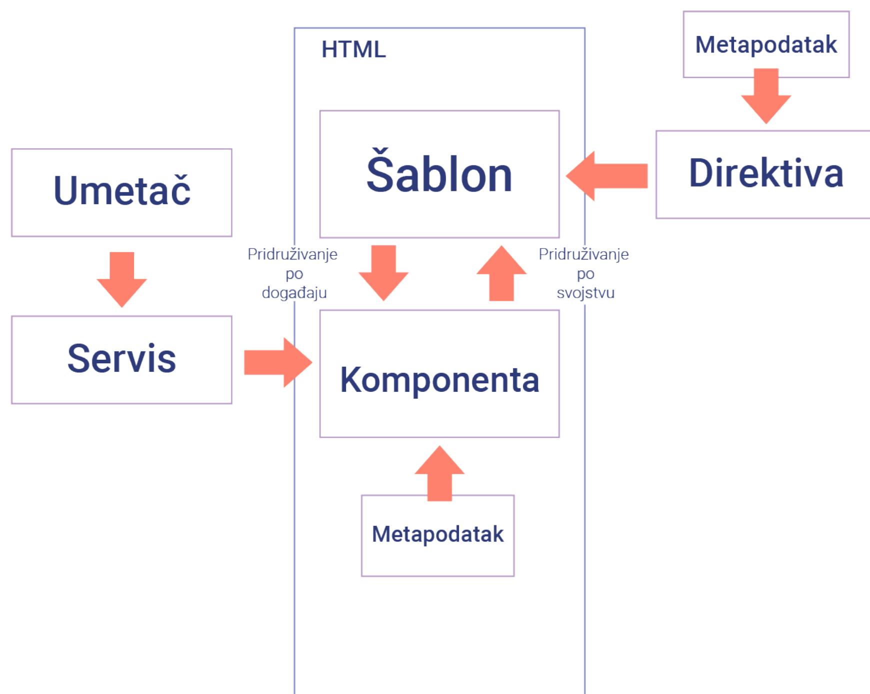
novi_dokument i *novi_novi_dokument*. Prilikom instanciranja, objektu se dodaje polje `_id` kome se automatski generiše vrednost tipa *ObjectID*. Nad objektom *Document* se dalje mogu pozivati razne metode i svojstva koje bi odredile stanje objekta, na primer svojstvo `isNew` vraća vrednost *true* ukoliko je dokument nov i vrednost *false* ukoliko taj dokument već postoji i izvučen je iz kolekcije, metoda `equal()` koja kao argument prima drugi dokument se može koristiti za poređenje dokumenata. Metoda `isModified()` vraća odgovor da li je polje prosleđeno u argumentu izmenjeno od kad je objekat nastao i ako jeste, vratiće *true* ukoliko ta izmena nije sačuvana. Kako bi se dokument sačuvao, dovoljno je da se nad njim pozove metoda `save()` koja kao argument prima povratnu funkciju pomoću koje se može utvrditi uspešnost čuvanja dokumenta u kolekciji. Više dokumenata od jednom se može sačuvati tako što će se nad objektom modela pozvati metoda `create()` i njoj treba da se proslediti niz *JavaScript* objekata. Slično se radi i ažuriranje dokumenata. Upitom se izdvaja dokument koji treba da se ažurira, ažuriraju se željena polja a zatim se tako izmenjen dokument sačuva pomoću metode `save()`. Drugi način je da se nad tim izdvojenim dokumentom pozove metoda `update()` koja kao prvi argument prima *JavaScript* objekat koji opisuje željeno ažuriranje. Metoda `update()` vraća objekat *Query*, tako da je nakon nje potrebno pozvati `exec()` metodu kako bi se ažuriranje izvršilo. Za ažuriranje više dokumenata potrebno je da se nad modelom pozove metoda `update()` i da se njoj proslede parametri za ažuriranje. Veoma bitan korak je da se nakon toga nad nad dobijenim objektom *Query* postavi svojstvo `multi` na vrednost *true* a zatim mogu dalje da se filtriraju dokumenti pomoću metoda `where()` i ostalih operatora i za kraj da se izvrši upit za ažuriranje pomoću metode `exec()`. Dokumenti iz kolekcije se uklanjaju tako što se nad željenim objektom dokumenta pozove metoda `remove()`, a uklanjanje više dokumenata iz kolekcije je slično ažuriranju više dokumenata u kolekciji samo što se umesto `update()` metode poziva metoda `remove()`.

3.5 Angular radni okvir

Angular je radni okvir koji se koristi za pravljenje klijentskih aplikacija koje se pokreću na veb pregledaču. Kada se govori o *Angularu*, potrebno je da se napravi razlika između *Angular.js* radnog okvira i *Angular.io* radnog okvira. *Angular.js* radni okvir je prva nastala verzija Angulara i on u svojoj sintaksi koristi *JavaScript* jezik dok *Angular.io* za sintaksu koristi *TypeScript* koji se posebnim kompajlerima prevodi u *JavaScript*. U ovom poglavlju, kada se bude pričalo o *Angularu*, podrazumevaće se *Angular.io*. Angular nije lak za korišćenje jer je potrebno dobro poznavati ne samo pojedinačne klase istog, već i arhitekturu, namenu i povezanost klasa, kao i na koji način se iste zajedno koriste u aplikaciji. Njegov osnovni element je modul koji služi da objedini komponente, direktive, servise, razne funkcije pa i druge module u jednu logičku celinu koja se kasnije može koristiti samostalno. Najveća prednost *Angulara* je njegova takozvana dvosmerna povezanost modela pomoću koje se izmene izvršene na interfejsu aplikacije automatski mogu pročitati i u pozadinskom delu aplikacije, bez korišćenja *AJAX* poziva i obrnuto.

3.5.1 Arhitektura Angulara

Angular aplikacija je kao i *Node.js* aplikacija organizovana kao skup mogula. Moduli služe da okupe sličan kod po funkcionalnosti i *Angular* aplikacija mora da poseduje barem jedan modul koji se naziva modul koren (*eng. root module*), oko koga se kasnije po potrebi nadograđuju ostali moduli. Za *Angular* aplikacije se može reći da koriste Model - Pogled - Kontroler (*eng. MVC*) šablon, pri čemu servisi preuzimaju ulogu modela a komponente mogu imati ulogu kontrolera. Međutim komponenta zajedno sa šablonom može imati i ulogu pogleda. Zbog malopre navedenog dvosmernog povezivanja modela i načina na koji se servisi "umeću" (*eng. inject*) u komponente pomoću mehanizma umetanja zavisnosti (*eng. dependency injection*), ni jedna od ovih stavki nije u potpunosti razdvojena da bi sa sigurnošću moglo da se kaže da *Angular* implementira Model - Pogled - Kontroler šablon.



Slika 6: *Grafički prikaz arhitekture Angulara*

3.5.2 Moduli

Angular radni okvir ima različite biblioteke koje su grupisane kao moduli pomoću kojih se pravi aplikacija. Aplikacije koje se prave korišćenjem *Angluara* su modularne u svojoj osnovi (sastoje se od puno malih delova) i grade se korišćenjem i pravljenjem različitih modula. Moduli mogu imati komponente, servise, funkcije, i/ili promenljive i vrednosti. Neki moduli mogu imati kolekcije drugih modula i tada se zovu biblioteke modula. Osnovni moduli *Angulara*, kao što su *core*, *common*, *http*, i *router* koji počinju oznakom `@angular` sadrže mnogo drugih podmodula. U aplikaciju se uvoze samo oni moduli koji su neophodni pomoću naredbe prikazane u primeru 25

```
import { BrowserModule } from '@angular/platform-browser';
```

Primer 25: *Uvoženje modula u Angular aplikaciju*

U primeru 25 se iz biblioteke modula *platform-browser* uvezao modul *BrowserModule.*

Bilo koji modul za koji je definisano da može biti izvezen može takođe biti i uvezen u neki drugi modul a iz modula se izvoze komponente, servisi, direktive, funkcije i drugi podaci. Svaka *Angular* aplikacija poseduje barem jedan modul i

taj modul se predefinisano naziva *AppModule*. Da bi neka klasa mogla da se nazove modulom ona mora da bude opisana dekoraterom `@NgModule` koji može da ima sledeća svojstva:

- `imports` - niz drugih modula čije komponente su potrebne za rad komponenti ovog modula
- `providers` - niz servisa koji se koriste u modulu
- `declarations` - niz komponenti, direktiva i filtera koji čine ovaj modul
- `exports` - niz koji čini podskup *declarations* svojstva i taj skup služi da se definiše koje deklaracije iz ovog modula se mogu uvesti u druge module.
- `bootstrap` - samo koreni odnosno *AppModule* bi trebalo da sadrži ovo svojstvo jer se pomoću njega definiše koreni (osnovni) pogled (u većini slučajeva komponenta *AppComponent*) u kojoj se nalazi svi pogledi koji se koriste u aplikaciji

Pored opisivanja klase pomoću `@NgModule` funkcije da bi modul mogao da se koristi potrebno ga je i izvesti, tj. eksportovati što je prikazano u primeru 26.

```
export class AppModule { }
```

Primer 26: *Izvoženje modula*

3.5.3 Komponente

Komponente su klase koje se koriste kako bi se implementirao pogled tj. *HTML* strana. Svojstva i metodi komponente korišćeni u šablonu omogućavaju da šablon interaktivno razmenjuje podatke sa komponentama. Koreni modul sadrži najmanje jednu korenu komponentu. Korena komponenta se u aplikacijama često naziva *AppComponent*. Ova komponenta objedinjuje sve ostale komponente koje postoje u aplikaciji, odnosno služi da se iz nje pozivaju ostale komponente. Kako se korisnik kreće kroz aplikaciju, tako se komponente kreiraju, menjaju i uništavaju. Period od kreiranja komponente do njenog uništavanja se naziva životni ciklus komponente. Životni ciklus komponente se sastoji od nekoliko bitnih događaja kao što su *OnChanges, OnInit, DoCheck, AfterContentInit, OnDestroy* i drugi. Za svaki od ovih događaja definisani su rukovaoci događaja (*eng. Life-Cycle hooks*) koji se zovu isto kao i događaji sa prefiksom *ng* pa je tako za događaj *OnInit* predefinisani rukovalac događaja funkcija `ngOnInit()` i td. Da bi klasa bila komponenta ona mora biti

opisana pomoću dekoratera*@Component* na sličan način kako se opisuje i modul. Pomoću dekoratera se u komponentu ubacuju metapodaci, a dekorater komponente može da sadrži:

- `selektor` - niska koja jedistveno određuje komponentu i omogućava da se komponenta u šablonima poziva pomoću *HTML* elementa
- `templateUrl` - adresa *HTML* fajla (šablona) koji zajedno sa komponentom čini pogled
- `template` - niska koja predstavlja šablon. Slično kao i `templateUrl` samo što se šablon ne učitava iz eksternog fajla već se definiše kao niska
- `providers` - niz provajdera odnosno "umetajućih" servisa koje komponenta koristi i koji će biti inicijalizovani pomoću mehanizma umetanja zavisnosti
- `styleUrls` - niz eksternih *CSS* fajlova čiji sadržaj ima uticaj na izgled šablona ove komponente

U primeru 27 je prikazano na koji način komponenta može biti opisana dekoraterom.

```
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
```

Primer 27: *Dekorater komponente*

Da bi se koristila `@Component()` funkcija ona mora biti prvo uvezena iz modula *core*. Pored nje iz modula *core* se mogu uvesti i interfejsi događaja životnog ciklusa komponente na primer *OnInit* koje klasa koja je opisana dekoraterom `@Component` može implementirati.

Kako u primeru 28 *PocetnaComponent* nije korena komponenta već je jedna od dece/naslednika korene komponente, svako pojavljivanje *HTML* elementa `<app-pocetna></app-poceta>` će rezultovati inicijalizacijom komponente *PocetnaComponent* i na tom mestu će se prikazati šablon `pocetna.component.html`.

```
import { Component, OnInit } from '@angular/core';

@Component({
 selector: 'app-pocetna',
 templateUrl: './pocetna.component.html',
 styleUrls: ['./pocetna.component.css']
})
export class PocetnaComponent implements OnInit {

 constructor() { }

 ngOnInit() {
 }
}
```

Primer 28: *Komponenta*

3.5.4 Šabloni

Šabloni (*eng. template*) se mogu posmatrati kao vizualna reprezentacija komponenti. Oni su odgovorni za prikaz podatka i izmenu podataka u zavisnosti od akcija korisnika. Dok se u komponentama koristi *TypeScript*, šabloni se pišu pomoću *HTML*-a, ponekad i *CSS*-a ali su obezbeđeni načini da se neko svojstvo komponente ili metoda iskoristi u šablonu i umetne u *HTML*.

3.5.5 Povezivanje podataka

Kako se *Angular* većinom koristi kao aplikacija koja se prikazuje na strani klijenta njegov zadatak je uglavnom da prikaže podatke na korisničkom interfejsu i da se omogući promena podataka u skladu sa odgovorima korisnika. To znači da se svojstva klasa i komponenti prikazuju na veb stranici, a izmena istih, treba da se odrazi na komponente iz kojih potiču podaci. *Angluar* upravlja povezivanjem podataka, koordinirajući stalno stanja istih između šablona i komponenti. Šabloni daju instrukcije *Angularu* šta i na koji način se povezuje. Postoje dva načina povezivanja podataka: jednosmerno povezivanje i dvosmerno povezivanje. Jednosmerno povezivanje pridružuje podatke iz komponente nekom elementu na *HTML* stranici ili obrnuto. Dvosmerno povezivanje obezbeđuje da se momentalno odrazi bilo koja promena na bilo kojoj strani: komponente na *HTML* element ili element na komponentu.

U primeru 29 *naslov* unutar dvostrukih vitičastih zagrada predstavlja primer jednosmernog povezivanja koji se naziva interpolacija. Interpolacijom se ne moraju

povezivati samo svojstva komponente već mogu da se koriste i izrazi i pozivi funkcija koje vraćaju neki rezultat. U primeru 29 se podaci iz komponente prosleđuju šablonu. Pored interpolacije, u jednosmerna povezvanja spadaju i poverzivanje po svojstvu i povezivanje po događaju. Povezivanje po svojstvu (*eng. property binding*) je oblika `[svojstvo]="vrednost"` i uglavnom služi kako bi se podaci iz nadkomponente prosledili podkomponenti putem *HTML* elementa komponente. Međutim, da bi podkomponenta prihvatila podatke koji joj se prosleđuju putem povezivanja preko svojstva ona mora da ima definisano svojstvo `@Input()`. Povezivanje po događaju (*eng. event binding*) je obrnuto od interpolacije jer se u ovom slučaju podaci iz šablona prosleđuju komponenti. Povezivanje po događaju je oblika `(dogadjaj)="rukovalacdogadjaja"`. *Angular* je obezbedio podršku za sve *DOM* događaje i za događaj `onClick` na elementu <*div*> povezivanje bi izgledalo ovako: `<div (click)="clickFunkcija()"> </div>`. Dvosmerno povezivanje modela se uglavnom koristi na formama. Ono omogućava da se promene koje korisnik unosi u, na primer, <*input*> element koji je povezan sa modelom automatski očitaju u kontroleru. Elementi se dvosmerno povezuju pomoću direktive `[(ngModel)]="svojstvo"` gde *svojstvo* označava ime svojstva čiju vrednost treba pročitati u šablonu i/ili izmeniti u kontroleru. Ako bi se putem dvosmernog povezivanja povezalo svojstvo objekta neke klase koja je definisana u kontroleru, referenca za povezivanje bi bila `"objekat.svojstvo"`. Na primer, ukoliko postoji objekat koji se naziva *prijava*, koji sadrži svojstvo *korisnickoime*, element <*input*> bi se pomoću modela povezao na sledeći način: `<input type="text"[(ngModel)]= "prijava.korisnickoime">`

```
<h1>{{naslov}}</h1>
```

Primer 29: *Interpolacija*

3.5.6 Direktive

Direktive su instrukcije ili vodiči za prikaz šablona. Klasa opisana metapodacima (dekoraterom) sa oznakom `@Directive` se naziva direktivom. Postoji tri tipa direktiva podržana od strane *Angulara*, a to su direktiva komponente, direktiva strukture i direktiva atributa. Direktive komponenti su zapravo komponente, samo što nemaju dekorater `@Directive()` već dekorater `@Component()` koji se od dekoratera `@Directive()` razlikuje samo po tome što ima dodato svojstvo *template* odnosno šablon. Direktive strukture utiču na izgled i strukturu *HTML* stranice. One mogu da menjaju *HTML* elemente. Postoje tri ugrađene direktive strukture: `*ngFor`, `*ngIf` i `[NgSwitch]`. Direktiva `*ngFor` radi isto što i `for` petlja u *JavaScriptu* - služi da

iterira kroz zadati niz elemenata i dodatno da za svaku iteraciju primeni određeni *HTML* šablon. Kako se niz kroz koji direktiva iterira dinamički menja, tako se menja i struktura na koju `*ngFor` utiče. Na primer ako postoji svojstvo komponente koje se zove *korpa* i sadrži niz niski koje označavaju artikle, direktiva `*ngFor` može da se iskoristi kako bi se napravila struktura *HTML* liste koja će prikazivati saržaj korpe što je prikazano u primeru 30.

```
<ul>
 <li *ngFor="let artikal of korpa">{{artikal}}</li>
</ul>
```

Primer 30: *Direktiva *ngFor*

Ovom iteracijom dobijena je struktura u kojoj se za svaki artikal iz korpe pravi *<li>* element. Kako se artikli dodaju u korpu ili iz nje uklanjaju, tako će se te promene odraziti i na strukturu *HTML* elementa koji se generiše pomoću ove direktive. `*ngIf` prikazuje ili sakriva element za koji je vezan u zavisnosti od vrednosti prosleđenog svojstva. `NgSwitch` je zapravo skup više direktiva koje rade zajedno `NgSwitch`, `NgSwitchCase` i `NgSwitchDefault`. Ekvivalent ove direktive je `switch` izraz u *JavaScript* jeziku. `NgSwitch` sama po sebi nije direktiva strukture već direktiva atributa ali pošto se pomoću nje koriste direktive `NgSwitchCase` i `NgSwitchDefault` ona je svrstana u direktive strukture. Direktive atributa menjaju stil prikazivanja elemenata na stranici. Ugrađena direktiva atributa `NgStyle` se koristi za menjanje *style* svojsva *HTML* elementa nad kojim je definisana. Direktive atributa se mogu i kreirati, a kao direktive se mogu definisati i funkcionalnosti koje proveravaju ispravnost forme (sinhrone i asinhrone).

3.5.7 Servisi i umetanje zavisnosti

Angluar preporučuje da komponente sadrže samo podatke specifične za šablone. Biznis logika i obrada podataka treba da se nalazi u servisima. U ovom slučaju komponente su korisnici servisa i služe da proslede šablonu podatke iz servisa za prikaz. Na neki način komponente samo delegiraju posao servisima i koriste njihove usluge. Da bi se klasa definisala kao servis ona mora da bude opisana dekoraterom `@Injectable()`. Servisi mogu da imaju zavisnost i od drugih servisa, funkcija ili vrednosti pa je tako česta upotreba servisa koji zavise od servisa *HttpClient* koji služi za slanje zahteva ka zadatim adresama. Kada je kreirana instanca neke klase, obezbedjivanje svih elemenata od kojih ta klasa zavisi se zove umetanje zavisnosti (*eng. dependecy injection*). U *Angluaru*, "umetač" (*eng. injector*) održava skladište servisa i podataka od kojih zavisi fukcionisanje svakog pojedinačnog servisa. Ako neka

komponenta zahteva korišćenje nekog servisa, umetač prvo proverava da li postoji instanca tog servisa i ako ne postoji kreira je, a zatim instancu servisa prosleđuje kroz provajdere (*eng. providers*) nazad komponenti na korišćenje. Umetač na osnovu konstruktora komponente određuje koji servisi su toj komponenti potrebni i u koju promenljivu da ih stavi, jer se servisi koji se koriste prosleđuju kao argumenti konstruktora. U tom slučaju bi kreiranje instance komponente zavisilo i od servisa.

3.5.8 Obećanja i Posmatrači

U *JavaScriptu* i *TypeScriptu* se često koriste asinhroni pozivi kojima se prosleđuju povratne funkcije pa se često dešava da nastane nešto što se naziva pakao povratnih funkcija (*eng. callback hell*). To je struktura koja je opisana u primeru 31.

```
fun1(function(x){
    fun2(x, function(y){
        fun3(y, function(z){
            ...
        });
    });
});
```

Primer 31: *Pakao povratnih funkcija*

Ovakav kod je veoma težak za čitanje i održavanje i da bi se on izbegao koriste se različite tehnike i jedna od njih je korišćenje *Obećanja* (*eng. Promise*). Uz upotrebu *Obećanja* primer 31 bi mogao da se transformiše u primer 32 pri čemu bi i funkcije `fun1()`, `fun2()i` `fun3()` morale da se preprave na odgovarajući način.

```
fun1() .then(fun2).then(fun3);
function fun() {
  return new Promise(resolve) => resolve('x vrednost'); }
function fun2(x) {
  return new Promise((resolve, reject) => resolve('y vrednost')); }
function fun3(y) {
  return new Promise((resolve, reject) => resolve('z vrednost'));
}
```

Primer 32: *Obećanja*

U primeru 32 se vidi da je kod jasniji, pregledniji i lakši za korišćenje i razumevanje. Pomoću metode `resolve()` se narednoj funkciji u nizu prosleđuje vrednost

prethodne funkcije a metoda `then()` čeka da se jedan poziv završi pre nego što drugi počne. *Obećanja* su više jednokratne funkcije i jednom kada se razreše ne može se očekivati da se od njih dobije bilo kakva informacija ili promena stanja, osim ako se ne pokrene iz početka. Takođe mana *Obećanja* je što se ne mogu prekinuti i što kao rezultatu mogu da daju samo jednu vrednost. Sve ove mane se mogu razrešiti upotrebom *Posmatrača* (*eng. Observator*). *Posmatrači* posmatraju promene stanja nad objektom nad kojim su definisani (*eng. observable*) i te promene distribuiraju svim objektima koji su na njih prijavljeni (*eng. subscribers*). Nešto slično slanju novog broja časopisa na adrese pretplatnika. U svakom trenutku, pretplatnik se može odjaviti sa liste prijavljenih na posmatranoj, isto kao što se može odjaviti sa pretplate časopisa, usled na primer isteka pretplate ili nedostatka novca za novu pretplatu.

```
let posmatrana= new Observable((posmatrac)=>{
 setInterval(() => {
   posmatrana.next('Asinhrona operacija');
 }, 2000);
 });
posmatrana.subscribe(
 next(prosledjeni_podaci),
 error(greska),
 complete()
);
```

Primer 33: *Posmatrana*

Implementacija *Posmatrača* i *Obećanja* je u *Angularu* obezbeđena pomoću biblioteke `rxjs`. Ova biblioteka služi i za prevođenje asinhronih operacija u posmatrane, za filtriranje podataka (pomoću `filter()` i `debounceTime()` funkcija) i za konvertovanje rezultata u neku drugu vrednost (`map()`, `switchMap()` funkcije). U *Angularu* dosta komponenti i servisa koristi *Posmatrače* a među njima su i moduli *HttpClient, Router i ActiveForms*.

3.5.9 Servis HTTP Client

Kada se pravi *Angular* aplikacija tako da koristi *REST API*, najbitniji servis je svakako *HTTPClient*, jer se pomoću njega mogu dalje implementirati specifični servisi koji komuniciraju sa spoljnom aplikacijom i koji prikupljaju ili šalju podatke. U tom slučaju, ovakvi servisi služe kao veza između klijentskog i serverskog dela aplikacije. Servis *HttpClient* ima obezbeđene metode za svaki tip zahteva

(*GET,POST,PUT,DELETE,..*) i svaka od ovih metoda vraća posmatranu na koju se korisnik servisa kasnije može pretplatiti.

3.5.10 Modul za rutiranje

Modul *Router* se koristi za navigaciju kroz komponente koje čine aplikaciju. Uloga koju klasa *Router* ima jeste da za konfigurisane putanje prikaže dodeljene komponente. Ta konfiguracija je globalna na nivou aplikacije i ona se uvozi u modul koren. Definisanje ruta se sastoji od nekoliko faza. Prvo se u zaglavlju *HTML* stranice definiše element `<base>` koji označava osnovnu putanju na koju bi se ostale putanje nadograđivale. U većini slučajeva to je karakter "/" a ceo element se navodi kao `<base href="/">`. Zatim se konfiguriše putanja. Konfiguracija je zapravo niz *JavaScript* objekata koji imaju određena svojstva kao što su:

- `path` - *URL* koji se prikazuje kao adresa u veb pregledaču (bez vodećeg karaktera "/")
- `component` - pripadajuća komponenta koja se inicijalizuje prilikom navigacije ka prethodno definisanom *URL*-u
- `redirectTo` - sa gore navedenog *URL*-a korisnik može biti preusmeren na ovu putanju
- `pathMatch` - navodi se ukoliko se upotrebi i `redirectTo` svojstvo i definiše kako se uparuju putanja za usmeravanje i originalna putanja. Može imati vrednosti *full* ili *prefix*.
- `children` - niz *JavaScript* objekata koji može sadržati ista svojstva. Svojstvo `children` određuje vezu roditelj-dete između *URL*-ova.
- `data` - dodatni podaci koji se prosleđuju komponenti

Kada se definiše putanja može joj se dodeliti i čuvar za neki parametar. Parametri u `path` svojstvu se označavaju prefiksom ":" pa na primer putanja `"korisnici/:id"` označava da će ako se u veb pregledaču vrši navigacija ka adresi `"/korisnici/1"`, broju *1* će moći da se pristupi preko parametra *id*. Ako se putanja definiše kao "**", ukoliko nije konfigurisan *URL* za neku traženu adresu u pretraživaču, pozvaće se komponenta koje je pridružena ovoj putanji. Ovako definisana putanja govori: "Ako ne postoji definisana putanja za taj *URL*, iskoristi mene". Konfiguracija se dalje prosleđuje modulu uz pomoć `RouterModule.forRoot()` funkcije koja se poziva prilikom uvoženja

u modul koren. Komponente se u šablonu zamenjuju pomoću elementa `<router-outlet></router-outlet>`, a pozivaju se pomoću atributa `routerLink="/definisanpath"` elementa <a>.

4 Detalji implementacije aplikacije Travelerko

4.1 Opis aplikacije Travelerko

Veb aplikacija koja se naziva *Travelerko* je namenjena putnicima koji bi obilazili razne destinacije ali nemaju sa kim da podele to iskustvo. Takođe je namenjena ljudima koji jednostavno žele da okupe veće društvo kako bi se bolje proveli prilikom obilaska, osetili sigurnije prilikom putovanja ili umanjili sebi troškove smeštaja (prema pregledu ponuda - sve je jeftinije kada se prijavi više ljudi). Ideja veb aplikacije Travelerko je da posreduje i spoji ljude sa sličnim putnim interesima gde organizacija putovanja ostaje samo na korisnicima ove aplikacije. Aplikacija Travelerko je zamišljena po principu društvene mreže. Korisnici aplikacije mogu biti neprijavljeni odnosno imati ulogu Gosta ili prijavljeni odnosno imati ulogu Travelerka. Gosti na aplikaciji imaju ograničene mogućnosti, dok se Travelercima nudi mnoštvo opcija koje su od Gostiju sakrivene. Spajanje korisnika za putovanje je omogućena samo Travelercima.

Aplikacija je podeljena u četiri modula:

1. Prijava/Registracija
2. Profil
3. Pretraga
4. Putovanje

Modul *Prijava/Registracija* je namenjen "administrativnom" radu sa korisnicima. On sadrži mogućnosti prijave na aplikaciju kako bi se iz uloge Gost prešlo u ulogu Travelerko i suprotno - odjave i prelaska iz uloge Travelerko u ulogu Gost i zatim registracije pomoću koje se kasnije omogućava prijava. Pored ovih osnovnih stvari, modul sarži i mehanizme za potvrđivanje elektonske adrese korisnika, pošto je sva komunikacija između korisnika zamišljena da ide preko elektronske pošte, kao i mehanizam za zamenu šifre ukoliko je korisnik zaboravi.

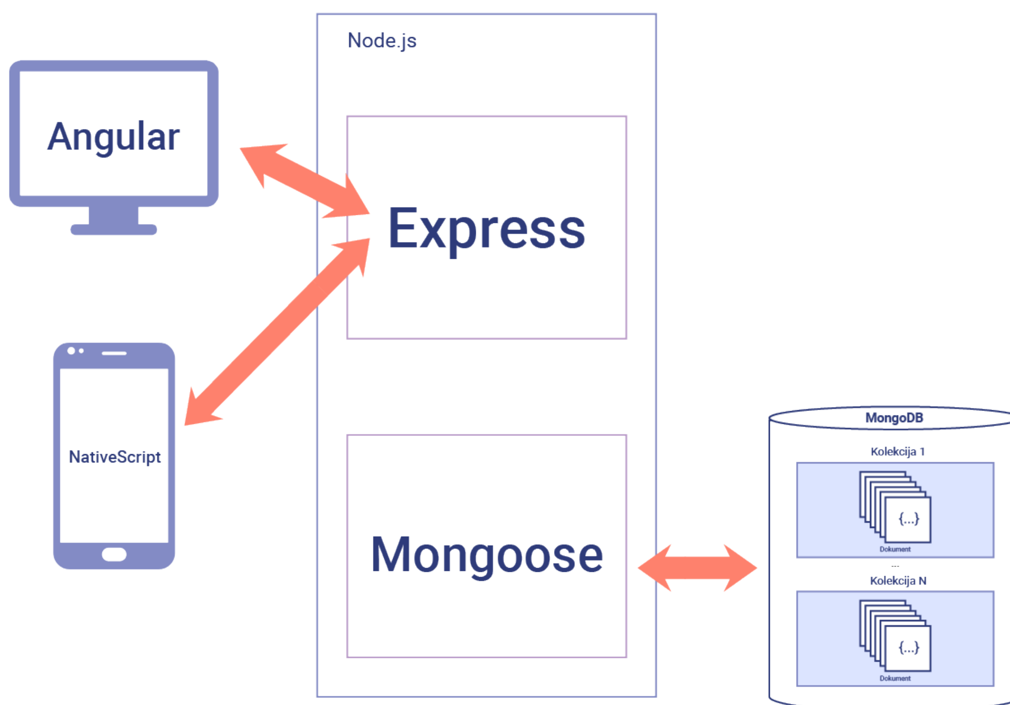
Modul *Profil* je većinom posvećen korisnicima sa ulogom Travelerko dok je jedan deo modula obezbeđen i za korisnike koji imaju ulogu Gost. Modul ima mogućnosti pregleda i izmene podataka na privatnom profilu Travelerka. U izmenu podataka se računa kako izmena informacija o Travelerku tako i dodavanje ili brisanje slika koje je Travelerko sačuvao u aplikaciji, menjanje opisa slika, menjanje korisnika i mesta označenih na slikama i izmena izgleda mape Travelerka. Gost na ovom modulu nema opciju izmene bilo kakvog sadržaja i ima samo opciju pregleda profila Travelerka. Takođe Travelerko može menjati samo informacije koje je on postavio u aplikaciji.

Modul *Pretraga* se odnosi na pretragu putovanja i ovaj modul je dostupan i Gostu i Travelerku s tim što Travelerko ima dodatne mogućnost prijave za Saputnika na putovanju. Saputnik je naziv veze između dva Travelerka, nešto slično prijateljstvu dok Saputnik u kontekstu prijave na putovanju ima malo drugačiji smisao. Ako se Travelerko prijavi kao saputnik na putovanju, ta prijava može a i ne mora rezultovati Saputničkom vezom između dva Travelerka. Pretaga se vrši na principu popunjavanja forme gde je rezultat pretrage lista putovanja koja odgovaraju zadatim parametrima ili ukoliko takvo putovanje ne postoji, lista svih ostalih putovanja rangiranih prema kriterijumu od destinacije najbliže traženoj do destinacije koja je najudaljenija traženoj destinaciji.

Modul *Putovanje* je i najzanimljiviji modul jer upravo on predstavlja suštinu cele aplikacije. On je jako sličan modulu Profil. Ceo modul je zadužen za sve radnje vezane za pojedinačno Putovanje kao što su pravljenje putovanja, pregled pojedinačnog putovanja, ažuriranje putovanja, prijava za Saputnika na Putovanju, pregled svih prijavljenih saputnika i brisanja putovanja.

4.2 Arhitektura aplikacije

Travelerko aplikacija je napravljena korišćenjem tehnologija *MEAN* steka. Aplikacija se sastoji iz tri dela. Jedan deo čini baza podataka *MongoDB*, drugi deo je serverska aplikacija koja je implementirana pomoću platforme *Node.js* i modula *express* po *REST* principima a treći deo je klijentska aplikacija, koji se takođe može podeliti na veb aplikaciju, rađenu u *Angular* radnom okviru i mobilnu aplikaciju rađenu u *NativeScript* radnom okviru. Ovi delovi su fizički razdvojeni i nalaze se na različitim IP adresama. Na slici 7 je prikazan način na koji ova tri razdvojena dela međusobno komuniciraju.



Slika 7: *Arhitektura aplikacije*

Angular veb aplikacija pomoću svojih servisa šalje zahteve *Node.js* serverskoj aplikaciji. Serverska aplikacija prihvata zahtev, razrešuje njegovu adresu i njegovu metodu. Ukoliko je definisana *express* ruta nad tom adresom i sa tom metodom, ona obrađuje zahtev, i po potrebi pomoću *Mongoose ODM*-a komunicira sa bazom podataka i šalje odgovor nazad klijentskoj aplikaciji. Klijentska aplikacija pomoću posebnih mehanizama prihvata odgovor i reaguje u odnosu na taj odgovor. Na sličan način i mobilna aplikacija komunicira sa serverskom aplikacijom. Iako su ovo tri zasebna dela, zajedno su oni korisniku aplikacije predstavljeni kao celina. Korisnik ima uvid samo u klijentske aplikacije, odnosno u veb ili mobilnu aplikaciju i samo klijentska aplikacija ima mogućnost direktne interakcije sa korisnikom, kao što samo serverska aplikacija ima interakciju sa bazom podataka.

4.3 Baza podataka

Instanca *MongoDB* je postavljena na platformi koja se naziva *MongoDB Atlas*, pa je organizacija replika i klastera prepuštena predefinisanoj konfiguraciji platforme. Na *Atlas* klasteru pod nazivom “travelerkoDB” definisana su dva korisnika sa različitim privilegijama. Jedan korisnik ima *admin* privilegije koje uključuju pravljenje i brisanje novih baza i kolekcija unutar njih a drugi korisnik ima privilegije samo za čitanje i pisanje i služi za povezivanje aplikacije sa bazom podataka. Baza podataka koju aplikacija koristi se zove “travelerko” i to je jedina baza na koju korisnik iz aplikacije može da se poveže. Baza ima više definisanih kolekcija. Kolekcije koje se u njoj nalaze su :

- korisnici
- mesta
- slike
- putovanja

Kolekcija *korisnici* služi da sačuva dokumente u kojima se nalaze opšte informacije o registrovanim korisnicima aplikacije (Travelercima) i njihovim sadržajima u aplikaciji. Jedan dokument iz kolekcije *korisnici* ima sledeća polja koja su definisana odgovarajućim tipom:

- _id - tip *ObjectID* koji jedinstveno određuje dokument na nivou kolekcije
- korisnicko_ime - niska
- sifra - niska
- ime - niska
- prezime - niska
- email - niska
- datum_rodjenja - tip *Date*
- salt - niska
- pol - niska
- datum_registracije - tip *Date*
- o_sebi - niska

- lokacija - niska, predviđeno je da čuva referencu na polje _id kolekcije *mesta*
- potvrdjena_registracija - tip *Boolean*, inicijalno postavljeno na vrednost *false*
- profilna - tip *ObjectID* koji služi da čuva referencu na _id polje kolekcije *slike*
- mesta - Niz objekata oblika `{"mesto":mesto_vrednost, "tip":tip_vrednost}`, gde je mesto_vrednost niska i čuva referencu na polje _id kolekcije *mesta* a tip_vrednost je takođe niska
- saputnici - Niz koji sadrži vrednosti tipa *ObjectID* koji čuvaju referencu na polje _id kolekcije *korisnici*

Kolekcija *slike* služi za čuvanje fizičke adrese na kojoj se nalazi slika na serveru kao i dodatnih podataka o istoj i sadrži sledeća polja koja su definisana odgovarajućim tipom:

- _id - tip *ObjectID* koji jedinstveno ordeđuje dokument u kolekciji
- putanja - niska
- opis - niska koja je inicijalno prazna
- datum_ubacivanja - tip *Date*, automatski se popunjava datum i vreme ubacivanja dokumenta u kolekciju
- mesto - niska koja inicijalno ima vrednost *null* a predviđeno je da čuva reference na _id kolekcije *mesta*
- putovanje- tip *ObjectID*, inicijalno ima vrednost *null* a predviđeno je da čuva referencu na polje _id kolekcije *putovanja*
- korisnici - Niz vrednosti koje su tipa *ObjectID* i koje čuvaju reference na _id polje kolekcije *korisnici*
- vlasnik - tip *ObjectID*, inicijalna vrednost je *null* a predviđeno je da čuva referencu na polje _id kolekcije *korisnici*
- javna - tip *Boolean*, inicijalno postavljeno na vrednost *true*
- mimetype - niska
- velicina - tip *Number*

Kolekcija *mesta* sadrži sledeća polja koja su definisana odgovarajućim tipom:

- _id - niska
- labela - niska
- naziv - niska
- longitude - tip *Number*
- latitude - tip *Number*
- koordinate - niz od dve vrednosti tipa *Number*
- drzava - niska
- drzava_skraceno - niska
- kontinent - niska

Nad kolekcijom *mesta* je postavljen prostorni *2dsphere* indeks nad poljem koordinate kako bi se omogućili geoprostorni upiti koji su potrebni za rad aplikacije. Kolekcija *putovanja* sadrži sledeća polja koja su definisana odgovarajućim tipom:

- _id - tip *ObjectID* koji jedinstveno ordeđuje dokument u kolekciji
- naziv - niska
- opis - niska
- mesto_od - niska, referenca na _id polje kolekcije *mesta*
- mesto_do - niska, referenca na _id polje kolekcije *mesta*
- datum_od - tip *Date*
- datum_do - tip *Date*
- organizator - tip *ObjectID*, referenca na _id polje kolekcije *korisnici*
- smestaj - Niz niski
- prevoz - Niz niski
- prijavljeni_saputnici - Niz vrednosti tipa *ObjectID* koje čuvaju reference na _id polje kolekcije *korisnici*
- izabrani_saputnici - Niz vrednosti tipa *ObjectID* koje čuvaju reference na _id polje kolekcije *korisnici*

Jedan dokument kolekcije *korisnici* označava jednog Travelerka. Kako Travelerko na svom profilu ima mogućnosti ubacivanja sadržaja kao što su slike, dodavanja mesta koja je posetio ili mesta koja želi da poseti, kao i dodavanje putovanja i saputnika, kolekcija *korisnici* je uvezana sa svim ostalim kolekcijama. Jedan Travelerko može imati samo jednu profilnu sliku pa se zato referenca na _id polje kolekcije *slike* čuva pod svojstvom profilna kolekcije *korisnici*. Takođe, jedan Travelerko se nalazi na jednoj lokaciji dok se na jednoj lokaciji može nalaziti više Travelerka. Zbog toga je bilo optimalnije lokaciju Travelerka pamtiti u kolekciji *korisnici*, nego da se svi Travelerci pamte u dokumentu kolekcije *mesta* koje označava lokaciju. Isto važi i za svojstvo mesta, gde uz _id dokumenta iz kolekcije stoji i tip mesta pa je kombinacija mesto-tip jednistvena na nivou dokumenta. Jedan Travelerko može biti u saputničkom odnosu sa više Traveleraka pa se zato u svojstvu saputnici kolekcije *korisnici* pamti niz _id svojstva kolekcije *korisnici*. Jedna slika može imati samo jednog vlasnika (Travelerka) dok jedan Travelerko može biti vlasnik više slika, s toga se veza vlasnik pamti u kolekciji *slike*. Jedna slika može biti deljena između više Traveleraka i može biti "uslikana" samo na jednoj lokaciji, odnosno na jednom putovanju, dok jedno putovanje i jedno mesto mogu imati više slika. Zbog toga se reference na _id kolekcija *mesta* i *putovanje* pamte u kolekciji *slike* a ne obrnuto. Jedno putovanje može imati jednu početno mesto i jedno krajnje mesto, dok jedno mesto može biti početna ili krajnja tačna na više putovanja. Jedno putovanje ima samo jednog organizatora (Travelerka), a može imati više prijavljenih i odabraniih saputnika (Traveleraka).

4.4 Serverska aplikacija

Node.js serverska aplikacija je podignuta na zasebnom serveru. Pomoću *express generatora* je napravljen kostur aplikacije kome su kasnije ubačeni dodatni moduli. Moduli koji se koriste u serverskoj aplikaciji su:

- *express* - koristi se kao pomoć prilikom pravljena veb servera
- *mongoose* - koristi se prilikom komunikacije sa bazom podataka
- *body-parser* - koristi se kao posrednička funkcija
- *passport* - koristi se kao pomoć prilikom procedure prijavljivanja
- *passport-local* - koristi se kao pomoć prilikom procedure prijavljivanja
- *express-jwt* - koristi se za prihvatanje i dekriptovanje Barier Tokena-a
- *jsonwebtoken* - koristi se za pravljene *JWT* tokena
- *nodemailer* - koristi se kao alata za slanje elektronske pošte iz aplikacije
- *multer* - koristi se kao alat sa kladištenje dokumenata
- *@google/maps* - koristi se kao biblioteka za povezivanje na servis Gugl mape (*eng. Google Maps API*)

Serverska aplikacija je napravljena po REST principima i prihvata zahteve ka sledećim adresama:

Metoda	Putanja	Opis
POST	/registracija	Prihvata podatke od klijenta i čuva ih u bazi podataka, šalje se elektronska pošta korisniku sa daljim uputstvima.
POST	/prijava	Provera ispravnost prosleđenih kredencijala i ako su ispravni kao odgovor šalje generisani *JWT* token.
POST	/zaboravljenasifra	Za traženog korisnika generiše jednosatni *JWT* token i šalje ga putem elektronske pošte na prijavljenu adresu, zajedno sa uputstvima za poništavanje šifre.

POST	/promenisifru	Zamenjuje trenutnu šifru sa šifrom poslatom u telu zahteva, primenjujući na novu šifru mehanizme heširanja.
GET	/jedinstvenokorisnickoime /:korisnicko_ime	Proverava da li postoji dokument koji ima istu vrednost polja korisnicko_ime kao i istoimeni prosleđeni parametar.
GET	/jedinstveniemail/:email	Proverava da li postoji dokument koji ima istu vrednost polja *email* kao i istoimeni prosleđeni parametar.
GET	/aktivirajnalog	Postavlja vrednost polja potvrdjena_registracija na *true*
GET	/profil/:korisnickoime /mapa	Iz baze podataka izvlači podatke o ubačenim mestima korisnika sa korisničkim imenom :korisnickoime i vraća podatke klijentu, kako bi ih klijent prikazao na mapi.
GET	/profil/:korisnickoime /omeni	Vraća dokument iz kolekcije *korisnici* čije svojstvo korisnicko_ime odgovara parametru :korisnickoime.
GET	/profil/:korisnickoime /profilna	Vraća dokument iz kolekcije *slike* koji odgovara profilnoj slici korisnika za korisničkim imenom :korisnickoime.
GET	/profil/:korisnickoime /slike	Preuzima dokumente kolekcije *slike* koji odgovaraju svim slikama koje su u vlasništvu korisnika sa korisničkim imenom :korisnickoime, ili slike na kojima je korisnik označen a pritom su one javne.
PUT	/profil/uredi/omeni	Zamenjuje informacije o korisniku koje se prikazuju na njegovom profilu.
POST	/profil/uredi/profilna/nova	Zamenjuje informaciju o profilnoj slici korisnika i prikazuje istu na njegovom profilu.
DELETE	/profil/obrisi/	Briše profil korisnika i onemogućava mu dalje prijavljivanje.
PUT	/profil/uredi/mesta	Uređuje mapu korisnika koja se prikazuje na profilu korisnika
POST	/profil/slike/	Čuva slike koje je korisnik uneo na svoj profil.

PUT	/profil/slike/:idslike	Menja dodatne informacije o slici koja se nalazi na profilu korisnika.
DELETE	/profil/slike/:idslike	Briše slike koje su u vlasništvu korisnika ili skida oznaku sa slika koje nisu u vlasništvu korisnika koji pokreće operaciju brisanja.
GET	/profil/slike /postaviProfilnu/:idSlike	Postavlja izabranu sliku kao profilnu i prikazuje je na profilu korisnika koji je pokrenuo operaciju postavljanja slike.
GET	/profil/:korisnickoime /saputnici	Prikazuje sve korisnike koji su u saputničkom odnosu sa korisnikom čije je korisničko ime :korisnickoime.
PUT	/profil/saputnici	Dodaje novog saputnika korisniku koji pokreće operaciju dodavanja.
DELETE	/profil/saputnici/:saputnik	Briše saputničku vezu između dva korisnika (Travelerka).
GET	/profil/saputnik/:saputnik	Proverava odnos između dva korisnika (Travelerka).
GET	profil/:korisnickoime /putovanja/:tip	Prikazuje informacije o putovanjima korisnika čije je korisničko ime :korisnickoime. Parametar tip koji se prosleđuje služi da bliže odredi tip putovanja - da li se prikazuju sva putovanja, samo ona koja je korisnik organizovao, samo ona na koja se prijavio i ona na kojima je bio saputnik
POST	/pretraga	Na osnovu prosleđenih parametara vrši se proces pretrage putovanja i vraćaju se rezultati koji približno zadovoljavaju upit.
GET	/putovanja/:idputovanja	Prikazuje informacije o putovanju čiji je jedinstveni identifikator :idputovanja.
PUT	/putovanja/:idputovanja	Menja informacije o putovanju čiji je jedinstveni identifikator :idputovanja.
DELETE	/putovanja/:idputovanja	Briše infomacije o putovanju čiji je jedinstveni identifikator :idputovanja.
POST	/putovanja	Pravi novo putovanje.

GET	/putovanja/:idputovanja /prijava	Prijavljuje korisnika koji pokreće operaciju prijave kao saputnika na putovanju čiji je jedinstveni identifikator :idputovanja.
DELETE	/putovanja/:idputovanja /prijava	Odjavljuje korisnika koji pokreće operaciju odjave sa putovanja čiji je jedinstveni identifikator :idputovanja
GET	/putovanja/:idputovanja /saputnik/:idkorisnika	Organizator putovanja bira saputnika sa jedinstvenim identifikatorom :idkorisnika na putovanju čiji je jedinstveni identifikator :idputovanja.
DELETE	/putovanja/:idputovanja /saputnik/:idkorisnika	Organizator putovanja briše saputnika sa jedinstvenim identifikatorom :idkorisnika na putovanju čiji je jedinstveni identifikator :idputovanja.

Struktura serverske aplikacije je organizovana na sledeći način:

- bin - sadrži *www.js* skript koji inicijalizuje server
- config - sadrži *JavaScript* fajl u kome su sačuvana podešavanja kao što je niska za povezivanje na bazu podataka ili ključ koji se koristi za heširanje tokena
- models - *Mongoose* modeli i njihova logika
- modules - aplikacijska logika i procesi, funkcije koje se pozivaju na definisanim rutama
- node_modules - svi modeli koje serverska aplikacija koristi
- public - služi za smeštanje dokumenata kojima se može pristupiti i pomoću zahteva koji dolaze sa drugih IP adresa
- routes - definisane rute, logicki podeljene prema modulima
- app.js - konfiguracija servera
- package.json - metapodatak koji opisuje koren aplikacije

Sve povratne funkcije koje se pozivaju prilikom prihvatanja zahteva su organizovane u četiri modula: index, profil, putovanje i pretraga. Od osnovnih posredničkih funkcija postavljene su *body-parser* funkcije za parsiranje tela zahteva i podešen je *public* direktorijum koji opslužuje statičke fajlove (u ovom slučaju slike). Povezivanje sa bazom podataka je uspostavljeno na način prikazan na primeru 34.

```
mongoose.Promise = global.Promise;
mongoose.connect(config.database,{ autoIndex: false }).then(() =>
   console.log("Uspostavljena je konekcija sa bazom");
}).catch(err => {console.log('Server ne moze da se poveze sa bazom'
   ); process.exit(); });
```

Primer 34: *Povezivanje sa bazom pomoću Mongoose ODM*

Pošto je ovo aplikacija na koju će pristizati zahtevi sa drugih servera, njihovo omogućavanje je prikazano na primeru 35.

```
app.use(function(req, res, next) {
   res.header("Access-Control-Allow-Origin", "*");
   res.header("Access-Control-Allow-Headers", "Access-Control-Allow
      -Headers, Origin,Accept, X-Requested-With, Content-Type,
      Access-Control-Request-Method, Access-Control-Request-Headers
      , Authorization");
   res.header("Access-Control-Allow-Credentials", "true");
   res.header("Access-Control-Allow-Methods", "GET,HEAD,OPTIONS,
      POST,PUT,DELETE");
   next();
 });
```

Primer 35: *Konfiguracija zahteva*

Isto podešavanje se mora napraviti i za direktorijum *public* jer se u njemu čuvaju fotografije koje će se učitavati na klijentskoj aplikaciji. Podešavanje zahteva ka statičkom *public* direktorijumu je prikazano u primeru 36.

```
app.use(express.static('public', {
  setHeaders: function(res, path) {
    res.set("Access-Control-Allow-Origin", "*");
    res.set("Access-Control-Allow-Headers", "Access-Control-Allow-
        Headers, Origin,Accept, X-Requested-With, Content-Type,
        Access-Control-Request-Method, Access-Control-Request-
        Headers, Authorization");
    res.set("Access-Control-Allow-Credentials", "true");
    res.set("Access-Control-Allow-Methods","PUT,POST,GET,DELETE,
        OPTIONS");
} }));
```

Primer 36: *Konfiguracija zahteva ka public direktorijumu*

Kada klijentska aplikacija šalje zahteve ka drugom serveru, veb pregledač šalje prvo probni zahtev tipa *OPTIONS* čiji je zadatak da utvrdi da li je bezbedno poslati zahtev ka tom serveru i da li taj server prihvata zahteve, što se podešava putem zaglavlja odgovora. `res.set("Access-Control-Allow-Origin", "*")` obaveštava da mogu da pristignu zahtevi sa svih servera, pomoću `res.set( "Access-Control-Allow-Headers", "Access-Control-Allow-Headers, Origin, Accept, X-Requested-With, Content-Type, Access-Control-Request-Method, Access-Control-Request-Headers, Authorization")` se obaveštava koja svojstva je dozvoljeno poslati u zaglavlju zahteva a `res.header ("Access-Control-Allow-Methods", "GET,HEAD,OPTIONS,POST,PUT,DELETE")` obaveštava nazad veb pregledač o dozvoljenim metodama. U odnosu na primljeni odgovor pregledač dalje šalje pravi zahtev ili javlja grešku da zahtev tog tipa ka tom serveru nije dozvoljen.

Prilikom registracije, radi bezbednosti, izabrana šifra korisnika se hešira pomoću modula *crypto* i tako heširana se čuva u bazi podataka. Heširanje je prikazano na primeru 37.

```
var crypto = require('crypto');
sifra=sifra.trim();
this.salt = crypto.randomBytes(16).toString('hex');
this.sifra = crypto.pbkdf2Sync(sifra, this.salt, 1000, 64, 'sha512'
  ).toString('hex');
```

Primer 37: *Heširanje šifre*

Svojstvo `salt` ima zadatak da čuva jednistvenu nisku (tajni ključ) pomoću koje se zadata šifra kriptuje podsredstvom *PBKDF2* funkcije i *SHA512* algoritma.

Za prijavljivanje odnosno autentifikaciju koriste se moduli *passport* i *passport-local*. Prilikom inicijalizacije lokalne strategije definiše se funkcija kojim se utvrđuje

postojanje korisnika sa prosleđenim korisničkim imenom i šifrom. Funkcija kao rezultat treba da vrati funkciju `done()` kojoj se redom kao argumenti prosleđuju greška, korisnik (ako postoji) i dodatne informacije.

```
var passport = require('passport');
var LocalStrategy =require('passport-local').Strategy;
app.use(passport.initialize());
passport.use(new LocalStrategy({
    usernameField: 'korisnicko_ime',
    passwordField: 'sifra'},
  function(username, password, done) {
    Korisnici.findOne({ korisnicko_ime: username }, function(err,
       korisnik) {
      if (err) { return done(err); }
      if (!korisnik) {
      return done(null, false, { message: 'Pogresno korisnicko ime.
         ' });}
      if (!korisnik.validirajSifru(password)) {
      return done(null, false, { message: 'Pogresna sifra' });}
      return done(null, korisnik);});
      }));
```

Primer 38: *Inicijalizacija* passport *objekta i konfiguracija lokalne strategije*

Sama autentifikacija se odvija u modelu, nakon što server prihvati zahtev sa klijenta. U modelu se poziva metoda `authenticate()` objekta *Passport* i njoj je prosleđuje niska koja označava strategiju koju treba koristiti prilikom autentifikacije. Pošto je za potrebe aplikacije definisana samo lokalna strategija, u aplikaciji se prosleđuje niska *"local"*. Kao drugi argument metoda `authenticate()` prima povratnu funkciju dizajniranu da ogovara potrebama aplikacije. U povratnoj funkciji se ispituje uspešnost lokalne strategije, odnosno, da li je autentifikacija rezultovala uspehom ili nije. Ukoliko je autentifikacija uspešna preduzimaju se određene radnje kako bi se obezbedila dalja autorizacija a ukoliko nije, šalje se kao odgovor klijentu status *401 Unauthorized*.

```
var jwt = require('jsonwebtoken');
Korisnici.methods.generisiToken = function() {
return jwt.sign({ _id: this._id, korisnicko_ime: this.
   korisnicko_ime, }, config.secret); };
```

Primer 39: *Metoda modela* Korisnici *koja pomoću objekta* jsonwebtoken *generiše token*

Serverska aplikacija je dizajnirana tako da ne pamti stanja (*eng. stateless*). Ipak, za potrebe Travelerko aplikacije je potrebno odrediti dva nivoa pristupa aplikaciji tj. autorizaciju. Kako samo prijavljeni korisnici treba da imaju pristup odeđenim delovima aplikacije, a na serverskoj aplikaciji se nigde ne pamti korisnička sesija, prilikom authentifikacije generiše se *JWT* (*eng. JSON Web Token*) pomoću modula *jsonwebtoken*, i taj token se kao odgovor šalje nazad klijentu. *JWT* je heširana niska koja se sastoji iz tri dela koja su odvojena tačkama, a najbitniji je drugi deo koji čuva podatke _id i korisnicko_ime prijavljenog korisnika. Pomoću ovog tokena i pomoću modula *express-jwt* se vrši autorizacija. U zaglavlju zahteva koji pristiže sa klijenta za rute za koje je autorizacija potrebna se šalje i token. Zadatak objekta modula *express-jwt* koji se rutama prosleđuje kao posrednička funkcija je da proveri da li taj token postoji i ako postoji da preuzme podatke iz njega i sačuva ga u definisanom svojstvu zahteva koji se dalje prosleđuje modulu za obradu zahteva. *JWT* se može iskoristi za mnogo toga i u aplikaciji Travelerko se koristi i prilikom potvrđivanja elektonske adrese i prilikom poništavanja šifre, samo što se u ova dva slučaja on ne prosleđuje putem odgovora klijentu već preko elektronske pošte.

```
var jwt = require('express-jwt');
var auth = jwt({
  secret: config.secret,
  userProperty: 'travelerko'
});
var auth_opcioni=jwt({
  secret: config.secret,
  userProperty: 'travelerko',
  credentialsRequired: false
});

router.get('/:korisnickoime/slike',auth_opcioni,profilModul.
    prikaziSlike);
router.put('/uredi/omeni', auth,profilModul.urediOMeni);
```

Primer 40: *Upotreba* express-jwt

Slanje elektonske pošte sa uputstvima za potvrđivanje elektronske adrese ili menjanje šifre se radi pomoću modula *nodemailer*. Elektronska pošta se šalje pomoću objekta *Transport*. Kada se kreira objekat *Transport*, potrebno je da se proslede informacije o serveru elektronske pošte sa kog se šalje elektronska pošta. U ovom slučaju kao server elektronske pošte se koristi *gmail* pa je potrebno konfigurisati objekat *Transport* na način prikazan u primeru 41. Konkretno elektronska pošta se šalje tako što se nad objektom *Transport* pozove metoda

`sendMail()` koja kao argument prima konfiguraciju elektronske pošte. Opcije koje se prosleđuju uključuju adresu sa koje se šalje elektronska pošta, adresu na koju se šalje, može se dodati opcija *subject*. Telo elektronske pošte, odnosno njen sadržaj se konfiguriše pomoću svojstva *html*. Kao drugi parametar prosleđuje se povratna funkcija pomoću koje se utvrđuje da li je elektronska pošta uspešno poslata.

```
var transporter = nodemailer.createTransport({
       service: 'gmail',
       auth: {
              user: 'mojaemail@adresa'
              pass: 'mojasifra'
           }
      });
```

Primer 41: *Konfigurisanje* transport *objekta*

Slike koje se u aplikaciji koriste se čuvaju na serverskoj aplikaciji, što trenutno i nije najbolje rešenje. Klijent ih šalje serveru u telu zahteva kao fajlove, a server ih obrađuje pomoću modula *multer* i automatski ih skladišti na definisanoj lokaciji na serveru. Informacije o uskladištenim slikama se se čuvaju u svojstvu *files* objekta *Request*. Korišćenje modula *multer* je prikazano u primeru 42.

```
var multer = require('multer');
var Storage = multer.diskStorage({
    destination: function(req, file, callback) {
        callback(null, "./public/images/slike");
    },
    filename: function(req, file, callback) {
        callback(null, file.fieldname + "_" + Date.now());
    }
});
var upload = multer({
    storage: Storage
}).array("slike");
router.post("/slike/",auth,upload,profilModul.dodajSlike);
```

Primer 42: *Upotreba multer modula i njegova upotreba prilikom definisanja rute*

Najvažniji aspekt aplikacije jeste pretraga. Inicijalna ideja je da se podudaraju samo putovanja koja se poklapaju po destinaciji i drugim navedenim parametrima, međutim, zbog mogućnosti da se desi da neka pretraga ne vrati ni jedan rezultat za traženu destinaciju, druga ideja je da se pretragom upare i destinacije koje bi bile relativno blizu. Ostaje problem kako odrediti šta je to relativno blizu. Ako na primer

korisnik1 želi da putuje u destinaciju1, a korisnik2 želi da putuje na destinaciju2 i ako se pretpostavi da su te destinacije jedna od druge vazdušnom linijom udaljene manje od 50km, postavlja se pitanje da li samo kilometraža može da utiče na to da se te dve destinacije spoje. Jer iako je udaljenost u kilometrima relativno mala, postoji mogućnost da je jako teško ili čak nemoguće doći od destinacije1 do destinacije2 ili da se te dve destinacije nalaze u dve različite države. Prema toj logici destinacija1 i destinacija2 nisu uparive. Opet postoji i suprotan primer gde se Kopenhagen nalazi u Danskoj a Malme u Švedskoj a vozom između ta dva grada i u ovom slučaju države je potrebno nekih 30 minuta vožnje što dovodi do zaključka da su Kopenhagen i Malme uparivi. Za tu potrebu se koristi servis Gugl matrica udaljenosti (*eng. Google Distance Matrix API*) kome će se početna tačka proslediti iz upita pretrage dok će se kao krajnja tačka iskoristi lokacija nekog putovanja koje je još uvek nije počelo. Međutim kako se ne bi slali upiti gde je na primer početna tačka, lokacija petrage grad Toronto u Kanadi a krajnja Tokio u Japanu (postoji putovanje kome je Tokio destinacija), pre korišćenja matrice udaljenosti treba ograničiti krajnje tačke a to se radi pomoću geoprostornih upita. Pomoću njih će iz baze prvo izolovati mesta koja su u radijusu od 50km a zatim će se pretraživati putovanja ka tim mestima i ta putovanja sortirati po minimalnom vremenu potrebnom od tražene destinacije do približne destinacije u radijusu. Tako se pretraga odvija u tri koraka:

1. korak - Pretraga se vrši po svim prikupljenim parametrima od klijenta. Ukoliko takva vrsta pretrage daje neki rezultat, on će biti vraćen nazad klijentu, ukoliko ne vraća pokrenuće se drugi korak.

2. korak - Proverava se mesto do kojeg se putuje, ukoliko on postoji u bazi prosleđuje se međukoraju pretrage po radijusu, a ukoliko ne postoji prvo se kreira a zatim prosleđuje međukoraku pretrage po radijusu.

    - pretraga po radijusu - Iz baze podataka se pomoću geoprostornih upita pretražuju mesta koja u radijusu od 50km od početne tačke pretrage. Ukoliko postoje mesta u tom radijusu prelazi se na međukorak obrade podataka matricom a aukoliko ne postoje prelazi se na treći korak.

    - pretraga matricom - Pretraga matricom se vrši pomoću Gugl matrice udaljenosti, podešenu tako se vraćeni rezultati odnose na tip javni prevoz (*eng. transit mode*). Dalje se vraćeni rezultati opet filtriraju po kriterijumu tako da ulaze u obzir samo rezultati kod kojis svojstvo koje označava vreme (*eng. duration*) ima vrednost manju od 5400 sekundi (90 minuta). Ukoliko ovaj međukorak da rezultate, oni se prosleđuju nazad

klijentu, a ukoliko ne vrati, ovako izmenjeni upiz za krajnju destinaciju se prosleđuje trećem koraku.

3. korak - Iz upita pretrage se uklanjaju informacije o smeštaju i prevozu i na taj način se po poslednji put pokušava sa pretragom putovanja. Rezultat upita se vraća klijentu.

4.5 Klijentska aplikacija

4.5.1 Veb aplikacija

Veb aplikacija je napravljenja u *Angular.io* radnom okviru pomoću alata *Angular CLI*. Pravljenje aplikacije uz pomoć *Angular* radnog okvira se svodi na praćenje uputstava za korišćenje jer se sva komplikovanija logika nalazi na serverskoj aplikaciji. Od dodatnih paketa aplikacija koristi *@agm/core* koji služi kao dodatak za Gugl mape, *@ng-bootstrap* koji je podrška za biblioteku *Bootstrap 4* koja se koristi za stilizovanje izgleda stranice pomoću *CSS-a* i paket *ng2-img-cropper* pomoću koga se profilne slike smanjuju na odgovarajuće dimenzije i kao takve se prosleđuju serveru koji ih skladišti. Arhitektura veb aplikacije je organizovana u nekoliko modula:

- app.module - Modul koren aplikacije i njegov zadatak je da objedini sve ostale module tako što će ih učitati u sebe.
- ruter.module - Pomoću njega se definišu putanje za kretanje kroz aplikaciju.
- layout.module - Služi da objedini sve grafičke elemente koji se ponavljaju na stranicama kao što su mapa, glavni navigacioni meni, i forma za putovanja.
- pocetna.module - Modul koji služi samo za prikaz komponenti koje čine početnu stranicu.
- prijavaregistracija.modul - Modul koji se zadužen za radnje vezane za prijavu, odjavu, registraciju, poništavanje šifre i potvrđivanje elektronske adrese.
- profil.module - Modul koji je vezan za sve radnje koje su vezane sa korisničkim profilom kao što su izmena profila i ubacivanje raznih tipova sadržaja
- pretraga.module - Pomoću njega se prikazuje stranica pretrage i stranica rezultata petrage putovanja kao i sama operacija pretrage se izvršva u okviru ovog modula.
- putovanja.module - Koristi se za kreiranje, izmenu, brisanje putovanja kao i za kontrolu prjavljenih saputnika i izabranih.

router.module je u aplikaciji implementiran pomoću biblioteke *RouterModule* i pomoću servisa *auth.guard*. Zadatak ovog modula je da omogući kretanje kroz aplikaciju odnosno da za svaku definisanu putanju pokrene odgovarajuću komponentu i pomoću njega se zapravo u glavnoj (korenoj) komponenti učitavaju ostale komponente koje se koriste u aplikaciji. Definisanje putanje je dato u primeru 43. Putanja se definiše pod svojstvom *path* i to bez vodećeg karaktera “/” ukoliko je

on definisan kao bazna adresa aplikacije u zaglavlju *HTML* stranice. Svojstvom *component* se govori koja komponenta treba da se prikaže kada se aplikacija nađe na toj putanji. Pored adrese, putanji se mogu proslediti i dodatni parametri koji se definišu tako što se ispred njihovog imena nalazi karakter ":". Putanja može imati i podputanje koje se definišu pod svostvom *children* kao što je prikazano u primeru 43. U tom primeru ukoliko se aplikacija nađe na adresi `/uredi/profilna` učitaće se komponenta *UrediProfilnaComponent*. Putanja "**" označava svaku drugu putanju koja nije navedena i uglavnom se koristi za prikaz stranice koja obaveštava o nepostojećoj putanji.

```
const putanje: Routes = [
  {path: '', component: PocetnaComponent},
  {path: 'prijava', component: PrijavaComponent},
          ...
  {path: 'uredi', component: UrediComponent, canActivate: [
     PermisijeGuard],
    children: [
      {path: '', component: UrediOMeniComponent},
      {path: 'profilna', component: UrediProfilnaComponent},
                     ...
    ]},
  {path: 'potvrdjenemail/:aktivacioni_token', component:
     PotvrdjenemailComponent},
                     ...
  { path: '**', redirectTo: '' }
];
```

Primer 43: *Primer definisanja putanje pomoću biblioteke RouterModule*

Svojstvo *canActivate* prima kao parametar niz implementiranih servisa *auth.guard* koji se u slučaju aplikacije zove *PermisijeGuard*. Zadatak servisa *PermisijeGuard* je da za stranice za koje je potrebna autorizacija proveri da li je trenutni korisnik poseduje i ako je poseduje tj. ako je prijavljen biće mu omogućen pristup stranici nad kojoj je postavljen servis a ukoliko nije, prosledиće korisnika na odgovarajuću adresu.

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot,
   Router } from '@angular/router';
import { Observable } from 'rxjs/Observable';
import {LoginregistracijaService} from './loginregistracija.service
   ';
@Injectable()
export class PermisijeGuard implements CanActivate {
  constructor(private router: Router, private login:
      LoginregistracijaService) { }
  canActivate(route: ActivatedRouteSnapshot, state:
      RouterStateSnapshot) {
    if (this.login.prijavljen()) {return true; }
    this.router.navigate(['/prijava'], { queryParams: { returnUrl:
       state.url }});
    return false;}
}
```

Primer 44: *Primer servisa PermisijeGuard*

Da bi se komponente prikazale na dogovarajućem mestu pomoću modula *ruter*, neophodno je da se u šablonu glavne komponente postavi *HTML* element `<router-outlet></router-outlet>`.

Prijava se sastoji iz nekoliko delova:

- korisnik popunjava formu za prijavu
- aplikacija preuzima popunjene podatke i putem servisa ih šalje serveru
- server proverava podatke i ukoliko ne postoji korisnik sa datim podacima, obaveštava nazad klijenta kroz odgovor zahteva, a ukoliko postoji korisnik server generiše *JWT* token koji se putem odgovora prosleđuje nazad klijentu
- klijent čuva *JWT* token u lokalnom skladištu kako bi kasnije mogao da ga uz svaki sledeći zahtev šalje nazad klijentu

Sa tehničke strane provera da li je klijent prijavljen podrazumeva proveru da li token postoji u lokalnom skladištu pretraživača. Suprotno, odjava se radi tako što se taj token obriše iz skladišta čime se onemogućuje njegovo slanje uz zahteve. Za slanje tokena, tj. njegovo ubacivanje u zaglavlje zahteva se vrši pomoću implementacije servisa *HttpInterceptor*. Kao što sam naziv kaže ovaj servis presreće svaki zahtev koji veb aplikacija šalje. Njegov zadatak je da za svaki zahtev proveri da li postoji token u skladištu i ako postoji da ga ubaci u zaglavlje zahteva pod svojstvom

Authorization i tako prosledi dalje, odnosno pošalje ga. Takođe, presretač se može koristiti i da se ceo zahtev, kao takav, izmeni i tako izmenjen prosledi. U aplikaciji presretač se koristi i da se izmeni putanja ka serverskoj aplikaciji i to deo koji se odnosi na adresu serverske aplikacije. Primer 45 prikazuje kako se koristi lokalno skladište pretraživača kako bi se sačuvao, prihvatio ili obrisao token iz skladišta a primer 46 prikazuje servis koji presreće zahteve i uz njih šalje odgovarajući token ukoliko on postoji.

```
import { Injectable } from '@angular/core';
interface TokenOdgovor {token: string;}

@Injectable() export class TokenizacijaService {
  private token: string;

  constructor() { }

  public sacuvajTokenUSkladistu(token: TokenOdgovor): void {
        if (token.token) {
            localStorage.setItem('travelerkotoken', token.token);
            this.token = token.token; }
  }

  public preuzmiTokenIzSkladista(): string {
    if (!this.token) {
        this.token = localStorage.getItem('travelerkotoken');
    }
    return this.token;
  }

  public obrisiTokenIzSkladista(): void {
    this.token = '';
    window.localStorage.removeItem('travelerkotoken');
  }
}
```

Primer 45: *Primer servisa koji upravlja tokenima u lokalnom skladištu pretraživača*

```
import { Injectable } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor }
   from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import {environment} from '../../environments/environment';

@Injectable()
export class HeaderApiInterceptor implements HttpInterceptor {

    intercept(request: HttpRequest<any>, next: HttpHandler):
        Observable<HttpEvent<any>> {
         const token = localStorage.getItem('travelerkotoken');
         const API_URL = environment.api_url ? environment.api_url :
              '';
         const url = API_URL + request.url;
         if (token) {
              request = request.clone({setHeaders: {Authorization: '
                  Bearer ' + token } });
         }
         request = request.clone({ url: url });
         return next.handle(request);
    }
}
```

Primer 46: *Presretač zahteva koji zahteve usmerava na odgovarajući server i po potrebi ubacuje tokene*

Servisi u aplikaciji su organizovani tako da svaki modul ima najmanje jedan servis koji ima implementirane sve funkcionalnosti koje modul obavlja. Njihov zadatak je uglavnom da šalju zahteve ka serverskoj aplikaciji i od iste primaju odgovore koji se kasnije prosleđuju komponenti koja taj servis koristi. Servisi se dostavljaju (*eng. provide*) komponenti pomoću kontruktora komponetne gde se rezerviše i promenljiva pomoću koje se kasnije pozivaju funkcionalnosti servisa. Na primer, dostavljanje servisa *profil.servis* se dostavlja komponenti na način prikazan u primeru 47.

```
import {ProfilService} from '../../servisi/profil.service';
...
@Component({
  selector: 'app-profil',
  templateUrl: './profil.component.html',
  styleUrls: ['./profil.component.css']
})
export class ProfilComponent implements OnInit {
    constructor(private profil_servis: ProfilService) {
    }
    ...
}
```

Primer 47: *Dostavljanje servisa komponenti*

4.5.2 Mobilna aplikacija

Pravljenje mobilne aplikacije je samo prezentativog karaktera i služi kako bi se prezentovalo kako ista serverska aplikacija može prihvatati zahteve sa dve različite klijentske aplikacije. Kao platforma za mobilnu aplikaciju odabran je *NativeScript* radni okvir kako bi se komponente napisane u *Angular* radnom okviru za veb aplikaciju iskoristile i za mobilnu aplikaciju, čime bi se smanjilo vreme programiranja jer sa *NativeSript* radnim okvirom, najviše razlika između klijentske i mobilne aplikacije ima u šablonima komponenti. Za potrebe ovog rada implementira je početna strana i prijava i to samo za telefone sa *Android* operativnim sistemom. Dodatni modul koji se koristi u mobilnoj aplikaciji je *nativescript-google-maps-sdk* koji služi da poveže veb aplikaciju sa Gugl servisom koji pruža podršku za mape. Programiranje mobilne aplikacije je jako slično programiranju veb aplikacije tako da je deo veb aplikacije prepisan u mobilnu gde su dalje u mobilnoj aplikaciji napravljenje manje izmene koje se pre svega odnose na izgled stranice. U mobilnoj aplikaciji se mogu uvesti svi moduli i biblioteke modula koje se koriste u osnovnoj *Angular* biblioteci odnosno biblioteke čiji naziv počinje niskom *@angular*, tako da se za implementaciju modula, komponenti, servisa i svih ostalih delova mogu koristiti iste biblioteke što znači da se one prave na isti način kao i u veb aplikaciji. Pored *Angular* biblioteka postoje i *NativeScript* biblioteke koje se mogu korisititii koje su prilagođenije *NativeScript* radnom okviru. Česta je upotreba biblioteka koje sadže komponente koje utiču na izgled stranice. Pored drugačije izgleda stranice, još jedna razlika se nalazi u servisu koji rukuje tokenima. S obzirom da mobilna aplikacija ne podržava opciju lokalnog skladišta, u te svrhe se koristi biblioteka *application-settings* i tokeni se čuvaju kao podešavanje aplikacije, što se može videti

u primeru 48.

```
import { Injectable } from '@angular/core';
import * as appset from 'application-settings';

interface TokenOdgovor {
  token: string;
}
@Injectable()
export class TokenizacijaService {

  private token: string;
  constructor() {}

  public sacuvajTokenUSkladistu(token: TokenOdgovor): void {
    if (token.token) {
      appset.setString('travelerkotoken', token.token);
      this.token = token.token;
    }

  }

  public preuzmiTokenIzSkladista(): string {
    if (!this.token) {
      this.token = appset.getString('travelerkotoken');
    }
    return this.token;
  }

  public obrisiTokenIzSkladista(): void {
    this.token = '';
    appset.remove('travelerkotoken');

  }
}
```

Primer 48: *Tokenizacija servis za Android mobilnu aplikaciju*

5 Zaključak

MEAN stek se pokazao kao jako praktičan skup tehnologija za pravljenje veb aplikacija. Uz dobro projektovanje, serverski deo aplikacije se postavlja jako brzo a može se iskoristiti za više različitih klijentskih aplikacija. Greške koje nastaju prilikom pravljenja aplikacije u *Node.js-u* u početku su uglavnom vezane za organizaciju koda tako da podržava asinhroni način rada. Za *Angular* je u početku potrebno malo više vremena za razumevanje jer se sastoji iz više delova a i piše se u *TypeScriptu*, ali kada se on savlada veoma brzo se može savladati i *NativeScript* koji je korišćen za implementaciju mobilne aplikacije. Zapravo, kombinacija *Angulara* i *NativeScripta* štedi vreme programeru jer se *Angular* kod reciklira u *NativeScriptu*. *MongoDB* takođe nije težak za korišćenje a pored standardne literature, *MongoDB* univerzitet omogućava besplatne kurseve koji su posvećeni kako administraciji baze podataka tako i radu sa njom. Prilikom korišćenja baze podataka *MongoDB* najveći izazov je dobro projektovanje baze. Ako se baza projektuje kako treba, onda je i korišćenje baze jednostavno, a ukoliko je baza loše projektovana to će se odraziti i na performanse cele aplikacije jer se onda teže dolazi do željenih podataka.

Iako se *MEAN* stek pokazao kao izuzetno brz, lak i jednostavan stek za pravljenje veb i mobilnih aplikacija, tržište je i dalje orijentisano ka standarnom LAMP pristupu što dovodi do problema nalaženja jeftinih rešenja za hostovanje aplikacije. Pored problema sa hostovanjem, drugi problem je svakako to što je stek relativno mlad, pa je količina kvalitetne literature jako ograničena, a na srpskom jeziku je do sada prevedena samo jedna knjiga. Najveću podršku prilikom razvoja predstavljaju zvanične veb stranice tehnologija na kojima se nalazi dokumentacija a i forumi gde se o nekoj nedoumici može diskutovati sa drugim poznavaocima tehnologije.

Prethodno je rečeno da je prednost tehnologija *MEAN* steka u tome što sve tehnologije u osnovi rade na *JavaScriptu* pa zato programer ne mora da uči više različitih programskih jezika. Međutim, realnost je da bez obzira što je u osnovi sintaksa ista, opet je potrebno usavršiti četiri različite tehnologije. Tako da ako se posmatra sa vremenske strane prilikom učenja, ništa manje vremena se ne ulaže u učenje *MEAN* steka u odnosu na *LAMP* stek. Velika ušteda na vremenu se primećuje tek kasnije, prilikom korišćenja.

6 Literatura i korisni sajtovi

[1] *Angular.io v.6.0.7 dokumentacija.* URL: https://angular.io/docs.

[2] *Besplatni MongoDB kursevi.* URL: https://university.mongodb.com/.

[3] Valentin Bojinov. *RESTful Web API Design with Node.js 10 - Third Edition.* Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Packt Publishing Ltd., 2018.

[4] Brad Dayley, Brendan Dayley, and Caleb Dayley. *Node.js, MongoDB i Angular (integrisane alatke za razvoj veb strana - prevod drugog izdanja).* Obalskih radnika 4a, Beograd: Kompjuter biblioteka, 2018.

[5] Bruno Joseph Dmello. *What you need to know about Node.js.* Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Packt Publishing Ltd., 2016.

[6] Amos Q. Haviv. *MEAN Web Development - Second Edition.* Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Packt Publishing Ltd., 2016.

[7] *MongoDB Atlas.* URL: https://www.mongodb.com/cloud/atlas.

[8] *MongoDB v.4.0 dokumentacija.* URL: https://docs.mongodb.com/manual/.

[9] *Mongoose v.5.1.12 dokumentacija.* URL: https://mongoosejs.com/docs/guide.html.

[10] *NativeScript Angular uputstvo za integraciju.* URL: https://docs.nativescript.org/angular/start/introduction.

[11] Mathieu Nayrolles, Rajesh Gunasundaram, and Sridhar Rao. *Expert Angular.* Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Packt Publishing Ltd., 2017.

[12] *Node.js registar modula.* URL: https://www.npmjs.com/.

7 Dodatak

Za instalaciju cele Travelerko aplikacije potrebno je instalirati *Node.js*, *Angular.io* i *NativeScript*. Za instalaciju svih platformi najbolje je pratiti uputstva sa zvaničnih veb sajtova. Kod se može klonirati sa adrese `https://bitbucket.org/miri91/master_projekat.git` uz pomoć alata *git* ili se može preuzeti iz direktorijuma *master_projekat* koji se nalazi kao dodatak uz elektronsku verziju rada. U direktorijumu projekta se nalaze tri poddirektorijuma:

- *server* - sadrži implementiran *Node.js* REST server
- *veb* - sadrži implementiranu klijentsku veb aplikaciju korišćenjem *Angular* radnog okvira i *Angular CLI* alata.
- *mobilna* - sadrži implementiranu mobilnu aplikaciju za *Android OS*.

Potrebno je u svakom poddirektorijumu pokrenuti komandu `npm install` kako bi se instalirali svi moduli koje aplikacija koristi. Server se iz komandne linije pokreće tako što se u direktorijumu *server* pokrene komanda `npm start`. Veb aplikacija se pokreće tako što se u direktorijumu *veb* pokreće komanda `ng -serve`. Kada se projekat "izgradi" aplikaciji se može pristupiti putem veb pregledača na adresi `http://localhost:4200/`. Veb aplikaciji se takođe može pristupiti na adresi `http://46.101.114.254`. Za mobilnu aplikaciju je neophodno imati ili emulator ili uređaj sa *Android* operativnim sistemom na kome je omogućena opcija *USB debugging* koja se nalazi u opcijama za podešavanje uređaja. Kada se to obezbedi, mobilna aplikacija se pokreće tako što se u direktorijumu *mobilna* pokrene komanda `tns run android`. Za potrebe mobilne aplikacije, serverska je prebačena na javni server koji se nalazi na adresi `http://104.248.26.95`. *MongoDB* nije potrebno instalirati jer se on nalazi na *MongoDB Atlas* platformi.