|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ** |  |

Ивана Савин

**ВЕБ АПЛИКАЦИЈА ЗА ВОЂЕЊЕ БЕЛЕШКИ**

ДИПЛОМСКИ РАД

- Основне академске студије -

Нови Сад, 2018

|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**  21000 НОВИ САД, Трг Доситеја Обрадовића 6 | Датум: |
|  |
| **ЗАДАТАК ЗА ИЗРАДУ ДИПЛОМСКОГ (BACHELOR) РАДА** | Лист/Листова: |
|  |

*(Податке уноси предметни наставник - ментор)*

| Врста студија: | Основне академске студије  Основне струковне студије |
| --- | --- |
| Студијски програм: | **Софтверско инжењерство и информационе технологије** |
| Руководилац студијског програма: | **Проф. др Бранко Милосављевић** |

|  |  |  |  |
| --- | --- | --- | --- |
| Студент: | **Ивана Савин** | Број индекса: | **SW17/13** |
| Област: | **Веб програмирање** | | |
| Ментор: | **Проф. др Милан Видаковић** | | |
| НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ  И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА  ДИПЛОМСКИ (Bachelor) РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:   * проблем – тема рада; * начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна; * литература | | | |

**НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:**

|  |
| --- |
| **Веб апликација за вођење белешки** |

**ТЕКСТ ЗАДАТКА:**

|  |
| --- |
| Задатак рада представља развој једноставне веб апликације за управљање белешкама инспирисану већ постојећим решењем (Google Keep). Серверски део апликације ће бити реализован у програмском језику Јава, коришћењем Spring окружења а клијентски део у Angular 6 окружењу. Апликација омогућава креирање белешки, размештање по екрану и њихово ажурирање. Спецификација ће бити представљена UML дијаграмима. |

|  |  |
| --- | --- |
| Руководилац студијског програма: | Ментор рада: |
|  |  |

|  |
| --- |
| Примерак за:   Студента;   Ментора |

Образац **Q2.НА.15-04** - Издање 1

SADRŽAJ

[1. Uvod 3](#__RefHeading___Toc10932_4196831394)

[2. Opis korišćenih tehnologija 5](#__RefHeading___Toc10934_4196831394)

[2.1 Spring okruženje 5](#__RefHeading___Toc10936_4196831394)

[2.1.1 Spring MVC 6](#__RefHeading___Toc10938_4196831394)

[2.1.2 Spring Data 8](#__RefHeading___Toc10940_4196831394)

[2.2 Angular okruženje 11](#__RefHeading___Toc10942_4196831394)

[2.2.1 Komponenta 11](#__RefHeading___Toc10944_4196831394)

[2.2.2 Rutiranje 13](#__RefHeading___Toc10946_4196831394)

[2.2.3 Servisi 14](#__RefHeading___Toc10948_4196831394)

[2.3 Gridster biblioteka 15](#__RefHeading___Toc10950_4196831394)

[2.3.1 GridsterComponent 17](#__RefHeading___Toc10952_4196831394)

[2.3.2 GridsterItemComponent 19](#__RefHeading___Toc10954_4196831394)

[3. Specifikacija aplikacije 21](#__RefHeading___Toc10956_4196831394)

[3.1 Dijagram slučajeva korišćenja 22](#__RefHeading___Toc10958_4196831394)

[3.2 Dijagram klasa 23](#__RefHeading___Toc10960_4196831394)

[3.3 Dijagram šeme baze podataka 24](#__RefHeading___Toc10962_4196831394)

[4. Opis implementacije 25](#__RefHeading___Toc10964_4196831394)

[4.1 Serverski deo 25](#__RefHeading___Toc10966_4196831394)

[4.1.1 Model 26](#__RefHeading___Toc10968_4196831394)

[4.1.2 Repository 32](#__RefHeading___Toc10970_4196831394)

[4.1.3 Service 33](#__RefHeading___Toc10972_4196831394)

[4.1.4 Data Transfer Object 42](#__RefHeading___Toc10974_4196831394)

[4.1.5 Controller 43](#__RefHeading___Toc10976_4196831394)

[4.2 Klijentski deo 46](#__RefHeading___Toc10978_4196831394)

[5. Zaključak 63](#__RefHeading___Toc10980_4196831394)

[6.KLJUČNA DOKUMENTACIJSKA INFORMACIJA 65](#__RefHeading___Toc10982_4196831394)

[7.KEY WORDS DOCUMENTATION 67](#__RefHeading___Toc10984_4196831394)

# Uvod

Zadatak rada predstavlja razvoj jednostavne aplikacije za upravljanje beleškama inspirisanu već postojećim rešenjem (Google Keep) [1].

Budući da je zadatak razvoj veb aplikacije, bilo je potrebno razviti klijentski i serverski deo, kao i izabrati pogodne tehnologije za implemetaciju.

Najveći izazov je bio pronalazak ogovarajuće biblioteke za razvoj klijentskog dela koja bi pružila ponašanje koje nudi originalna aplikacija.

Prvo poglavlje je uvodno, dok se u drugom nalazi opis izabranih tehnologija koji su korišćeni pri izradi. Treće poglavlje predstavlja detaljnu specifikaciju aplikacije kroz UML dijagrame. Četvrto poglavlje predstavlja opis konkretne implementacije veb aplikacije, dok je u petom poglavlju iznet zaključak.

# Opis korišćenih tehnologija

Serverski deo aplikacije je implementiran u programskom jeziku Java [2], verzije 1.8, korišćenjem Spring okruženja [3]. Radi pojednostavljenog korišćenja Spring okruženja korišćen je Spring Boot [4] razvojni okvir kao i Spring Data [5] za lakše upravljanje podacima.

Za implementaciju klijentskog dela aplikacije korišćeno je Angular 6 okruženje [6] uz dodatak Gridster biblioteke [7]. Kako bi se razvoj klijentskog dela pojednostavio korišćen je Angular CLI (*Command Line Interface*) [8].

U daljem tekstu će se zbog jednostavnosti za Angular 6 okruženje navoditi samo Angular.

## Spring okruženje

Spring je okurženje za jednostavniji razvoj poslovnih aplikacija u Javi. Trenutno je najpopularniji i najuticajnije okruženje za razvoj Java veb aplikacija. Omogućuje kreiranje J2EE aplikacija na jednostavniji način. Sadrži infrastrukturu za izgradnju Java aplikacija, upravlja infrastrukturom tako da se standardne funkcionalnosti realizuju brzo i lako.

Glavni koncepti ovog okruženja su povezivanje objektata kroz *dependecy injection* mehanizam [9] i deklarativno programiranje korišćenjem aspekata.

Prvi koncept omogućava da se funkcionalnosti realizuju zajedničkim radom međusobno povezanih komponenti, a drugi koncept omogućava izdvajanje često korišćenih funkcionalnosti u aspekte i njihovo ponovno iskorišćenje.

### Spring MVC

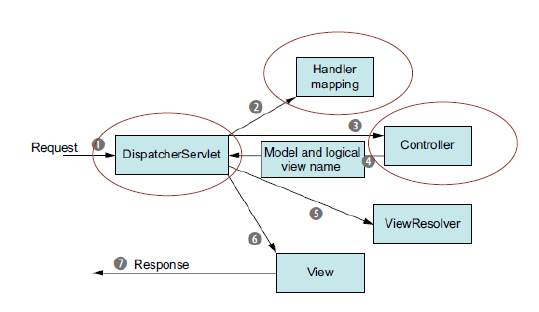
REST (*Representational State Transfer*) [10] je stil softverske arhitekture koji se zasniva na postojanju resursa i uniformnom upravljanju njima putem skupa predefinisanih operacija. Najčešće se koristi za upravljanje resursima koje obezbeđuje serverski API (*Application programming interface*) aplikacije.

Resurs predstavljaju podaci ili funkcionalnosti identifikovani jedinstvenim identifikatorom (URI – *Uniform Resource Identifier*). Resurs je odvojen od konkretnog formata reprezentacije, tj. isti resurs može biti predstavljen različitim formatima. Za upravljanje resursima se koriste predefinisane operacije: *create*, *read*, *update* i *delete*.

HTTP (*Hyper Text Transfer Protocol*) [11] je najčešći protokol za komunikaciju, a koriste se njegove metode POST, GET, PUT, DELETE.

Spring podrška za REST se oslanja na Spring MVC [12]. Spring MVC je Spring podrška za veb aplikacije, razdvaja veb aplikaciju u *model*, *view* i *controller* sloj. Ovo ne odgovara savremenim pristupima koji tehnološki odvajaju klijenta i server i koriste na klijentu JavaScript [13] bazirane radne okvire. Ipak *controller* sloj se može koristiti i ako je *view* realizovan na klijentu.

Za realizaciju REST servisa koristi se sloj za obradu zahteva. Na slici 1.1 je prikazana Spring MVC odbrada zahteva.

Slika 1.1 Spring MVC obrada zahteva

*DispatcherServlet* predstavlja *front controller* u Spring veb aplikaciji. *Front controller* je softverski obrazac koji obezbeđuje centralizovanu ulaznu tačku za obradu veb zahteva. On analizira zahtev i prosleđuje ga odgovarajućem delu aplikacije, tj. odgovarajućem kontroleru koji vrši obradu zahteva.

*Handler mapping* je komponenta zadužena za analizu URL-a zahteva i na osnovu toga se zaključuje koji kontroler treba da obradi zahtev.

*Controller* je obična Java klasa, anotirana sa @Controller ili @RestController. Metode ove klase su anotirane sa @RequestMapping. Ova anotacija opisuje zahtev koji treba biti obrađenu toj metodi kontrolera. Anotacija se može postaviti i na nivou klase. Tako definisan URL (*Uniform Resource Locator*) je prefiks za svaku metodu klase a metode se onda razlikuju po tipu HTTP metode koju obrađuju.

Zahtev može sadržati parametre i oni mogu biti prosleđeni kontroleru kao parametar koji je promenjljiva u URL-u zahteva ili kao parametar HTTP zahteva.

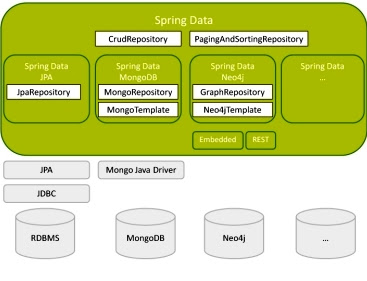
Kontroler radi sa Java objektima i ne bavi se formatom u kojem će resurs biti isporučen klijentu. Spring podržava dva načina za transformisanje REST resursa iz Java objekata u format koji klijent očekuje. Prvi pristup je *content negotiation* koji koristi klasični Spring MVC mehanizam za kreiranje *view* objekta koji se šalje klijentu. Drugi pristup je konverzija poruke gde poseban konverter samo pretvara Java objakat u ciljni format bez prolaska kroz kompletan Spring MVC tok. Ovaj pristup od celog Spring MVC koristi samo *controller*. *DispatcherServlet* ne koristi *model* i *view* komponentu. Podatke koje vrati *controller*, *DispatcherServlet* konvertuje i šalje preko mreže. Konvertor podataka se automatski primenjuje na osnovu formata resursa koji je metoda naznačila da vraća.

Spring omogućuje i drugi smer konverzije. Konverzija podataka primljenih sa klijenta se vrši tako što konvertor konvertuje podatke primljene u telu HTTP zahteva u Java objekat. Potrebno je očekivani Java objekat definisati kao parametar REST metode i anotirati parametar @RequestBody anotacijom. Zavisno od tipa podatka koje metoda očekuje primenjuje se odgovarajući konverter. Metode kontrolera mogu da vrate *ResponseEntity* objekat koji sadrži više informacija o odgovoru, ne samo podatke nego i metapodatke.

### Spring Data

Spring Data je poseban projekat napravljen da bi se olakšalo upravljanje podacima. Pojednostavljuje klasične Spring tehnike upravljanja podacima, kraći kod kroz ugrađenu podršku za standardne funkcionalnosti, jednostavnije je vršenje upita ka bazi podataka.

Spring Data JPA (*Java Persitence* *API*) uvodi koncept repozitorijuma koji izbacije potrebu za pisanjem ponavljajućeg koda. Repozitorijum je interfejs, mogu postojati različiti tipovi, svi izvedeni iz pretka *Repository*. Kreiranjem interfejsa koji nasleđuje neki od repozitorijumskih interfejsa obezbeđuje se upravljanje podacima. Nije potrebno praviti klasu koja implementira interfejs, samim postojanjem interfejsa je moguće koristiti operacije predviđene nasleđenim repozitorijumskim interfejsom.

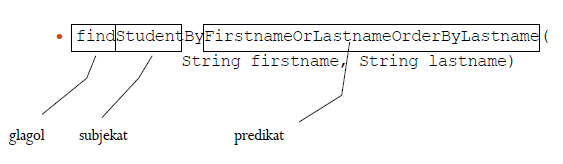


Slika 1.2 Spring Data

Kao što se može videti na slici 1.2 neki od repozitorijumskih interfejsa su:

* *CrudRepository* – specijalizacija generičkog repozitorijuma. Sadrzi standardne CRUD (*create, update, delete*) operacije nad entitetom,
* *PagingAndSortingRepository* – specijalizacija CRUD repozitorijuma, omogućuje paginaciju i sortiranje i
* *JPARepository* – specijalizacija *PagingAndSorting* interfejsa. Sadrži dodatnu podršku za JPA. Ovaj interfejs pruža sve najčešće operacije potrebne za rad sa podacima u standardnom veb informacionom sistemu.

Ukoliko postoji potreba za nestandardnim operacijama one se u kreirani interfejs mogu dodati deklaracijom *query* metode. Ideja je da se poštovanjem konvencije u imenovanju metode metoda samo deklariše. Na osnovu deklaracije koja poštuje specificiranu formu, Spring automatski obezbeđuje implemetaciju. Na slici 1.3 se može videti struktura *query* metode.

Slika 1.3 Struktura *query* metode

Ukoliko se nestandardni upiti ne mogu izraziti *query* metodom, moguće je definisati proizvoljan upit. Upit se zadaje u JPQL (*Java Persistence Query Language*) jeziku koji je deo JPA specifikacije. I dalje nema potrebe za implementacijom metode, samo je potrebno da se anotira željenim upitom. Na slici 1.4 se može videti primer upotrebe JPQL upita.

@Query("SELECT DISTINCT widget FROM Widget widget WHERE ( widget.title LIKE CONCAT ('%', :search, '%') OR

widget.content LIKE CONCAT ('%', :search, '%') OR

widget.date LIKE CONCAT ('%', :search, '%'))")

List<Widget> findBySearch(@Param("search") String search);

Slika 1.4 Primer JPQL upita

JPA Repository može biti bilo koja implementacija JPA. U ovom slučaju je to Hibernate, dok je za konkretan *Data Source* korišćena MySQL baza podataka.

*Hibernate* je okruženje koje pruža mogućnost mapiranja objektno orijentisanog modela na relacionu bazu podataka. Primarna uloga *Hibernate*-a je mapiranje javnih klasa na tabele u bazi. Hibernate omogućava razvoj perzistentnog modela korišćenjem idioma objektno-orijentisanog programiranja. Ne zahteva nikakve posebne interfejse ili klase i time omogućava da bilo koja klasa ili struktura podataka bude perzistentna.

## Angular okruženje

Angular je okruženje za razvoj klijentskih aplikacija. Osnovni gradivni blok aplikacije su komponente. Pomoću Angulara razvijamo SPA (*Single Page Application*). Budući da su aplikacije najčešće prevelike da bi se prikazivale u jednom prikazu sa jednim kontrolerom one se dele na komponente, gde je svaka komponenta zadužena za deo funkcionalnosti. Komponente su organizovane u stablo, tako da se aplikacija sastoji od stabla komponenti. Prelazak između komponenti se postiže rutiranjem. Na slici 1.5 se može videti arhitektura Angular aplikacije.

### 

Slika 1.5 Arhitektura Angular aplikacije

### Komponenta

Kao što je rečeno, komponenta je gradivna jedinica Angular aplikacije koja je zadužena za određeni deo funkcionalnosti. Ona objedinjuje element prikaza (HTML *template*) i aplikativnu logiku (TS klasu) koja ga kontroliše. Aplikativna logika komunicira sa prikazom kroz javni API klase.

#### Anatomija komponente

TS klasa je dekorisana dekoraterom @Component i sadrži:

* *selector* – naziv elementa koji će se koristiti u stranici,
* *templateUrl* – url HTML templejta i
* *styleUrls* – stilovi koji se primenjuju samo na ovu komponentu.

#### Registrovanje komponente

Kada je komponenta kreirana potrebno je registrovati je u aplikaciji. Registrovanje komponente se obavlja dodavanjem komponente u deklaraciju ngModule-a, može se videti na slici 1.6. NgModule predstavlja korenski modul aplikacije i svaka aplikacija ima bar jedan korenski modul.

@NgModule({

declarations: [

AppComponent,

LoginComponent,

RegisterComponent,

HomeComponent,

DashboardComponent,

WidgetUpdateComponent,

DashboardArchiveComponent,

DashboardTrashComponent,

ChangePasswordComponent,

UserProfileComponent,

UserProfileEditComponent

]

})

Slika 1.6 Primer registrovanja komponenti

### Rutiranje

Budući da Angular aplikaciju čini stablo komponenti a da svaka komponenta ima svoj prikaz potrebno je imati navigaciju među prikazima. S obzirom da je aplikacija stablo komponenti, rutiranje se svodi na zadavanje komponenti za rute. Samo rutiranje se postiže pravilima rutiranja što se može videti na slici 1.7.

const routes: Routes = [

{ path: "", component: LoginComponent },

{ path: "register", component: RegisterComponent },

{ path: "login", component: LoginComponent },

{ path: "change-password",

component: ChangePasswordComponent },

{ path: "home", component: HomeComponent,

children: [

{ path: "dashboard", component: DashboardComponent },

{ path: "archive", component: DashboardArchiveComponent },

{ path: "trash", component: DashboardTrashComponent },

{ path: "profile", component: UserProfileComponent },

{ path: "edit-profile",

component: UserProfileEditComponent }]

}

]

Slika 1.7 Zadavanje ruta

Pored samih navigacija među različitim prikazima nekada je potrebno da se između komponenti ruta prenose vrednosti. Jednostavne vrednosti se mogu preneti tako što se ugrade u rutu a može im se pristupiti u kontroleru kroz Route servis.

### Servisi

Bitan deo Angular aplikacije čine servisi. Servisi treba da omoguće pristup podacima i stoga predstavljaju sponu između klijenta i servera. Klijent i server komuniciraju putem HTTP protokola pa je potrebno obezbediti klijentu sigurnu komunikaciju. Za HTTP komunikaciju u Angular okruženju je preporučeno da se koristi HttpClient biblioteka. Ova biblioteka sadrži metode koje odgovaraju HTTP metodama.

Zahtevi se mogu parametrizovati ili se objekti mogu slati kao objekti tipa HttpParams ili tipa HttpHeaders. Servisi su anotrani sa @Injectable, što znači da se sa komponentom povezuju putem *dependency injection*-a i da komponenta može pozivati njihove metode kada joj zatrebaju. Na slici 1.8 se može videti jedna implementacija servisa.

@Injectable()

export class DashboardService {

private baseUrl = 'http://localhost:8080/api/dashboards';

constructor(private http: HttpClient) { }

getDashboard(id : number): Observable<Dashboard> {

return this.http.get<Dashboard>(`${this.baseUrl}/${id}`);

}

update(id:number, dashboard:Dashboard): Observable<Dashboard> {

return this.http.put<Dashboard>(`${this.baseUrl}/${id}`, dashboard);

}

changeDashboard(id : number, dashboard): Observable<Dashboard> {

return

this.http.put<Dashboard>(`${this.baseUrl}/${id}/change`, dashboard);

}

search(id : number, search: string): Observable<Dashboard> {

let params = new HttpParams().set('search', search);

return

this.http.get<Dashboard>(`${this.baseUrl}/$ {id}/search`, { params: params });

}

delete(id : number): Observable<Dashboard> {

return this.http.delete<Dashboard>(`${this.baseUrl}/${id}`);

}

}

Slika 1.8 Implementacija servisa

## Gridster biblioteka

Kada je razvoj klijentskog dela aplikacije u pitanju, birane su biblioteke koje bi korisniku pružile upravljanje beleškama što sličnije onom koje nudi originalna aplikacija. Kao najadekvatnije rešenje za implementaciju upravljanja beleškama izabrana je angular2gridster biblioteka. Detalje vezane za implementaciju ove biblioteke možete pronaći na github stranici <https://github.com/swiety85/angular2gridster>.

Biblioteka je jednostavna za korišćenje a poseduje i detaljnu dokumentaciju. Zavisno od verzije Angular-a potrebno je instalirati odgovarajuću verziju biblioteke. Za Angular 6 je potrebno instalirati poslednju verziju, tj. verziju 6.x. Biblioteka se može instalirati preko npm-a pokretanjem sledeće komande (slika 1.9):

npm install angular2gridster

Slika 1.9 Instaliranje Gridster biblioteke

Nakon što se instalira, biblioteku je potrebno importovati u projekat i nakon toga se može koristiti. Na slici 1.10 se može videti kako se importuje biblioteka.

...

import { GridsterModule } from 'angular2gridster';

@NgModule({

declarations: [

AppComponent

],

imports: [

...

GridsterModule.forRoot()

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Slika 1.10 Importovanje Gridster biblioteke.

Ova biblioteka omogućava upravljanje tablama sa komponenta, a neke od podržanih operacija su:

* dodavanje novih komponenti,
* dinamičko dodeljivanje najadekvatnije pozicije komponenti,
* prevlačenje komponenti (*drag-and-drop*),
* promena lokacije komponenti na tabli,
* pomena veličine komponenti u svim pravcima (*resize*),
* onemogućeno prevlačenje za selektovane komponente (*disabled drag-and-drop*),
* onemogućena promena veličine za selektovane komponente (*disabled resizing*) i
* odgovarajući pregled zavisno od veličine ekrana (*responsive view*).

Glavne komponente ove biblioteke su:

* GridsterComponent – predstavlja tablu sa komponentama
* GridsterItemComponent – predstavlja komponentu sa table i
* GridsterPrototypeDirective – predstavlja direktivu za dodavanje novih komponenti na tablu povlačenjem predefinisane komponente.

### GridsterComponent

Da bi smo dobili željeno rukovanje tablom i njenim komponentima potrebno je definisati *options*, tj. podešavanja vezna za tablu. Options je tipa *IGridsterOptions*, a na slici 1.11 se može videti detaljno podešavanje.

gridsterOptions: IGridsterOptions = {

lanes: 2, // amount of lanes (cells) in the grid

direction: 'vertical', // floating top - vertical, left - horizontal

floating: true,

dragAndDrop: true, // enable/disable drag and drop for all items in grid

resizable: true, // enable/disable resizing by drag and drop for all items in grid

// Defines in which direction resize is allowed.

resizeHandles: {

s: true,

e: true,

se: true

},

widthHeightRatio: 1, // proportion between item width and height

lines: {

visible: true,

color: '#afafaf',

width: 2

},

shrink: true,

useCSSTransforms: true,

responsiveView: true, // turn on adopting items sizes on window resize and enable responsiveOptions

responsiveDebounce: 500, // window resize debounce time

responsiveSizes: true,

responsiveOptions: [

{ breakpoint: 'sm', lanes: 4 },

{ breakpoint: 'md', minWidth: 768, lanes: 5 },

{ breakpoint: 'lg', minWidth: 1250, lanes: 6 },

{ breakpoint: 'xl', minWidth: 1800, lanes: 8 }

]

};

Slika 1.11 Podešavanje GridsterComponent komponente

Na nivou table postoje i nekoliko funkcija:

* *optionsChange* – funkcija koja se pozove nakon što je neko od podešavanja izmenjeno,
* *reflow* – funkcija koja se pozove svaki put kada se preračunava raspored između komponenti,
* *setOption*(*optionName*, *value*) – funkcija koja se može pozvati nad objektom tipa *GridsterComponent* i tako postaviti nova podešavanja za tu komponentu i
* *reload* – metoda koja može pozvati nad objektom tipa *GridsterComponent* i koja preračunava pozicije i veličine svih komponenti i primenjuje ih na tabli.

Neke od funkcija čija je realizacija u planu su:

* prevlačenje komponenti između različitih tabli,
* dinamičko dodeljivanje veličine komponeti na osnovu sadržaja i
* algoritam raspoređivanja komponenti (*layout packing algorithm*).

### GridsterItemComponent

Ova komponenta predstavlja jedan element table. Ima nekoliko atributa, pomoću kojih se može promeniti njeno ponašanje.

Neki od atributa su:

* x – horizontalna pozicija (računa se u broju ćelija),
* y – vertikalna pozicija (računa se u broju ćelija),
* w – širina komponente (računa se u broju ćelija),
* h – visina komponente (računa se u broju ćelija),
* dragAndDrop – ukoliko je vrednost false sprečava premeštanje komponente, u suprotnom premeštanje je moguće,
* resizable – ukoliko je vrednost false sprečava promenu veličine komponente, u suprotnom je promena moguća i
* options – podešavanja vezana za komponentu, neka od njih su:
  + minWidth – najmanja širina,
  + minHeight – najmanja visina,
  + maxWidth – najveća širina i
  + maxHeight – najveća visina.

Neke od funkcija koje se pozivavaju nad ovim objektom:

* xChange – pozvana nakon što se promeni pozicija x,
* yChange – pozvana nakon što se promeni pozicija y i
* change – pozvana nakon što se komponenta promeni.

# Specifikacija aplikacije

Zadatak obuhvata izradu veb aplikacije koja predstavlja klon Googe Keep aplikacije (<https://keep.google.com/>). Aplikacija omogućava kreiranje beleški, rukovanje njima i njihovo ažuriranje.

Prilikom dolaska na veb stranicu aplikacije korisniku je omogućena prijava na sistem ukoliko korisnik već ima nalog ili registrovanje u slučaju da nema nalog.

Nakon uspešne prijave na sistem korisniku se prikazuje glavna stranica aplikacije na kojoj korisnik može da vidi svoje beleške kao i da dodaje nove, menja postojeće ili da postojeće prebaci u arhivu. Pored glavne stranice postoje još stranica sa arhiviranim beleškama i stranica sa beleškama za brisanje kao i stranica za pregled i ažuriranje korisnikovih podataka.

Na glavnoj stranici korisnik ima mogućnost da pretražuje beleške po naslovu, sadržaju i datumu kada su kreirane. Pored toga, korisnik može da izmeni raspored po kom su poređane beleške, da im promeni veličinu ili da ih zakači na izabrano mesto. Takođe, može da promeni sadržaj i naslov beleški.

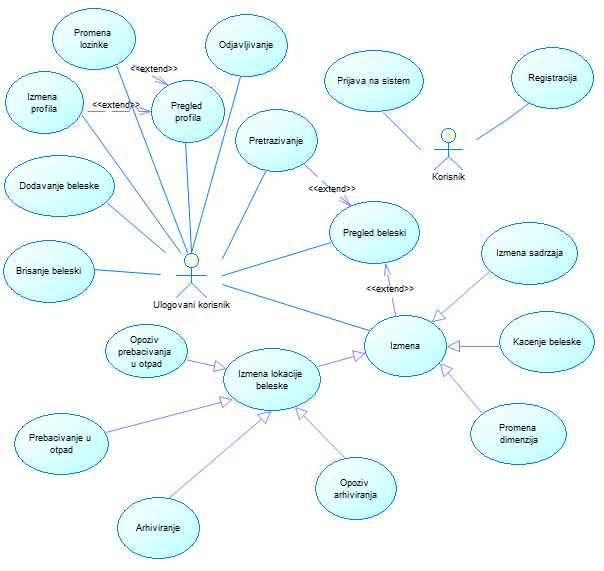
Na stranci sa arhiviranim beleškama korisnik može da pregleda beleške koje je prebacio u arhivu i ukoliko se predomisli da opozove arhiviranje ili da ih prebaci u kantu za brisanje. Nakon što beleška dospe u kantu sa brisanje nakon određenog vremenskog perioda se automatski briše iz sistema.

Pored stranica sa beleškama postoji i stranica na kojoj se prikazuju korisnikovi lični podaci sa koje može ažurirati svoje podatke.

U narednim poglavljima biće prikazani dijagram slučajeva korišćenja, dijagram klasa i šema baze podataka.

## Dijagram slučajeva korišćenja

Dijagram slučajeva korišćenja prikazan je na slici 2.1. Kao što se može videti na dijagramu, aplikaciju mogu koristiti dva tipa korisnika. Prvi, neregistrovani korisnik ima mogućnost da se registruje ili prijavi na sistem. Nakon uspešne prijave, korisnik postaje ulogovani korisnik i ima mogućnost da pregleda svoje beleške, pretražuje ih, dodaje nove, menja postojeće ili da ih briše. Takođe, korisnik ima mogućnost da pregleda svoj profil, ažurira ga i da se odjavi sa sistema.

Slika 2.1 Dijagram slučajeva korišćenja

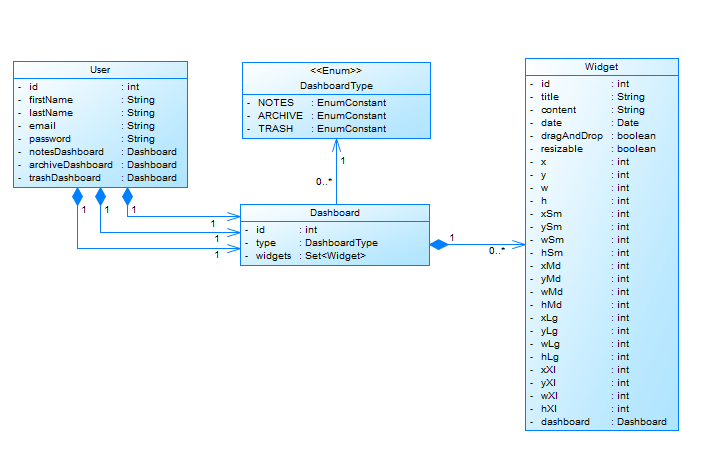
## Dijagram klasa

S obzirom da je domen aplikacije veoma jednostavan i model podataka je veoma jednostavan i sastoji se od 3 klase i jedne enumeracije.

Osnovu modela čini klasa Widget koja predstavlja belešku. Pored osnovnih obeležja *title* (naslov), *content* (sadžaj) i *date* (datum) u modelu se nalaze i obeležja neophodna za precizan prikaz na tabli. Zbog veće preglednosti formirana je klasa Dashboard koja predstavlja tablu sa beleškama kao i enumeracija DashboardType koja služi da raspoznamo o kojoj je tabli reč.

Treća klasa je klasa User koja pored osnovnih podataka o korisniku sadrži reference ka tri table sa beleškama različitog tipa.

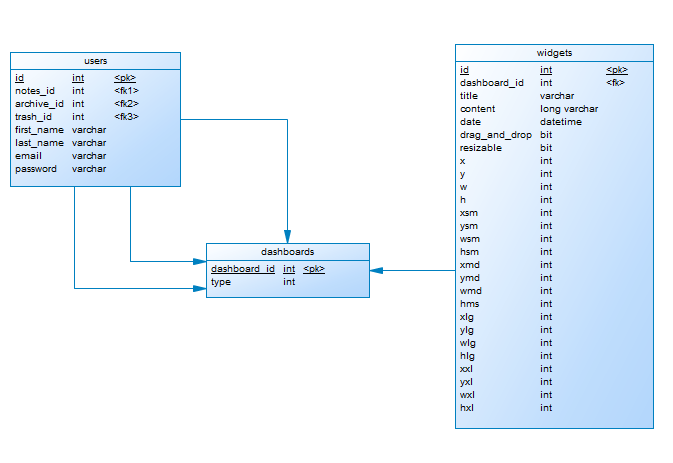
Dijagram klasa prikazan je na slici 2.2.

Slika 2.2 Dijagram klasa

## Dijagram šeme baze podataka

Na slici 3.3 prikazana je šema baze podataka. S obzirom na to da je šema baze podataka generisana na osnovu modela veoma je jednostavna i sastoji se od 3 tabele. Tabele koje se nalaze u bazi podataka su:

* users – tabela koja sadrzi podatke o korisnicima,
* dashboards – tabela koja sadrži podatke o tablama sa beleškama i
* widgets – tabela koja sadrži podatke o beleškama.

****

Slika 2.3 Šema baze

# Opis implementacije

Aplikaciju čine dve celine, serverski i klijentski deo. Serverski deo aplikacije je implementiran u programskom jeziku Java, verzije 1.8 korišćenjem Spring okruženja. Za implementaciju klijentskog dela aplikacije korišćen je Angular okruženje uz dodatak Gridster biblioteke.

Arhitektura serverske aplikacije je zasnovana na Spring MVC pristupu koji razdvaja aplikaciju u *model*, *view* i *controller* sloj. Razlika u odnosu na klasičan Spring MVC pristup je ta da je kompletan *view* razvijen na klijentu.

## Serverski deo

Kao što je već rečeno serverski deo aplikacije je implementran po ugledu na Spring MVC gde se na serverskoj strani nalaze *model* i *controller*.

Ovo znači da je aplikacija horizontalno podeljena i sastoji se iz nekoliko slojeva:

* *Model* – klase koje opisuju modele podataka,
* *Repository* – objekti koji sadrže metode za pristup i komunikaciju sa bazom podataka,
* *Service* – servisni sloj, nosioci biznis logike, validatori i konverteri,
* DTO (*Data Transfer Object*) – objekti koji se prenose preko mreže i
* *Controller* – sloj za komunikaciju sa klijentom.

Model čine tri klase i jedna enumeracija. Klase su User koja predstavlja korisnika aplikacije, Widget koja predstavlja belešku i Dashboard koji predstavlja tablu sa beleškama. Radi lakšeg raspoznavanja uvedena je enumeracije DashboardType koja predstavlja tip table sa beleškama.

Sloj kontrolera čine dva kontrolera, UserController i DashboardController. Kontroleri za komunikaciju sa klijentom ne koriste klase modela već njihove podskupove, DTO klase. Sloj DTO klasa čini nekoliko klasa, zavisno od zahteva koji se dobija od klijenta. Kontroleri koriste servise UserService i DashboardService u kojima se nalazi poslovna logika. Na nivou servisa postoje još dva tipa servisa, jedni su za konverovanje modela u dto objekte i obrnuto dok su u drugi tip izdvojene validacije. Servisi za konverovanje su UserConverter, WidgetConverter i Dashboard Converter, dok servise za validaciju čine UserValidation i DashboardValidation servisi.

Servisni sloj koristi sloj repozitorijuma koji sluzi za komunikaciju sa bazom koji čine UserRepository, WidgetRepository i DashboardRepository.

### Model

Kao što je rečeno, modele čine tri klase i jedna enumeracija. Implementacija modela se može videti na slikama 3.1, 3.2, 3.3 i 3.4. Zbog preglednosti na slikama neće biti prikazane *property* metode kao i *import* sekcija.

@Entity

@Table(name = "users")

**public** **class** User **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 816250384561263812L;

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

@Column(name = "id", nullable = **false** , unique = **true**)

**private** **int** id;

@Column(name = "first\_name")

**private** String firstName;

@Column(name = "last\_name")

**private** String lastName;

@Column(name = "email", nullable = **false**, unique = **true**)

**private** String email;

@Column(name = "password")

**private** String password;

@OneToOne(fetch = FetchType.***EAGER***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "notes\_id")

**private** Dashboard notesDashboard;

@OneToOne(fetch = FetchType.***EAGER***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "archive\_id")

**private** Dashboard archiveDashboard;

@OneToOne(fetch = FetchType.***EAGER***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "trash\_id")

**private** Dashboard trashDashboard;

**public** User() {

**super**();

}

}

Slika 3.1 Implementacija klase User

@Entity

@Table(name = "widgets")

**public** **class** Widget **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 8405975872512623038L;

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

@Column(name = "id", nullable = **false** , unique = **true**)

**private** **int** id;

@Column(name = "title")

**private** String title;

@Lob

@Column(name = "content")

**private** String content;

@CreationTimestamp

@Column(name = "date")

@Temporal(TemporalType.***TIMESTAMP***)

**private** Date date;

@Column(name = "drag\_and\_drop")

**private** **boolean** dragAndDrop;

@Column(name = "resizable")

**private** **boolean** resizable;

@Column(name = "x")

**private** **int** x;

@Column(name = "y")

**private** **int** y;

@Column(name = "w")

**private** **int** w;

@Column(name = "h")

**private** **int** h;

@Column(name = "xsm")

**private** **int** xSm;

@Column(name = "ysm")

**private** **int** ySm;

@Column(name = "wsm")

**private** **int** wSm;

@Column(name = "hsm")

**private** **int** hSm;

@Column(name = "xmd")

**private** **int** xMd;

@Column(name = "ymd")

**private** **int** yMd;

@Column(name = "wmd")

**private** **int** wMd;

@Column(name = "hmd")

**private** **int** hMd;

@Column(name = "xlg")

**private** **int** xLg;

@Column(name = "ylg")

**private** **int** yLg;

@Column(name = "wlg")

**private** **int** wLg;

@Column(name = "hlg")

**private** **int** hLg;

@Column(name = "xxl")

**private** **int** xXl;

@Column(name = "yxl")

**private** **int** yXl;

@Column(name = "wxl")

**private** **int** wXl;

@Column(name = "hxl")

**private** **int** hXl;

@ManyToOne

@JoinColumn(name = "dashboard\_id", referencedColumnName = "id", nullable = **false**)

**private** Dashboard dashboard;

**public** Widget() {

**super**();

}

}

Slika 3.2 Implementacija klase Widget

@Entity

@Table(name = "dashboards")

**public** **class** Dashboard **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = -9056970010173391664L;

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

@Column(name = "id", nullable = **false** , unique = **true**)

**private** **int** id;

@Column(name = "type")

**private** DashboardType dashboardType;

@OneToMany(cascade = CascadeType.***ALL***, fetch = FetchType.***EAGER***, orphanRemoval = **true**, mappedBy = "dashboard")

**private** Set<Widget> widgets;

**public** Dashboard() {

**super**();

}

}

Slika 3.3 Implementacija klase Dashboard

**public** **enum** DashboardType {

***NOTES***, ***ARCHIVE***, ***TRASH***

}

Slika 3.4 Implementacija enumeracije DashboardType

### Repository

Sloj repozitorijuma čine tri interfejsa koji obezbeđuju potrebnu komunikaciju sa bazom. Za konkretnu implementaciju repozitorijuma korišćen je *JPARepository*. Pored standarnih operacija koje se nalaze u ovoj implementaciji bilo je potrebno deklarisati i nestandardne operacije. Većina nestandarnih operacija deklarisano je kao Query metoda dok je jedna kompleksnija operacija deklarisana JPQL jezikom. Više o deklarisanju ovih metoda opisano je u poglavlju 2.12. Iako se koriste metode iz JPARepository klase one nisu posebno deklarisane u ovim implementacijama repozitorijuma. UserRepository i DashboardRepository sadrže po nekoliko nekoliko nestandardnih operacija koje su poštovanjem konvencije deklarisane kao Query metode što se može videti na slikama 3.5 i 3.6.

**public** **interface** UserRepository **extends** JpaRepository<User, Integer> {

User findById(**int** id);

User findByEmail(String email);

User findByEmailAndPassword(String email, String password);

}

Slika 3.5 Implementacija UserRepository interfejsa

**public** **interface** DashboardRepository **extends** JpaRepository<Dashboard, Integer> {

Dashboard findById(**int** id);

Dashboard findByDashboardType(DashboardType dashboardType);

}

Slika 3.6 Implementacija DashboardRepository interfejsa

WidgetRepository kao i prethodna dva sadrži nestandardne metode, od kojih je jedna deklarisana pomoću JPQL jezika. Implementacija ovog interfejsa se može videti na slici 3.7.

**public** **interface** WidgetRepository **extends** JpaRepository<Widget, Integer> {

Widget findById(**int** id);

@Query("SELECT DISTINCT widget FROM Widget widget WHERE ( widget.title LIKE CONCAT ('%', :search, '%') OR

widget.content LIKE CONCAT ('%', :search, '%') OR

widget.date LIKE CONCAT ('%', :search, '%'))")

List<Widget> findBySearch(@Param("search") String search);

}

Slika 3.7 Implementacija WidgetRepository interfejsa

### Service

Servisni sloj čine tri vrste servisa:

* servisi koji predstavljaju poslovnu logiku,
* servisi za validaciju i
* servisi za konvertovanje modela u DTO objekte i obrnuto.

Svaki od ovih servisa je implementiran kao par interfejs – implementacija interfejsa.

#### Servisi koji predstavljaju poslovnu logiku

Servisi koji predstavljaju poslovnu logiku su implementirani oslanjajući se na sloj repozitorijuma koji je opisan u prethodnom poglavlju. Servisi rade sa objektima modela koje prethodno konvertuje servis zadužen za ovaj posao. Nakon izvršene operacije servisi takođe vraćaju objekte modela. Ovaj sloj podataka sadrži par interfejs i implementaciju interfejsa. U interfejsima su deklarisane metode dok se njihove implementacije nalaze u klasama. Na slici 3.8 se vidi primer deklarisanja metoda poslovne logike za DashboardService, a na slici 3.9 se vidi njihova implementacija.

**public** **interface** DashboardService {

**public** Dashboard findById(**int** id);

**public** Dashboard save(Dashboard dashboard);

**public** Dashboard update(**int** id, Dashboard dashboard);

**public** Dashboard changeDashboard(**int** id, Dashboard request);

**public** Dashboard clearDashboard(**int** id);

**public** Dashboard search(**int** id, String search);

}

Slika 3.8 Implementacija DashboardService interfejsa

@Service

**public** **class** DashboardServiceImpl **implements** DashboardService {

@Autowired

DashboardRepository dashboardRepository;

@Autowired

WidgetRepository widgetRepository;

@Override

**public** Dashboard findById(**int** id) {

**return** dashboardRepository.findById(id);

}

@Override

**public** Dashboard update(**int** id, Dashboard request) {

Dashboard dashboard = findById(id);

**if**(!dashboard.getWidgets()

.equals(request.getWidgets())) {

**if**(dashboard.getWidgets().isEmpty()) {

**for** (Widget widget : request.getWidgets()) {

widget.setDashboard(dashboard);

dashboard.getWidgets().add(widget);

}

dashboardRepository.save(dashboard);

**return** dashboard;

} **else** {

**for** (Widget widget : request.getWidgets()) {

**if**(widget.getId() == -1) {

widget.setDashboard(dashboard);

dashboard.getWidgets().add(widget);

dashboardRepository

.save(dashboard);

}

}

**for** (Widget widget : request.getWidgets()) {

**for** (Widget w : dashboard.getWidgets()) {

**if**(w.getId() == widget.getId() && ! w.equals(widget)) {

w.setTitle(widget.getTitle());

w.setContent(widget.getContent());

w.setDate(widget.getDate()); w.setDragAndDrop(

widget.isDragAndDrop());

w.setResizable(

widget.isResizable());

w.setX(widget.getX()); w.setY(widget.getY()); w.setW(widget.getW()); w.setH(widget.getH()); w.setxSm(widget.getxSm()); w.setySm(widget.getySm()); w.setwSm(widget.getwSm()); w.sethSm(widget.gethSm()); w.setxMd(widget.getxMd()); w.setyMd(widget.getyMd()); w.setwMd(widget.getwMd()); w.sethMd(widget.gethMd()); w.setxLg(widget.getxLg()); w.setyLg(widget.getyLg()); w.setwLg(widget.getwLg()); w.sethLg(widget.gethLg()); w.setxXl(widget.getxXl()); w.setyXl(widget.getyXl());

w.setwXl(widget.getwXl()); w.sethXl(widget.gethXl()); }

}

dashboardRepository.save(dashboard);

}

**return** dashboard;

}

}

**return** dashboard;

}

@Override

**public** Dashboard changeDashboard(**int** id, Dashboard request) {

Dashboard dashboard = findById(id);

**if**(dashboard.getWidgets().isEmpty()) {

**for** (Widget widget : request.getWidgets()) {

widget.setDashboard(dashboard);

dashboard.getWidgets().add(widget);

}

dashboardRepository.save(dashboard);

**return** dashboard;

} **else** {

@SuppressWarnings("rawtypes")

Collection result = CollectionUtils.*subtract*(request.getWidgets(), dashboard.getWidgets());

**for** (Object o : result) {

Widget widget = (Widget) o;

dashboard.getWidgets().add(widget);

dashboardRepository.save(dashboard);

}

}

**return** dashboard;

}

@Override

**public** Dashboard clearDashboard(**int** id) {

Dashboard dashboard = findById(id);

dashboard.getWidgets().clear();

dashboardRepository.save(dashboard);

**return** dashboard;

}

@Override

**public** Dashboard search(**int** id, String search) {

Dashboard dashboard = findById(id);

List<Widget> widgets = widgetRepository.findBySearch(search);

Set<Widget> set = **new** HashSet<>();

**if**(widgets.isEmpty()) {

set = dashboard.getWidgets();

} **else** {

**for** (Widget widget : widgets) {

**if**(widget.getDashboard().getId() == dashboard.getId()) {

set.add(widget);

}

}

}

dashboard.setWidgets(set);

**return** dashboard;

}

}

Slika 3.9 Implementacija DashboardService Impl klase

#### Servisi za konvertovanje

Na nivou servisa postoje i servisi za konverovanje podataka. Kako bi kod bio čistiji i pregledniji u ove servise je izdvojeno prebacivanje DTO objekata u model i obrnuto. Ove servise poziva kontroler pre pozivanja servisa koji vrši poslovnu logiku da, ukoliko je to potrebno, podatke dobijene od klijenta prebaci u model i pred samo vraćanje odgovora klijentu, da podatke dobijene od servisa koji vrši poslovnu logiku prebaci u podatke koji se šalju klijentu. Na slici 3.10 se može videti primer jednog konvertovanja podataka.

@Component

**public** **class** DashboardConverter {

@Autowired

**private** WidgetConverter widgetConverter;

**public** Dashboard convert(DashboardDTO dto) {

Dashboard dashboard = **new** Dashboard();

**if**(dto.getWidgets() != **null**) {

Set<Widget> widgets = **new** HashSet<>();

**for** (WidgetDTO widgetDTO : dto.getWidgets()) {

Widget widget = widgetConverter.convert(widgetDTO);

widgets.add(widget);

}

dashboard.setWidgets(widgets);

}

**return** dashboard;

}

**public** DashboardDTO convert(Dashboard dashboard) {

DashboardDTO dto = **new** DashboardDTO();

**if**(dashboard.getWidgets() != **null**) {

List<WidgetDTO> widgets = **new** ArrayList<>();

**for** (Widget widget : dashboard.getWidgets()) {

WidgetDTO widgetDTO = widgetConverter.convert(widget);

widgets.add(widgetDTO);

}

dto.setWidgets(widgets);

}

**return** dto;

}

}

Slika 3.10 Implementacija DashboardConverter servisa

#### Servisi za validaciju

Kao što je već rečeno, logika vezana za validaciju je izdvojena u poseban sloj servisa. Ovaj sloj servisa takođe sadrži par interfejs i implementaciju interfejsa. Ove servise poziva kontroler nakon što primi zahtev od klijenta. Ukoliko podaci iz zahteva ne odgovaraju trenutnom stanju aplikacije, servisi izazivaju izuzetak. Obrada izuzetaka je urađena tako što za svaki tip izuzetka postoji posebna klasa (slika 3.11) koje potom obrađivač izuzetka na osnovu konkretnog izuzetka šalje odgovor klijentu (slika 3.12).

@SuppressWarnings("serial")

**public** **class** BadRequestException **extends** RuntimeException {

**public** BadRequestException() {}

**public** BadRequestException(String message) {

**super**(message);

}

}

Slika 3.11 Implementacija BadRequestException klase

@ControllerAdvice

**public** **class** ExceptionResolver {

@ExceptionHandler(NotFoundException.**class**)

**public** ResponseEntity<String> notFoundException(NotFoundException exception) {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.***TEXT\_PLAIN***);

**return** **new** ResponseEntity<String>(exception.getMessage(), headers, HttpStatus.***NOT\_FOUND***);

}

@ExceptionHandler(BadRequestException.**class**)

**public** ResponseEntity<String> badRequestException(BadRequestException exception) {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.***TEXT\_PLAIN***);

**return** **new** ResponseEntity<String>(exception.getMessage(), headers, HttpStatus.***BAD\_REQUEST***);

}

@ExceptionHandler(ForbiddenException.**class**)

**public** ResponseEntity<String> forbiddenException(ForbiddenException exception) {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.***TEXT\_PLAIN***);

**return** **new** ResponseEntity<String>(exception.getMessage(), headers, HttpStatus.***FORBIDDEN***);

}

}

Slika 3.12 Implementacija obrađivača izuzetka

Na slikama 3.13 i 3.14 se može videti jedan primer servisa za validaciju.

**public** **interface** DashboardValidationService {

**public** **void** validateIfDashboardExist(**int** id);

}

Slika 3.13 Implementacija interfejsa DashboardValidationService

@Component

**public** **class** DashboardValidationServiceImpl **implements** DashboardValidationService {

@Autowired

**private** DashboardService dashboardService;

String exception = "Dashboard doesn't exist!";

@Override

**public** **void** validateIfDashboardExist(**int** id) {

**if**(dashboardService.findById(id) == **null**) {

**throw** **new** NotFoundException(exception);

}

}

}

Slika 3.14 Implementacija klase DashboardValidationServiceImpl

### Data Transfer Object

Ovaj sloj podataka čine objekti koji se prenose preko mreže. Implementiran je kako se ne bi kompletan model slao preko mreže jer za tim u većini slučajeva nema potrebe. Jedan primer DTO objekta je na slici 3.15. Ukoliko se uporedi sa modelom sa slike 3.3 može se zaključiti da se sadrži manje atributa nego ceo model.

**public** **class** DashboardDTO {

**private** List<WidgetDTO> widgets;

}

Slika 3.15 Implementacija DashboardDTO klase

### Controller

Sloj kontrolera služi za komunikaciju sa klijentom i oslanja se na servisni sloj. Zadatak kontrolera je da primi zahtev sa klijentske strane, prosledi podatke servisnom sloju i potom vrati odgovor. Ovaj sloj čine dva kontrolera, UserControler i DashboardController. UserController sadrži metode za rad sa korisnikom a DashboardController koji sadrži metode za rad sa beleskama. Metode kontrolera su implementirane tako da se prvo pozivaju servisi za validaciju, potom servisi za konvertovanje podatka, nakon toga se poziva servis koji vrši poslovnu logiku i nakon toga opet servis za konverovanje koji izmenjene podatke konvertuje u podatke koji se šalju klijentu. Detaljna implementacija kontrolera se može videti na slici 3.16.

@RestController

@RequestMapping("/api/dashboards")

**public** **class** DashboardController {

@Autowired

**private** UserValidationService userValidator;

@Autowired

**private** DashboardValidationService dashboardValidator;

@Autowired

**private** DashboardConverter converter;

@Autowired

**private** DashboardService service;

@RequestMapping(

value = "/{id}",

method = RequestMethod.***GET***

)

**public** ResponseEntity<DashboardDTO> getById(@PathVariable **int** id, Authentication authentication) {

userValidator.validateIfUserExist(authentication);

dashboardValidator.validateIfDashboardExist(id);

Dashboard dashboard = service.findById(id);

DashboardDTO response = converter.convert(dashboard);

**return** **new** ResponseEntity<DashboardDTO>(response, HttpStatus.***OK***);

}

@RequestMapping(

value = "/{id}",

method = RequestMethod.***PUT***

)

**public** ResponseEntity<DashboardDTO> update(@PathVariable **int** id, @RequestBody DashboardDTO request, Authentication authentication) {

userValidator.validateIfUserExist(authentication);

dashboardValidator.validateIfDashboardExist(id);

Dashboard dashboard = service.update(id, converter.convert(request));

DashboardDTO response = converter.convert(dashboard);

**return** **new** ResponseEntity<DashboardDTO>(response, HttpStatus.***CREATED***);

}

@RequestMapping(

value = "/{id}/change",

method = RequestMethod.***PUT***

)

**public** ResponseEntity<DashboardDTO> change(@PathVariable **int** id, @RequestBody DashboardDTO request, Authentication authentication) {

userValidator.validateIfUserExist(authentication);

dashboardValidator.validateIfDashboardExist(id);

Dashboard dashboard = service.changeDashboard(id, converter.convert(request));

DashboardDTO response = converter.convert(dashboard);

**return** **new** ResponseEntity<DashboardDTO>(response, HttpStatus.***CREATED***);

}

@RequestMapping(

value = "/{id}",

method = RequestMethod.***DELETE***

)

**public** ResponseEntity<DashboardDTO> delete(@PathVariable **int** id, Authentication authentication) {

userValidator.validateIfUserExist(authentication);

dashboardValidator.validateIfDashboardExist(id);

Dashboard dashboard = service.clearDashboard(id);

DashboardDTO response = converter.convert(dashboard);

**return** **new** ResponseEntity<DashboardDTO>(response, HttpStatus.***OK***);

}

@RequestMapping(

value = "/{id}/search",

method = RequestMethod.***GET***

)

**public** ResponseEntity<DashboardDTO> search(@PathVariable **int** id, @RequestParam String search, Authentication authentication) {

userValidator.validateIfUserExist(authentication);

dashboardValidator.validateIfDashboardExist(id);

Dashboard dashboard = service.search(id, search);

DashboardDTO response = converter.convert(dashboard);

**return** **new** ResponseEntity<DashboardDTO>(response, HttpStatus.***OK***);

}

}

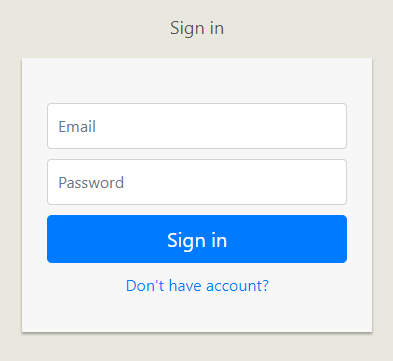
Slika 3.16 Implementacija DashboardController klase

## Klijentski deo

Klijentski deo predstavlja *view* komponentu u arhitekturi ove aplikacije. Uloga ove komponente je prvenstveno prikaz podataka, međutim ova komponenta je kompleksnija od samog prikaza. Za implementaciju ovog dela aplikacije korišćen je Angular okruženje, Gridster biblioteka za rad sa beleškama. Aplikacija je razvijana kao SPA što znači da se korisniku prikazuje samo jedna stranica čiji se sadržaj dinamički menja promenom stanja. Što se tiče samog izgleda aplikacije korišćene su Bootstrap i Material Design biblioteke. Bootstrap kako bi se sadžaj adekvatno prikazivao na različitim uređajima, a Material Design kako bi izgled bio sličniji onom koji ima originalna aplikacija.

Svaka od stranica predstavlja jedno stanje aplikacije, a u Angular aplikaciji jednu Angular komponentu. Više o Angular komponentama opisano je u poglavlju 2.2.1. Svaka komponenta ima svoj HTML *template* i TS klasu.

Početno stanje koje se prikazuje klijentu je stanje „login“ kao što se može videti na slici 3.17.



Slika 3.17 Login stranica

Funkcionalnost koja se nalazi iza ove stranice se može videti na slikama 3.18 i 3.19. Na slici 3.18 se nalazi HTML template, a na slici 3.19 TS klasa.

<div class="container">

<div class="row justify-content-md-center">

<div class="col-sm-6 col-md-4 col-md-offset-4">

<h1 class="text-center login-title">Sign in</h1>

<div class="account-wall">

<form class="form-signin">

<input type="email" class="form-control" placeholder="Email" name="email" [(ngModel)]="user.email" required autofocus>

<input type="password" class="form-control" placeholder="Password" name="password" [(ngModel)]="user.password" required autofocus>

<button class="btn btn-lg btn-primary btn-block" type="submit" (click)="login()">Sign in</button>

<a routerLink="/register" routerLinkActive="active"

class="text-center new-account">Don't have account?</a>

<span class="clearfix"></span>

</form>

</div>

</div>

</div>

</div>

Slika 3.18 Login HTML template

Kao što se može videti na slici 3.18, na formi za prijavu na sistem postoje dva polja za unos korisničkog imena i lozinke na koja su pomoću ngModel vezana za objekat koji se nalazi u TS klasi.

LoginComponent klasa u svom javnom API-u sadrži objekat *user* čiji se atributi popunjavaju nakon što korisnik na stranici unese podatke. Konstruktor ove komponente putem *dependency injection*-a dobija objekte iz DashboardService i Router klase. Unutar DasboardService klase se nalaze metode za komunikaciju sa klijentom, a Router klasa sadrži metode za navigaciju između komponenti.

@Component({

selector: 'app-login',

templateUrl: './login.component.html',

styleUrls: ['./login.component.css']

})

export class LoginComponent implements OnInit {

public user = { email: "", password: "" };

constructor(private service: UserService,

private router: Router) { }

ngOnInit() {

}

login() {

this.service.login(this.user).subscribe((data) => {

localStorage.setItem('token', JSON.stringify(data.token));

localStorage.setItem('notes', JSON.stringify(data.user.notesDashboard.widgets));

localStorage.setItem('notesID', JSON.stringify(data.user.notesId));

localStorage.setItem('archive', JSON.stringify(data.user.archiveDashboard.widgets));

localStorage.setItem('archiveID', JSON.stringify(data.user.archiveId));

localStorage.setItem('trash', JSON.stringify(data.user.trashDashboard.widgets));

localStorage.setItem('trashID', JSON.stringify(data.user.trashId));

this.router.navigateByUrl('/home/dashboard');

});

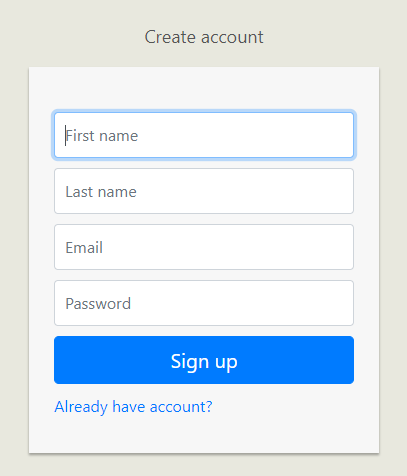
}

}

Slika 3.19 LoginComponent klasa

Nakon što korisnik pritisne dugme Sign in, poziva se login() metoda ove komponente koja dalje poziva metodu za prijavu na sistem iz DashboardService klase. Ukoliko je prijava na sistem uspešna, metoda iz servisa vraća podatke vezane za korisnika i smešta ih u *localStorage* klijenta i prebacije korisnika u novo stanje.

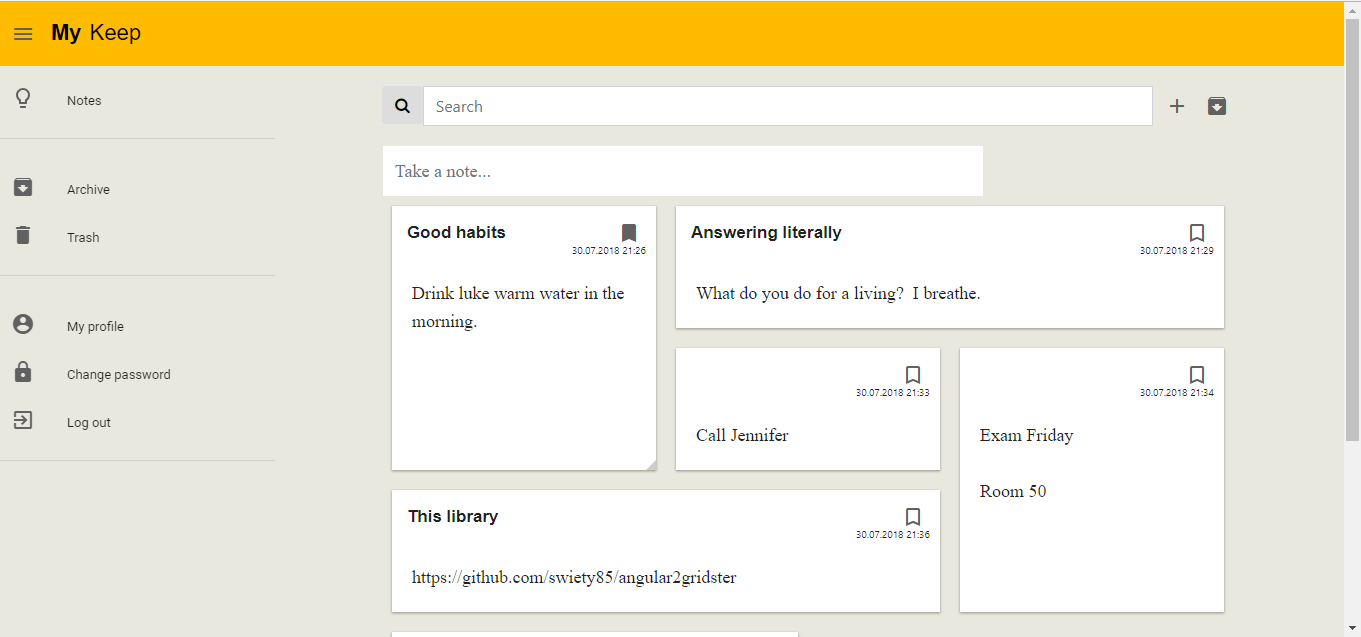
Ukoliko korisnik još nema nalog na sistemu, može da kreira nalog. Ukoliko izabere kreiranje naloga, klijent se prebacuje u novo stanje i prikazuje mu se sledeća stranica (slika 3.20).



Slika 3.20 Stranica za kreiranje naloga

Nakon uspešne registracije korisnik je preusmeren na stranicu za prijavu na sistem. Tek kada se korisnik prijavi prikazuje se glavna stranica. Ukoliko se korisnik tek registrovao prikazaće mu se stranica sa praznom tablom, a ukoliko je korisnik već koristio aplikaciju prikazaće mu se stranica sa aktivnim beleškama (slika 3.21).

Na glavnoj stranici, pored table sa beleškama postoji i sidebar meni iz kog korisnik može izabrati da ode na stranicu sa arhiviranim beleškama, na stranicu sa beleškama za brisanje, može da pogleda svoj profil, izmeni lozinku ili se odjavi sa sistema.

Slika 3.21 Početna stranica za aktivnog korisnika

Dok je korisnik na stranici na kojoj se prikazuju aktivne beleške aktivna je komponenta DashboardComponent. Na slici 3.22 se može videti HTML *template* za ovu komponentu. Zbog preglednosti HTML *template* se nalazi na dve slike.

<div class="buttons1">

<div class="input-group">

<span class="input-group-btn">

<button class="btn btn-default" type="button" (click)="populate(gridster2)"><i class="fa fa-search"></i>

</button>

</span>

<input type="text" class="form-control" placeholder="Search" [(ngModel)]="search"

(keydown)="searchIt($event,gridster2)"> &nbsp;

<button mat-icon-button

(click)="addWidgetWithoutData()"

title="Add widget without data">

<mat-icon class="icon">add</mat-icon>

</button>

<button mat-icon-button

(click)="removeAllWidgets()"

title="Archive all widgtes">

<mat-icon class="icon">archive</mat-icon>

</button>

</div>

</div>

<div class="omnibar">

<div \*ngIf="!open" class="toolbar-closed">

<div class="form-group" (click)="onopen()">

<input type="text" class="form-control" placeholder="Take a note..."></div>

</div>

<div \*ngIf="open" class="toolbar-open">

<div class="form-group" (keydown)="addWidgetWithData($event)"

<input type="text" class="form-control" placeholder="Title" [(ngModel)]="title">

<textarea rows="5" class="form-control" placeholder="Take a note..."

[(ngModel)]="content"></textarea>

<button class="form-control" (click)="onclose()">CLOSE</button>

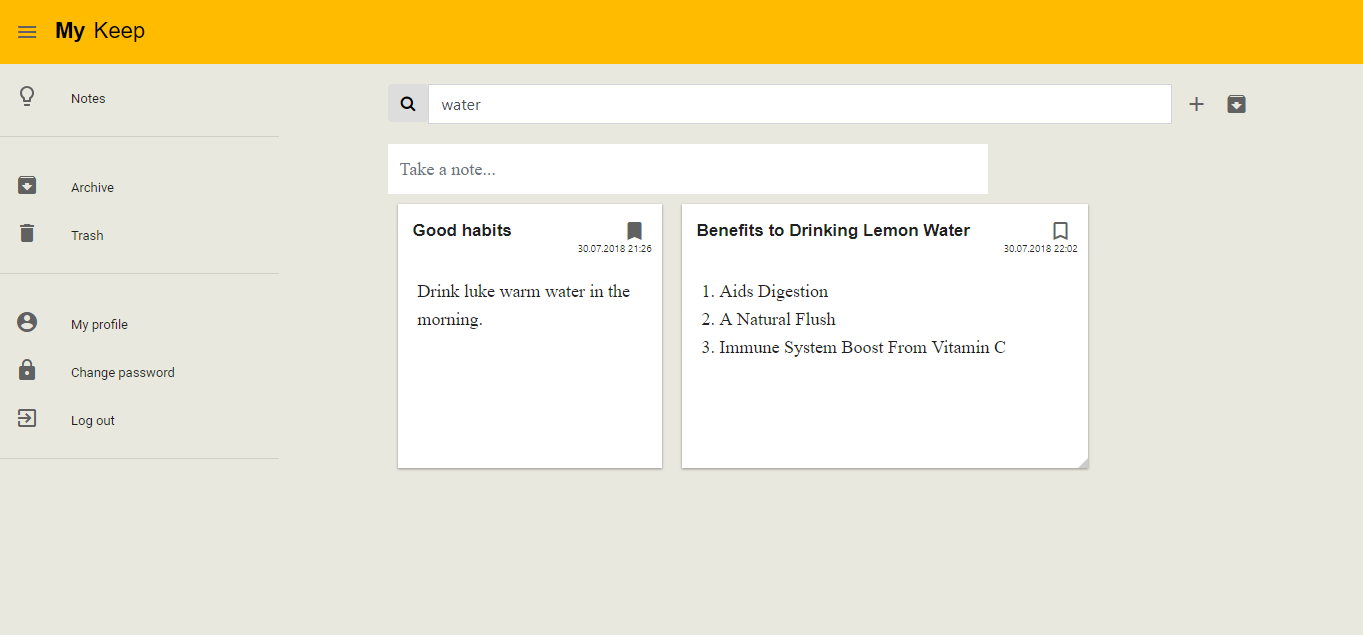
</div>

</div>

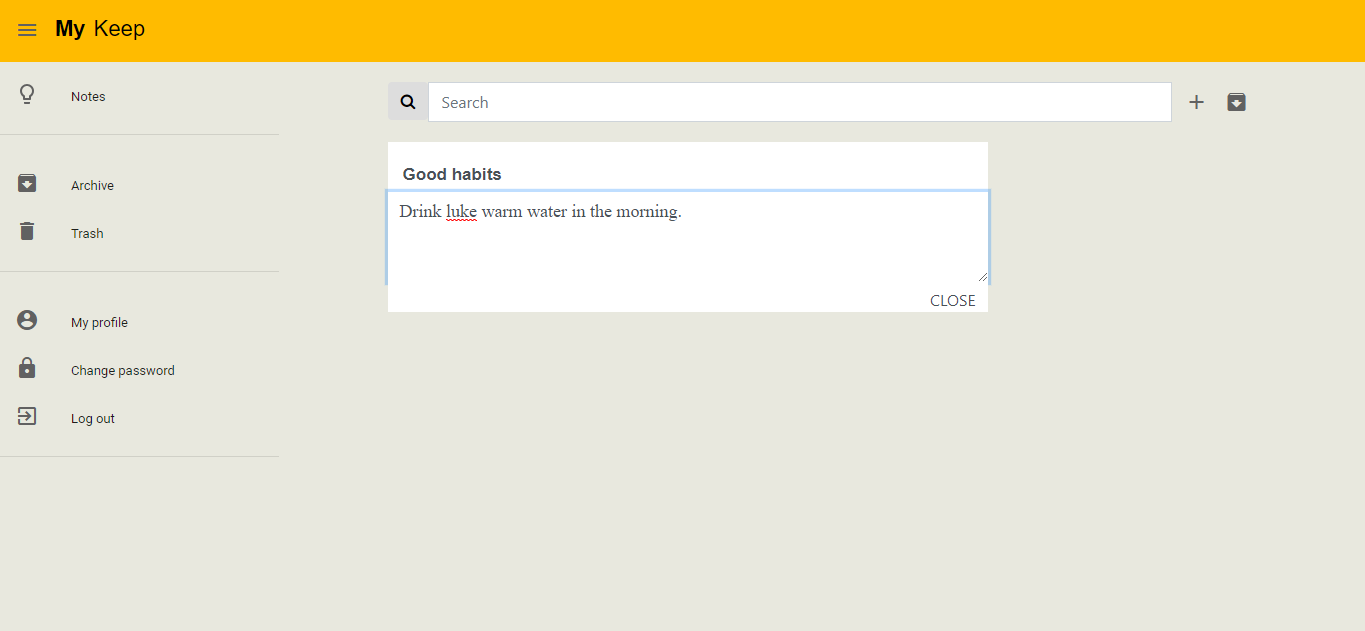
</div>

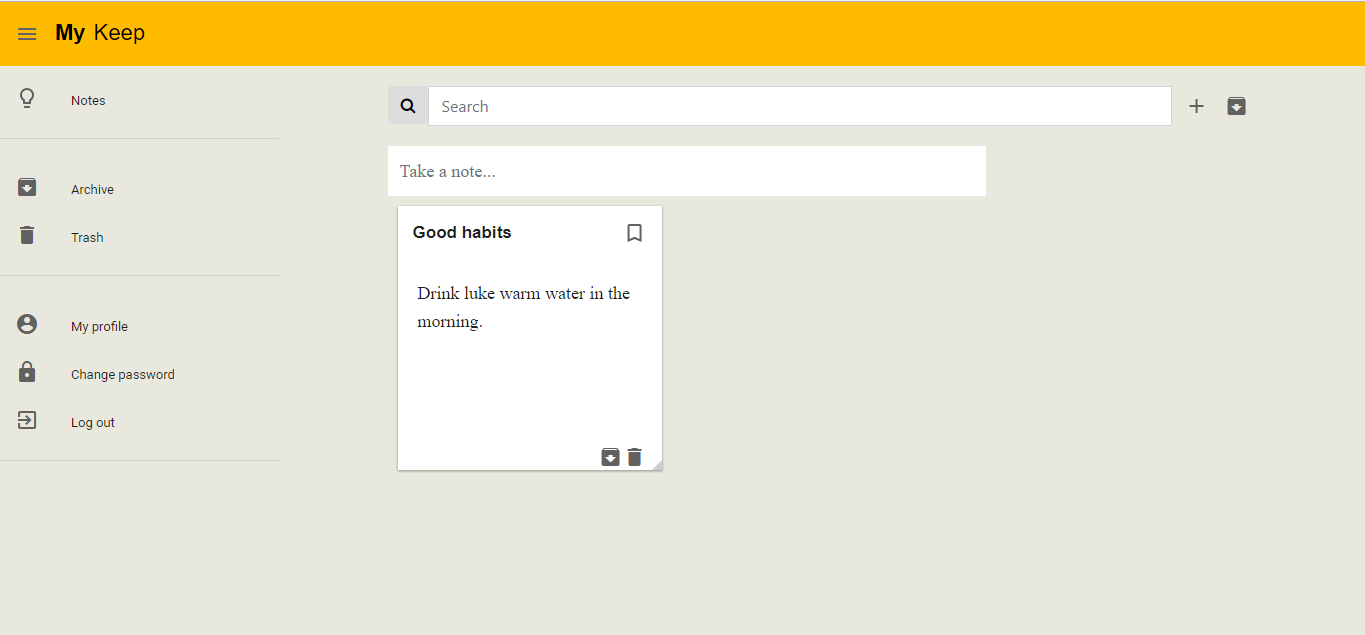
Slika 3.22 DashboardComponent HTML template 1

Pretraga beleški se može vršiti po naslovu, sadržaju beleški ili datumu kada su kreirane. Na slici 3.23 se može videti rezultat pretrage za zadati upit.

Slika 3.23 Home sranica, prikaz rezultata pretrage

Kreiranje beleške se vrši tako što se klikne na polje za unos i unesu sadržaj i naslov beleške. Na slici 3.24 se vidi postupak kreiranja nove beleške a potom i rezultat kreiranja (slika 3.25).

Slika 3.24 Home stranica, dodavanje nove beleške

Slika 3.2 Home stranica, stanje nakon dodate nove beleške

Prilikom dodavanja nove beleške ona se dodaje u listu svih beleški na tabli, a nakon određenog vremena se poziva servisna metoda koja šalje zahtev za upis beleški u bazu. Funkcija za dodavanje beleške u listu beleški se nalazi na slici 3.26, a poziv servisne metode za upis na slici 3.27.

public addWidgetWithData($event) {

if ($event.key === "Enter") {

this.notes.unshift({

id: -1,

w: 2, h: 2,

wSm: 2, hSm: 2, wMd: 2, hMd: 2,

wLg: 2, hLg: 2, wXl: 2, hXl: 2,

title: this.title, content: this.content,

dragAndDrop: true, resizable: true,

dashboardId: this.notesID

});

this.open = false;

this.title = ''; this.content = '';

this.counter1++;

}

}

Slika 3.26 Funkcija za dodavanje beleški u listu

public update() {

this.notesDTO.widgets = JSON.parse(localStorage.getItem('notes'));

this.dashboardService.update(this.notesID, this.notesDTO)

.subscribe(

data => {

localStorage.removeItem('notes');

localStorage.setItem('notes', JSON.stringify(data.widgets));

this.notes = data.widgets;

}, error => {

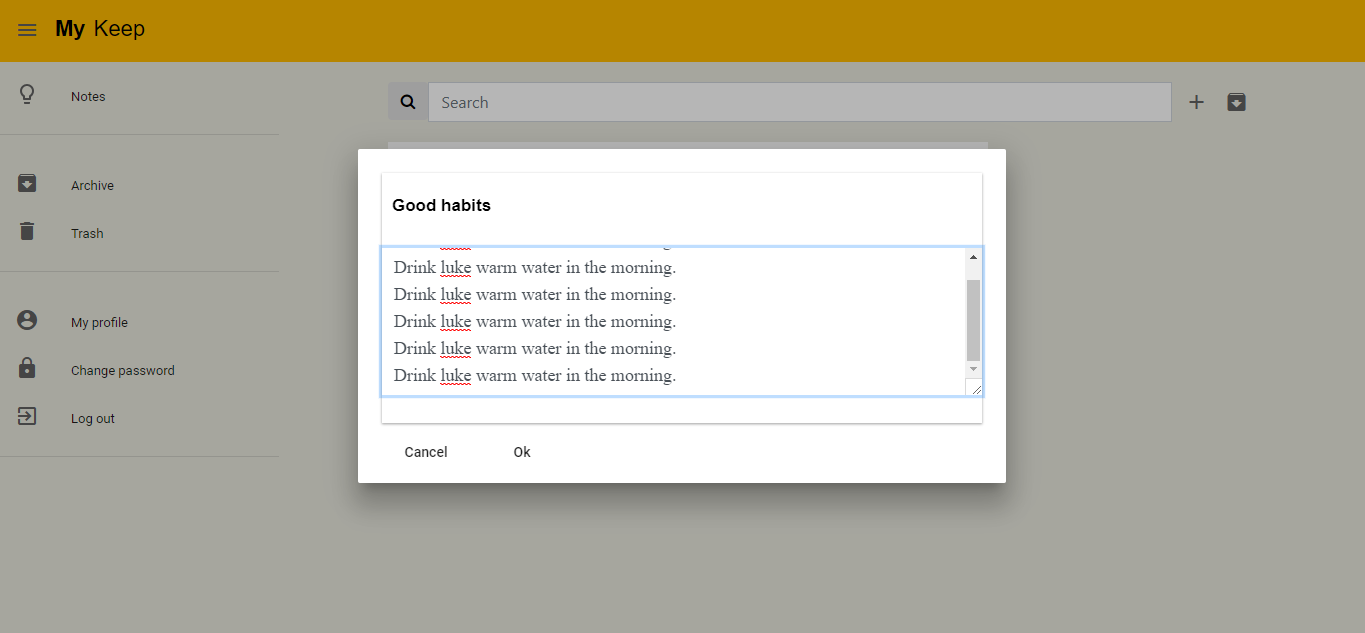
console.log(error);

})

}

Slika 3.27 Poziv servisne metode za upis beleški

Nakon što je beleška sačuvana ona može biti ažurirana, tj. može biti promenjen njen sadržaj, može se zakačiti na trenutnu lokaciju, može se prebaciti u arhivu ili obrisati, ili joj se može promeniti veličina. Na slici 3.28 se vidi primer promene sadržaja.

Slika 3.28 Home stranica, izmena sadržaja postojeće beleške

Na slici 3.29 prikazan je deo DashboardComponent HTML *template*-a na kom se nalazi tabla sa beleškama. Tabla sa beleškama je predstavljena GridsterComponent komponentom iz Gridster biblioteke. Za nju su definisani atributi *options* i *draggableOptions* i funkcije *reflow* i *optionsChange*. GridsterComponent sadrži GridsterItemComponent koja predstavlja jednu belešku na tabli. Za ovu komponentu su definisani standardni atributi i funkcije a pored njih i neke dodatne metode poput *pin*/*unpin* metoda koje služe da se beleška zakači na trenutnu poziciju, kao i metode *remove* i *delete*, gde *remove* prebacije selektovanu belešku u arhivu, dok *delete* prebacuje belešku na stranicu sa beleškama za brisanje. Bitna osobine ove komponente je odgovarajući pregled zavisno od veličine ekrana tj. *responsive view*. Stoga postoje definisani atributi x, y, w i h zavisno od same veličine ekrana. Na slici 1.11 se može videti podešavanje zavisno od veličine ekrana. Iako je sama aplikacija koristi Bootstrap biblioteku, za prikaz samih beleški se koriste dodatni atributi.

<ngx-gridster

[options]="gridsterOptions" [draggableOptions]="gridsterDraggableOptions" (reflow)="onReflow($event)" (optionsChange)="optionsChange($event)"

#gridster2 class="dashboard">

<ngx-gridster-item \*ngFor="let widget of notes; let indx = index"

#itemComp

[options]="itemOptions"

[dragAndDrop]="widget.dragAndDrop"

[resizable]="widget.resizable"

[(x)]="widget.x" [(y)]="widget.y"

[(xSm)]="widget.xSm" [(ySm)]="widget.ySm" [(xMd)]="widget.xMd" [(yMd)]="widget.yMd" [(xLg)]="widget.xLg" [(yLg)]="widget.yLg"

[(xXl)]="widget.xXl" [(yXl)]="widget.yXl"

[(w)]="widget.w" [(h)]="widget.h"

[(wSm)]="widget.wSm" [(hSm)]="widget.hSm" [(wMd)]="widget.wMd" [(hMd)]="widget.hMd" [(wLg)]="widget.wLg" [(hLg)]="widget.hLg" [(wXl)]="widget.wXl" [(hXl)]="widget.hXl" (xSmChange)="xSmChange(widget, indx)" (ySmChange)="ySmChange(widget, indx)"

(hSmChange)="hSmChange(widget, indx)" (wSmChange)="wSmChange(widget, indx)" (xMdChange)="xMdChange(widget, indx)"

(yMdChange)="yMdChange(widget, indx)" (hMdChange)="hMdChange(widget, indx)" (wMdChange)="wMdChange(widget, indx)"

(xLgChange)="xLgChange(widget, indx)" (yLgChange)="yLgChange(widget, indx)" (hLgChange)="hLgChange(widget, indx)"

(wLgChange)="wLgChange(widget, indx)" (xXlChange)="xXlChange(widget, indx)" (yXlChange)="yXlChange(widget, indx)"

(hXlChange)="hXlChange(widget, indx)" (wXlChange)="wXlChange(widget, indx)"

(change)="itemChange($event)">

<div class="panel-heading">

<div class="panel-title">{{ widget.title }}

<a \*ngIf="widget.dragAndDrop" (click)="pin(widget)"

class="pull-right" title="Pin it">

<mat-icon class="icon">bookmark\_border

</mat-icon>

</a>

<a \*ngIf="!widget.dragAndDrop"

(click)="unpin(widget)"

class="pull-right" title="Unpin">

<mat-icon class="icon">bookmark

</mat-icon>

</a>

</div>

</div>

<div class="panel-body" (click)="openDialog(widget)"

(mouseover)="onhover(widget)" (mouseleave)="onleave(widget)">

<small>{{widget.date|date:"dd.MM.y HH:mm"}}</small>

<br>

<div class="panel-content">

{{ widget.content }}</div>

<div \*ngIf="widget.hover"

class="panel-content-bottom">

<a (click)="delete(widget, indx)"

class="pull-right"

title="Delete">

<mat-icon class="icon">delete

</mat-icon>

</a>

<a (click)="remove(widget, indx)"

class="pull-right" title="Archive">

<mat-icon class="icon">archive

</mat-icon>

</a>

</div>

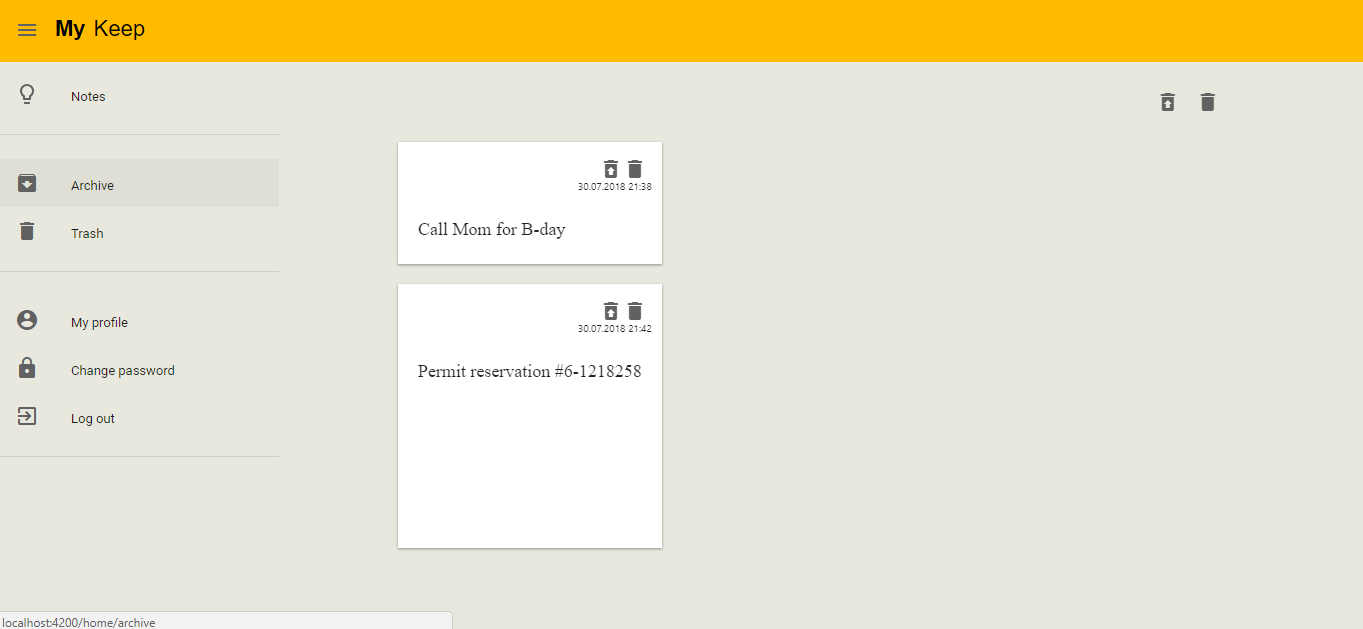
</div>

</ngx-gridster-item>

</ngx-gridster>

Slika 3.29 DashboardComponent HTML template 2

Ukoliko je korisnik neke beleške prebacio u arhivu, njih može pregledati ako iz sidebar menija izabere karticu Archive (slika 3.30). Tabla sa beleškama u arhivi je slična onoj sa aktivnim beleškama samo što korisnik samo ima mogućnost da prebaci belešku među beleške za brisanje ili opozove arhiviranje i vrati je među aktivne beleške.

Slika 3.30 ArchiveComponent HTML template

Prilikom arhiviranja beleške, ona se prebacuje u listu sa arhiviranim beleškama što se može videti na slici 3.31, a potom se poziva metoda koja poziva servisnu metodu koja vrši arhiviranje, tj. promenu table u kojoj se beleška nalazi (slika 3.32).

public remove(widget: Widget, index: number) {

this.archive.unshift({

id: widget.id,

x: widget.x, y: widget.y,

xSm: widget.xSm, ySm: widget.ySm,

xMd: widget.xMd, yMd: widget.yMd,

xLg: widget.xLg, yLg: widget.yLg,

xXl: widget.xXl, yXl: widget.yXl,

w: widget.w, h: widget.h,

wSm: widget.wSm, hSm: widget.hSm,

wMd: widget.wMd, hMd: widget.hMd,

wLg: widget.wLg, hLg: widget.hLg,

wXl: widget.wXl, hXl: widget.hXl,

title: widget.title,

content: widget.content,

dragAndDrop: widget.dragAndDrop,

resizable: widget.resizable,

date: widget.date,

dashboardId: this.archiveID

});

localStorage.removeItem('archive');

localStorage.setItem('archive', JSON.stringify(this.archive));

this.notes.splice(index, 1);

localStorage.removeItem('notes');

localStorage.setItem('notes', JSON.stringify(this.notes));

this.counter2++;

}

Slika 3.31 Funkcija za dodavanje beleški u listu arhiviranih

public moveToArchive() {

this.archiveDTO.widgets = JSON.parse(localStorage.getItem('archive'));

this.dashboardService.changeDashboard(this.archiveID, this.archiveDTO)

.subscribe(

data => {

localStorage.removeItem('archive');

localStorage.setItem('archive', JSON.stringify(data.widgets));

this.archive = data.widgets;

}, error => {

console.log(error);

})

}

Slika 3.32 Poziv servisne metode za promenu table

Ukoliko korisnik belešku prebaci među obrisane, određeni period ima priliku da ih pregleda, sam pregled beleški za brisanje se sličan pregledu arhiviranih beleški(slika 3.30) dok su jedine akcije koje može da izvrši opoziv prebacivanja među beleške za brisanje. Sam postupak prebacivanja beleški među beleške za brisanje je sličan postupku arhiviranja prikazanom na slici 3.32. Nakon određenog vremena beleške se automatski brišu iz sistema a na slici 3.33 se nalazi poziv servisne metode za brisanje.

public deleteAllForever(gridster: GridsterComponent) {

this.dashboardService.delete(this.trashID)

.subscribe(

data => {

this.trash = [];

localStorage.removeItem('trash');

localStorage.setItem('trash', JSON.stringify(this.trash));

gridster.reload();

})

}

Slika 3.33 Poziv servisne metode za brisanje beleški

# Zaključak

Projekat predstavlja jednostavnu aplikaciju za upravljanje beleškama inspirisanu već postojećim rešenjem. Dosadašnji rad obuhvata osnovni skup operacija za upravljanje beleškama poput dodavanja, izmene i brisanja kao i upravljanja samim rasporedom, slaganjem i prikazom beleški.

Aplikacija je razvijana u trenutno aktuelnim tehnologijama i ima dobru osnovu za dalji razvoj. I sama biblioteka koja je korišćena za upravljanje beleškama na klijentskoj strani aplikacije je u razvoju tako da su proširenja moguća.

LITERATURA

1. *Google Keep*

[*https://keep.google.com/*](https://keep.google.com/)

1. *Java*

[*https://docs.oracle.com/javase/8/docs/api/*](https://docs.oracle.com/javase/8/docs/api/)

1. *Spring okruženje*

[*https://spring.io/*](https://spring.io/)

1. *Spring Boot*

[*https://spring.io/projects/spring-boot*](https://spring.io/projects/spring-boot)

1. *Spring Data*

[*https://spring.io/projects/spring-data*](https://spring.io/projects/spring-data)

1. *Angular 6 okruženje*

[*https://angular.io/*](https://angular.io/)

1. *Gridster biblioteka*

[*https://github.com/swiety85/angular2gridster/wiki/Getting-started*](https://github.com/swiety85/angular2gridster/wiki/Getting-started)

1. *Angular CLI*

[*https://cli.angular.io/*](https://cli.angular.io/)

1. *Dependecy injection*

[*https://en.wikipedia.org/wiki/Dependency\_injection*](https://en.wikipedia.org/wiki/Dependency_injection)

1. *REST*

[*https://en.wikipedia.org/wiki/Representational\_state\_transfer*](https://en.wikipedia.org/wiki/Representational_state_transfer)

1. *HTTP*

[*https://en.wikipedia.org/wiki/Hypertext\_Transfer\_Protocol*](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)

1. *Spring MVC*

[*https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html*](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html)

1. *JavaScript*

[*https://en.wikipedia.org/wiki/JavaScript*](https://en.wikipedia.org/wiki/JavaScript)

# 6.KLJUČNA DOKUMENTACIJSKA INFORMACIJA

|  |  |  |  |
| --- | --- | --- | --- |
| Редни број, **РБР**: | |  | |
| Идентификациони број, **ИБР**: | |  | |
| Тип документације, **ТД**: | | монографска публикација | |
| Тип записа, **ТЗ**: | | текстуални штампани документ | |
| Врста рада, **ВР**: | | дипломски-бечелор рад | |
| Аутор, **АУ**: | | Ивана Савин | |
| Ментор, **МН**: | | Проф Др Милан Видаковић | |
| Наслов рада, **НР**: | | Веб апликација за вођење белешки | |
| Језик публикације, **ЈП**: | | српски | |
| Језик извода, **ЈИ**: | | српски / енглески | |
| Земља публикововања, **ЗП**: | | Србија | |
| Уже географско подручје, **УГП**: | | Војводина | |
| Година, **ГО**: | | 2018 | |
| Издавач, **ИЗ**: | | ауторски репринт | |
| Место и адреса, **МА**: | | Нови Сад, Факултет техничких наука, | |
| Физички опис рада, **ФО**:  (поглавља/страна/цитата/табела/слика/графика/прилога) | | 5 / 58 / 0 / 0 / 47/ 0 / 0 | |
| Научна област, **НО**: | | Информатика | |
| Научна дисциплина, **НД**: | | Рачунарске науке | |
| Предметна одредница/Кључне речи, **ПО**: | | Јava, ВЕБ, REST, Angular 6 | |
| **УДК** | |  | |
| Чува се, **ЧУ**: | | Библиотека Факултета техничких наука, Трг Доситеја Обрадовића 6, Нови Сад | |
| Важна напомена, **ВН**: | |  | |
| Извод, **ИЗ**: | | Задатак рада представља развој једноставне веб апликације за управљање белешкама инспирисану већ постојећим решењем (Google Keep). Серверски део апликације ће бити реализован у програмском језику Јава, коришћењем Spring окружења а клијентски део у Angular 6 окружењу. Апликација омогућава креирање белешки, размештање по екрану и њихово ажурирање. | |
| Датум прихватања теме, **ДП**: | |  | |
| Датум одбране, **ДО**: | |  | |
| Чланови комисије, **КО**: | Председник: | др Игор Дејановић, ванредни проф., ФТН Нови Сад |  |
|  | Члан: | др Стеван Гостојић, ванредни проф., ФТН Нови Сад | Potpis mentora |
|  | Члан,ментор: | др Милан Видаковић, ред. проф., ФТН Нови Сад |  |

# 7.KEY WORDS DOCUMENTATION

|  |  |  |  |
| --- | --- | --- | --- |
| Accession number, **ANO**: | |  | |
| Identification number, **INO**: | |  | |
| Document type, **DT**: | | monographic publication | |
| Type of record, **TR**: | | textual material | |
| Contents code, **CC**: | | BSc thesis | |
| Author, **AU**: | | Ivana Savin | |
| Mentor, **MN**: | | Prof Dr Milan Vidaković | |
| Title, **TI**: | | WEB APPLICATION FOR NOTES MANAGEMENT | |
| Language of text, **LT**: | | Serbian | |
| Language of abstract, **LA**: | | Serbian / English | |
| Country of publication, **CP**: | | Serbia | |
| Locality of publication, **LP**: | | Vojvodina | |
| Publication year, **PY**: | | 2018 | |
| Publisher, **PB**: | | author’s reprint | |
| Publication place, **PP**: | | Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6 | |
| Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes) | | 5 / 58 / 0 / 0 / 47/ 0 / 0 | |
| Scientific field, **SF**: | | Electrical Engineering | |
| Scientific discipline, **SD**: | | Computer Science | |
| Subject/Key words, **S**/**KW**: | | Јava, ВЕБ, REST, Angular 6 | |
| **UC** | |  | |
| Holding data, **HD**: | | Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad | |
| Note, **N**: | |  | |
| Abstract, **AB**: | | The thesis deals with the implementation of a simple web application for notes management inspired by already existing solution. The back-end part is implemented using Java programming language and Spring framework, and  the front-end is implemented using Angular 6 framework. The application will provide basic managing features. | |
| Accepted by the Scientific Board on, **ASB**: | |  | |
| Defended on, **DE**: | |  | |
| Defended Board, **DB**: | President: | Igor Dejanović, PhD, assoc. prof., FTN Novi Sad |  |
|  | Member: | Stevan Gostojić, PhD, assoc. prof., FTN Novi Sad | Menthor's sign |
|  | Member, Mentor: | Milan Vidaković, PhD, full prof., FTN Novi Sad |  |

Biografija

Ивана Савин, рођена 29.01.1995. у Зрењанину. Завршила Основну школу „Бранко Радичевић" у Александрову и Зрењанинску гимазију у Зрењанину.

Контакт емајл адреса:

ivanasavin95[@gmail.com](mailto:pepicsrdjan134@gmail.com)

Контакт мобилни:

065/84-76-018