|  |  |  |
| --- | --- | --- |
|  | UNIVERZITET “UNION”  **RAČUNARSKI FAKULTET**  Knez Mihailova 6/VI  11000 Beograd | Broj: |
| Datum: |

DIPLOMSKI RAD

Kandidat: Jovan Lazić

Broj indeksa: RN 114/20

Tema rada

Razvoj veb aplikacije učenja na daljinu u Java i Angular okruženju

Mentor rada: prof dr Vidaković Milan

Apstrakt

Java je programski jezik opšte namene. Koristi se za razvoj desktop, *web*, konzolnih aplikacija, igrica, umrežavanje računara i mnogo drugih oblasti. Poznata je po tome što se jednom napisani Java kod može pokrenuti na bilo kojoj mašini. Razvoj Jave počeo je u maju 1995. godine.

Kako je početkom 21. veka potreba za *web* aplikacijama naglo porasla, tako je i razvoj tehnologija za razvoj *web* aplikacija isto porastao. *Spring* *framework* je aplikacioni *framework* koji se koristi na Java platformi. Popularan je među programerima kao dodatak *Enterprise JavaBeans-u*. Smatra se lakim za korišćenje zato što programeru pruža pokretanje Java servera uz minimalno podešavanja konfiguracije.

*JavaScript* je već 20 godina dominantan jezik za izradu *web* stranica. Uprkos tome, razvoj jezika je počeo pre nešto manje od 10 godina.

*TypeScript* je programski jezik koji je takođe i nadskup *JavaScript* jezika. On podržava razvoj klijentske strane i serverske strane *web* aplikacija. Omogućava korisnicima da koriste stroge tipove podataka pri pisanju koda, koje *JavaScript* ne podržava, kako bi omogućio detekciju grešaka tokom faze razvoja.

*Angular* jeste *web framework* koji je baziran na *TypeScriptu*. Popularan je zato što uvodi SPA(*Single Page Application*) koji umesto kreiranje nove stranice prilikom svakog rutiranja, dinamički osvežava trenutnu stranicu novim podacima, čime doprinosi bržem odzivu prikaza.

*Angular* aplikacije su zasnovane na komponentama. Komponente predstavljaju delove korisničkog interfejsa i sadrže *HTML*, *TypeScript* i *CSS* fajlove koji zajedno komuniciraju kako bi stvorili određenu funkcionalnost. Ključni koncept u *Angularu* je ubrizgavanje zavisnosti u komponente, čime se olakšava održavanje, razvoj i testiranje aplikacije.

Prvi deo diplomskog rada će sadržati kratak istorijat Java i *JavaScript* jezika i novitete koje donosi rad u *Spring* *framework-u*, kao i biblioteke i zavisnosti koje se koriste prilikom razvoja *web* aplikacija.

Drugi deo diplomskog rada prati razvoj aplikacije učenja na daljinu u Java *Spring* i *Angular* tehnologiji, serverskoj i klijentskoj strani, respektivno. U ovom delu će biti detaljnije objašnjen razvoj komponenti i serverske logike potrebne da bi jedna aplikacija funkcionisala.

Cilj diplomskog rada je praćenje razvoja *web* aplikacije prilikom izrade aplikacije učenja na daljinu.

Sadržaj

[1 Uvod 4](#_Toc159021264)

[2 Tehnologije korišćene u radu 5](#_Toc159021265)

[2.1 Bekend tehnologije 5](#_Toc159021266)

[2.1.1 Java 5](#_Toc159021267)

[2.1.2 Spring framework 6](#_Toc159021268)

[2.1.3 Maven 8](#_Toc159021269)

[2.1.4 PostgreSQL 8](#_Toc159021270)

[2.1.5 Lombok 9](#_Toc159021271)

[2.1.6 JSON Web Token 9](#_Toc159021272)

[2.2 Frontend tehnologije 10](#_Toc159021273)

[2.2.1 Typescript 10](#_Toc159021274)

[2.2.2 Angular 11](#_Toc159021275)

[2.2.3 Angular CLI 13](#_Toc159021276)

[3 Specifikacije aplikacije 14](#_Toc159021277)

[3.1 Dijagram slučajeva korišćenja 14](#_Toc159021278)

[3.2. Dijagram aktivnosti 15](#_Toc159021279)

[4 Opis implementacije aplikacije 16](#_Toc159021280)

[4.1 Inicijalizacija aplikacije 16](#_Toc159021281)

[4.2 Korisnici i uloge 19](#_Toc159021282)

[4.3 Registracija i prijavljivanje 21](#_Toc159021283)

[4.4 Kreiranje kurseva 26](#_Toc159021284)

[4.5 Upravljanje kursevima 28](#_Toc159021285)

[4.6 Kreiranje lekcija 29](#_Toc159021286)

[4.7 Prijavljivanje na kurs i pretraga 31](#_Toc159021287)

[4.8 Kreiranje testova 33](#_Toc159021288)

[4.9 Pregled lekcija 36](#_Toc159021289)

[4.10 Izrada testova 37](#_Toc159021290)

[4.11 Pregled testiranja 41](#_Toc159021291)

[4.11.1 Tabela svih podataka 42](#_Toc159021292)

[4.11.2 Prikaz ocena po studentima 43](#_Toc159021293)

[4.11.3 Prikaz po lekciji 45](#_Toc159021294)

[5. Zaključak 46](#_Toc159021295)

[6. Literatura 47](#_Toc159021296)

# Uvod

U današnjem svetu sve je veća potreba pojedinca da se usavršava u što većem broju oblasti poput usavršavanja jezika, učenja novih veština, napredovanja u karijeri. Samim tim, aplikacije koje pojedincu mogu da pomognu da se usavršava naglo su počele sa razvojem u poslednjem periodu.

Nažalost, mnogobrojne *web* aplikacije koje imaju ovu namenu postoje, međutim, ne pružaju pristup otvorenog koda i nisu sklone prilagođavanju korisnika. Zbog toga ću u svom radu predstaviti aplikaciju za učenje na daljinu, otvorenog koda, korišćenjem modernih alata za kreiranje *web* aplikacija, Java *Spring* i *Angular*.

# Tehnologije korišćene u radu

## Bekend tehnologije

## 2.1.1 Java

Java je programski jezik baziran na objektno orijentisanom programiranju [1]. Nastao je kao projekat firme *Sun Microsystems* ranih devedesitih godina prošlog veka sa ciljem da se programski jezik pokreće na kućnim aparatima. Sintaksa Jave je upravo zbog toga slična sintaksi jezika *C++,* kako bi sistemski i aplikativni programeri lakše naučili jezik visokog nivoa.

Java se smatra jezikom visokog nivoa zato što programerima otklanja brigu o kontrolisanju memorije, već sama vodi računa o istoj. Uklanjanje nepotrebnih objekata iz memorije obavlja *garbage collector*, program koji radi nezavisno od naših aplikacija i samostalno odlučuje koje objekte će ukloniti iz memorije [2].

Java okruženje u kom se program izvršava, dizajnirano je tako da što manje zavisi od same arhitekture računarskog sistema. Java program se izvršava na Java virtuelnoj mašini, mašini koja postoji samo u memoriji [3]. Jednom napisan Java kod se uz pomoć Java kompajlera prevodi u *bytecode*, izvorni kod koji koriste Java virtuelne mašine kako bi ga dešifrovale u izvorni kod, koji je pogodan za izvršavanje na računaru. Java kod i *bytecode* ostaju isti bez obzira na kojoj platformi se izvršava program.

Sa obzirom da u Javi važi da se jednom napisan program može pokrenuti na bilo kojoj kompjuterskoj arhitekturi, *web browseri* su brzo uveli mogućnost pokretanja Java *Appleta* na *web* stranicama koji su se pokretali van *browsera* i Java je postala jako popularna u *web* industriji. Kasnije, Java *Appleti* su se pokazali nebezbednim na internetu sa glavnim razlogom da su imali potpun pristup računaru klijenta, pa je Java prestala da se koristi kao klijentska strana, već je svoju namenu i razvoj okrenula ka serverskoj strani.

Java koristi objektno orijentisanu paradigmu programiranja koja predstavlja tehniku programiranja fokusiranu na podatke i na interfejse ka tim objektima. Smatra se preglednijim i lakšim za razvojem zbog načina na koji može opisati podatke, kao što je prikazano u okviru listinga 1.

public class Covek {  
 private String ime, prezime;  
  
 void spavaj() {  
 *// logika metode* }  
}

Listing 1 Objektno orijentisana tehnika programiranja

### Spring framework

*Spring* jeste Java *framework* koja obezbeđuje infrastrukturu za razvoj aplikacije [3]. Prva verzija je objavljena 2003. godine. Popularnost je stekao zbog svoje lakoće razvijanja aplikacija čistog koda čime Java aplikacije čini lakim za održavanje i pospešuje jednostavnost razvoja velikih aplikacija. Organizovan je u dvadeset modula koji mogu da rade nezavisno jedan od drugog i otvorenog je koda. Neke od funkcionalnosti ovih modula su bezbednost, povezivanje sa bazama podataka, slanje poruka između sistema, testiranje, *model-view-controller* pristup programiranju i drugi.

*Spring* kontejner jeste glavna komponenta *Spring* *frameworka*. Odgovoran je za upravljanje komponentama i objektima unutar aplikacije. Kontejner kreira, konfiguriše i upravlja ovim objektima. Ključni koncepti *Springa* jesu inverzija kontrole i automatsko razrešavanje zavisnosti komponenti.

Inverzija kontrole jeste princip prema kojem kontrola nad tokom izvršavanja aplikacije nije u potpunosti u rukama samih komponenti, već se prebacuje na *Spring* kontejner. Umesto da komponente same kreiraju svoje zavisnosti ili objekte sa kojima rade, inverzija kontrole prebacuje ovu odgovornost na *Spring*, što rezultira manjom zavisnošću između komponenti.

*Dependency Injection* (DI) je tehnika koju *Spring* koristi kako bi implementirao inverziju kontrole. To znači da umesto da komponente same traže zavisnosti, ove zavisnosti se ubrizgavaju u komponentu uz pomoć tehnike u programiranju koja se zove refleksija. Na primer, ako imate servis koji zahteva pristup bazi podataka, umesto da servis sam kreira objekat za rad sa bazom, taj objekat će se ubrizgati u servis kroz konstruktor ili metodu, kao što je prikazano na listingu 2. Ovo olakšava testiranje, jer se zavisnosti mogu lako zameniti lažnim ili simuliranim objektima tokom testiranja.

public class Service {  
 @Autowired  
 private Datasource bazaPodataka;  
}

Listing 2 Injektovanje objekta uz pomoć polja

***Bean*** jeste osnovna komponenta unutar *Spring* aplikacije kojim upravlja kontejner. Upravo ove komponente mogu predstavljati različite delove aplikacije poput servisa, repozitorijuma, kontrolera i drugih komponenti.

***Controller*** je komponenta *Springa* koja se koristi u *web* aplikacijama. Nalazi se u okviru modula MVC(*model-view-controller*). Odgovaran je za obradu *HTTP* zahteva, komunikaciju sa klijentom i upravljanje tokom izvršavanja aplikacije. Njihov zadatak je da primaju zahteve, obrade zahtev i da vrate smislen odgovor u skladu sa zahtevom. Koriste se zajedno sa servisima koji se ubrizguju unutar *Controller-a*. Na listingu 3 prikazan je primer kontrolera koji koristi inverziju kontrole, obrađuje zahtev i vraća odgovor klijentu.

@RestController  
public class TestController {  
 @Autowired  
 private TestService testServis;  
  
 @GetMapping("/test/{id}")  
 public ResponseEntity<Test> vratiTest(@PathVariable Long id){  
 return new ResponseEntity<>(testServis.getTest(id), HttpStatus.*OK*);  
 }  
}

Listing 3 Primer kontrolera, injektovanje servisa I vraćanje odgovora

**Servisi** su takođe komponente *Spring* aplikacije i služe za obavljanje logičkih operacija unutar programa. Sva kompleksna logika i zahtevi se izvršavaju ovde. U komunikaciji sa repozitorijumima vraćaju odgovor koji *Controller* vraća klijentu. Primer servisa prikazan je na listingu 4.

@Service  
public class TestService {  
 @Autowired  
 private TestRepository testRepozitorijum;  
  
 public Test vratiTest(Long id){  
 Optional<Test> test = testRepozitorijum.getTest(id);  
  
 if(test.isPresent())  
 return test.get();  
 throw new TestDoesNotExist("Test ne postoji!");  
 }  
}

Listing 4 Primer servisa koji komunicira sa repozitorijumom i vraća odgovarajuću poruku

**Repozitorijumi** su komponente koje se koriste za pristupanje i manipulisanje podacima u bazama podataka. Oni pružaju sloj apstrakcije između aplikacije i baze podataka i omogućavaju osnovne ***CRUD*** [4] operacije nad podacima bez upotrebe *SQL* upita. Uglavnom se koriste uz ***Spring Data JPA*** tehnologiju kako bi pristupili i radili sa podacima, kao što je prikazano na listingu 5.

@Repository  
public interface TestRepository extends JpaRepository<Test, Long> {  
}

Listing 5 Primer repozitorijuma nad Test entitetom koji ima polje tipa Long kao identifikator

***Spring Data JPA*** jedan je od modula *Springa* koji pruža podršku za rad sa JPA (*Java Persistence API*) tehnologijom. JPA je Java standard za mapiranje objektno-relacionog modela (ORM), što omogućava programerima da rade sa bazama podataka koristeći Java objekte i modele umesto *SQL* upita.

### Maven

***Maven*** je alat za izgradnju projekata, upravljanje zavisnostima u projektima i automatizaciju procesa razvoja softvera. Primarno se koristi u Java projektima, ali se takođe koristi i u drugim programskim jezicima poput *C#, Ruby, Scala* i drugim.

Sva *Maven* konfiguracija definisana je u fajlu XML formata, tzv. ***pom.xml*** (Project Object Model). U ovom fajlu opisano je ime projekta, njegov kreator, kako će se projekat pokrenuti, njegove zavisnosti i njihove verzije i još mnogo toga. *Maven* dinamički preuzima sve Java biblioteke navedene u konfiguracije sa *Maven* centralnog repozitorijuma i kešira ih u lokalnoj memoriji. Primer *pom.xml* fajla prikazan je na listingu 6.

*Maven* omogućava automatsko izvršavanje raznih faza procesa izgradnje projekta kao što su kompilacija koda, pakovanje artefakata, testiranje, generisanje dokumentacije i sl.

*<?*xml version="1.0" encoding="UTF-8"*?>*<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.example</groupId>  
 <artifactId>your-project-name</artifactId>  
 <version>1.0.0</version>  
</project>

Listing 6 Primer pom.xml fajla

### PostgreSQL

***PostgreSQL*** jeste sistem za upravljanje relacionim bazama podataka [5]. Besplatan je i otvorenog je koda. Prva verzija objavljena je 1996. godine. Jedan je od najnaprednijih i najpopularnijih sistema koji se bave relacionim bazama zbog mogućnosti proširivanja sistema i brojnim funkcionalnostima. Podržava rad sa raznim strukturama podataka poput *JSON-a* [6], *XML-a*, geografskih podataka, relacionih i drugih.

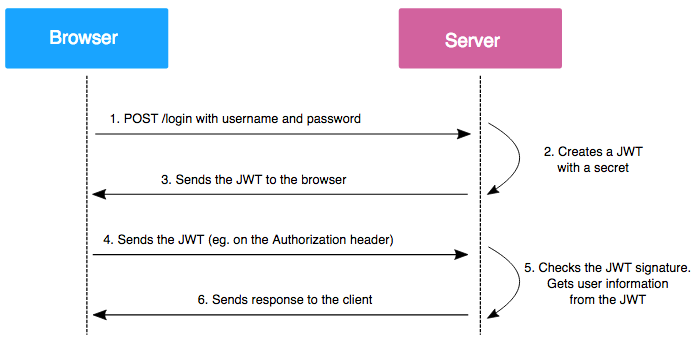
Transakcije u okviru *Postgresa* podržavaju *ACID [7]* principe, automatsko osvežavanje pregleda, okidače, strane ključeve i druge. *Postgres* sistem je dizajniran tako da podržava različite vrste poslova, od onih koji se odigravaju na jednoj mašini, do velikih skladišta podataka ili *web* servissistema sa mnogo konkurentnih korisnika.

### Lombok

*Lombok* ili *Project Lombok* jeste Java biblioteka koja omogućava pisanje čistijeg koda tako što u pozadini kreira potrebne metode bez pisanja eksplicitnog koda. *Lombok* automatski generiše metode na osnovu anotacije, čime se smanjuje količina koda koja se ponavlja i pruža lakšu skalabilnost i čitljivost aplikacije. Neke od anotacija koje se mogu dodati nad Java poljima, metodama i klasama jesu: *Getter, Setter, AllArgsConstructor, ToString, NonNull, Data, Builder* i druge.

### JSON Web Token

*JSON Web Token* (*JWT*) je internet standard za kreiranje podataka sa potpisom čiji sadržaj nosi u sebi *JSON* zapis koji sadrži informacije u obliku “ključ: vrednost”. Tokeni se sastoje od tri dela: poglavlja, sadržaja i potpisa. Tokeni se potpisuju privatnim ključem i samo sistem koja sadrži privatni ključ može da validira token. Na primer, server može da generiše token koji u sebi kaže „Prijavljen kao administrator“, zatim da ga potpiše sa privatnim ključem za koji samo on zna i da preda taj token klijentu. Klijent će taj token skladištiti i slati prilikom svakog HTTP zahteva. Kada server primi poruku od klijenta, on dekodira sam potpis uz pomoć svog privatnog ključa i ako se potpis poklapa, znaće da je u pitanju validan token.



Slika 1 Tok razmena poruka sa JSON Web Tokenom

## Frontend tehnologije

## 2.2.1 TypeScript

*TypeScript* jeste programski jezik otvorenog koda razvijen od strane Majkrosofta [8]. Osmišljen je kao proširenje *JavaScript* jezika zbog izazova koji postoje prilikom razvoja *JavaScript* aplikacija. *TypeScript* je moguće pisati i u projektima koji koriste *JavaScript* kod zato što se *TypeScript* kompajlira u *JavaScript* i time olakšava postepeno uvođenje *TypeScripta* u *JavaScript* projekat.

U *JavaScriptu* tipovi promenljivih se mogu dinamični menjati u toku rada programa i to dovodi do raznih problema poput grešaka koje se pojavljuju tek tokom izvršavanja programa, slabo čitljivog koda, povećane verovatnoće za pisanjem grešaka, performanse i druge.

*TypeScript* je uveo dodatne mogućnosti nad *JavaScriptom* kao što su provera tipova za vreme kompilacije, klase, interfejsi, brojački tipovi, generici, modularno programiranje, tuplove i druge.

Najveća prednost *TypeScripta* nad *JavaScriptom* jeste upravo korišćenje tipova promenljivih, odakle i potiče sam naziv jezika. Sa obzirom da u *JavaScriptu* tipove za promenljive ne postoje, oni mogu biti promenjeni u toku izvršavanja programa. Uvođenjem tipova *TypeScript* obavlja proveru prilikom kompajliranja koda i time se sve greške vezane za tipove promenljivih prikazuju nakon kompajliranja, tj. ne prikazuju se u krajnjem *JavaScript* kodu. Tip se određuje tako što se posle naziva promenljive doda dvotačka i naziv tipa (*number, string, boolean, BigInteger*..). Takođe postoji i *any* tip podataka koji je dodat u *TypeScript* kako bi omogućio kompatibilnost sa *JavaScript* kodom i podrazumeva da promenljiva može imati bilo koji tip. Takođe, promenljivoj se može dodeliti i mogućnost da koristi dva ili više tipova podataka korišćenjem prave crte (npr. *string | boolean*). Ova funkcionalnost nije preporučljiva zajedno sa *any* tipom zbog potencijalnih grešaka tokom rada programa. Ukoliko se promenljiva inicijalizuje sa jednim tipom podatka, a zatim joj se u nastavku dodeli podatak sa drugim tipom, *TypeScript* kompajler će baciti grešku i kod neće biti iskompajliran u *JavaScript* kod, kao što je prikazano na listingu 7.

// validan kod u JavaScriptu

var boje = ['Crvena', 'Plava', 'Zelena'];

boje = 3;

// TypeScript

let boje: string[] = ['Crvena', 'Plava', 'Zelena'];

// baciće grešku prilikom kompajliranja, niz nije tipa string[]

boje = 3;

Listing 7 Razlika između JavaScript i TypeScript koda

### Angular

***Angular*** je framework za izradu web aplikacija na jednoj *web* stranici, tzv. *Single Page Applications* (SPA). Aplikacije u *Angularu* se najčešće razvijaju u *TypeScript* jeziku i sama platforma uvodi neke nove koncepte poput automatskog razrešavanja zavisnosti komponenti, ubrizgivanje zavisnosti, povezivanje podataka, rutiranje, testiranje aplikacije i druge. Prva verzija *Angulara* izdata je 2010. godine i nju je razvio tim iz *Googla [9]*. Obzirom na to da je *framework* otvorenog koda, svi programeri mogu da kontribuiraju razvoju platforme.

*Angular* je dizajniran tako da prilagođava i proširuje tradicionalni *HTML* kako bi predstavio dinamički sadržaj kroz dvosmerno vezivanje podataka koje omogućava automatsku sinhronizaciju modela i korisničkog interfejsa u *Model-view-controller* arhitekturi.

**Komponente** u *Angularu* su gradivni elementi aplikacije koji imaju svoja polja, metode i mogu da prikažu odgovarajući deo korisničkog interfejsa. Koriste se za organizaciju koda i podele funkcionalnosti aplikacije na manje delove, što olakšava održavanje i ponovnu upotrebu. Svaka komponenta se sastoji od prezentacije (*HTML* i *CSS* deo koji definiše kako će komponenta biti prikazana) i kontrolera (*TypeScript* klasa koja definiše logiku komponente, sadrži metode i polja koji se koriste unutar *HTML* komponente).

***Two-way data binding*** (dvosmerno povezivanje podataka) jeste jedna od glavnih karakteristika *Angulara* koja omogućava sinhronizaciju podataka između kontrolera i korisničkog interfejsa. Ovo omogućava da se promene u modelu automatski reflektuju u prikazu i obrnuto, što olakšava razvoj interaktivnih aplikacija. Postiže se kroz ***ngModel*** direktivu. Na listingu 8 prikazan je *HTML* element koji dinamički ispisuje vrednost polja *naslovProjekta* iz skript fajla.

<input [(ngModel)]='naslovProjekta'>

Listing 8 Prikazivanje podataka iz kontrolera u okviru korisničkog interfejsa

**Direktive** u *Angularu* imaju funkcionalnost da menjaju izgled ili ponašanje postojećih *HTML* elemenata dodavanjem, brisanjem ili modifikacijom atributa elemenata. Neke od direktiva koje se najčešće koriste jesu ***ngFor***, koja služi za iteriranje kroz kolekciju podataka iz modela i **ngIf,** koja prikazuje *HTML* element samo ukoliko je određeni uslov ispunjen. Na listingu 9 je prikazano korišćenje direktiva u realnom primeru.

<h1>Studenti koji su polozili predmet</h1>

<div \*ngFor="let student of studenti">

    <div \*ngIf="student.ocena >= 6">

        {{ student.ime }}

    </div>

</div>

Listing 9 Realna primena direktia u projektu

***Routing*** (rutiranje) jeste *Angular* modul koji se odnosi na navigaciju prikaza komponenata unutar aplikacije. Obzirom na to da *Angular* služi za kreiranje SPA *web* stranica, korisnik će za izmenu korisničkog interfejsa prikazivati i skrivati korisnički interfejs postojećih komponenti pre nego što će zatražiti novu stranicu od strane servera, kako je slučaj kod *web* sajtova sa više stranica. Svaka komponenta sadrži putanju (*route*) koja se prikazuje adekvatnom navigacijom.

Kako bi se u *Angularu* postiglo rutiranje, potrebno je da se konfigurišu rute unutar ***app-routing.module.ts*** fajla pomoću ***RouterModule.forRoot()*** metode. U ovoj metodi se navode putanje i komponente koje će biti prikazane kada se rutira na određenu putanju. U okviru listinga 10 imamo primer konfiguracije rutiranja koja pokazuje da ukoliko unesemo putanju *login*, prikazuje nam se korisnički interfejs *LoginComponent*, dok ako je putanja prazna, prikazuje nam se *HomeComponent*.

const routes: Routes = [

  { path: '', component: HomeComponent },

  { path: 'login', component: LoginComponent }

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

Listing 10 Konfiguracija rutiranja u app-routing.module.ts fajlu

U okviru *HTML-a* ***router-outlet*** se koristi za prikazivanje komponenti na osnovu trenutne putanje i može se koristiti u bilo kojoj komponenti (listing 11). Kada korisnik u okviru aplikacije menja putanju, *Angular* dinamički učitava odgovarajuće komponente unutar *router-outlet* taga i tako omogućava interaktivnu navigaciju bez potrebe za ponovnim učitavanjem cele stranice.

**<router-outlet></router-outlet>**

Listing 11 Rutiranje u okviru HTML fajla

**Servisi** u *Angularu* jesu klase koje se koriste za deljenje funkcionalnosti i podataka između različitih delova aplikacije. Oni omogućavaju centralizovanu logiku u okviru jedne funkcionalnosti aplikacije koja se može koristiti u više delova aplikacije, kao što su komponente ili drugi servisi. Dekorišu se sa ***@Injectable***, a nakon toga se instanca doda u globalnu listu servisa aplikacije tako što se klasa registruje u *providers* listi nekog modula ili komponente. Listu registrovanih servisa njihovih instanci čuva deo *Angular* aplikacije koji se zove *injector*.

***HttpClient*** je deo *Angular* *framework-a* koji omogućava komunikaciju sa *HTTP* serverima putem *AJAX* zahteva. Slanjem *HTTP* zahteva prema serveru omogućava dobijanje potrebnih podataka ili slanje podataka ka serveru, u zavisnosti od zahteva.

@Injectable({

  providedIn: 'root'

})

export class TestService {

  constructor(private http: HttpClient) { }

  getTests(): Observable<any> {

    return this.http.get('https://mojServer.com/testovi');

  }

}

Listing 12 Servis koji skladišti podatke I komunicira sa serverom

***Observable*** je deo *RxJS* biblioteke koja se koristi za asinhrono programiranje [10]. Predstavlja tok vrednosti koje mogu biti primljene tokom vremena i omogućava asinhronu obradu tih vrednosti. Prilikom korišćenja *HttpClienta* u *Angularu* za slanje *HTTP* zahteva, kao u listingu 12. Rezultat zahteva je *Observable* objekat. Ovo omogućava obradu asinhrono vraćenih podataka ili grešaka koje mogu nastati prilikom *HTTP* komunikacije sa serverom. Vrsta poruke koja može stići u zavisnosti od zahteva jesu *next*, *error* i *complete*.

### Angular CLI

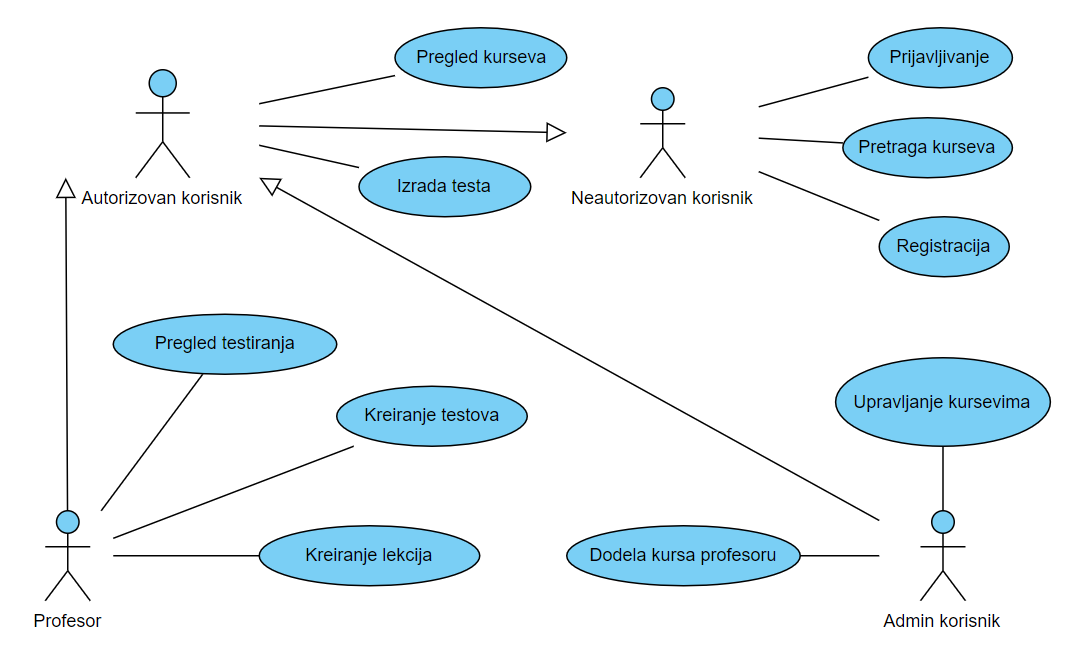
*Angular CLI* (*Command-Line Interface*) je alat koji programerima olakšava razvoj Angular aplikacija. Funkcije koje izvršava jesu generisanje projekata i komponenata, pokretanje lokalnog servera za razvoj, kompajliranje *TypeScript* koda u *JavaScript*, testiranje aplikacija, upravljanje zavisnostima i konfiguracijama i druge. Prilikom razvoja novih komponenti i servisa, alat automatski kreira nove direktorijume i imenuje komponente na način koji je lako čitljiv, čime doprinosi lakšem razvoju većih aplikacija. Moguće je kreirati osnovu za nove module, komponente, servise i ostale delove *Angular* aplikacije, generisati dokumentaciju i još mnogo toga.

# Specifikacije aplikacije

Koristeći aplikaciju, korisnik treba da ima mogućnost da se prijavi sa svojim nalogom ili ukoliko nema svoj nalog treba mu omogućiti da se registruje. Kada se korisnik uspešno prijavi, otvoriće mu se prozor sa postojećim kursevima u sistemu, kao i prozor koji omogućava da izmeni kurseve ukoliko je on predavač na nekom od njih. Nakon odabira kursa kojeg želi da sluša otvara se prozor kursa koji sadrži sve lekcije kursa i moći će da radi testove u okviru lekcija ukoliko je test trenutno u toku.

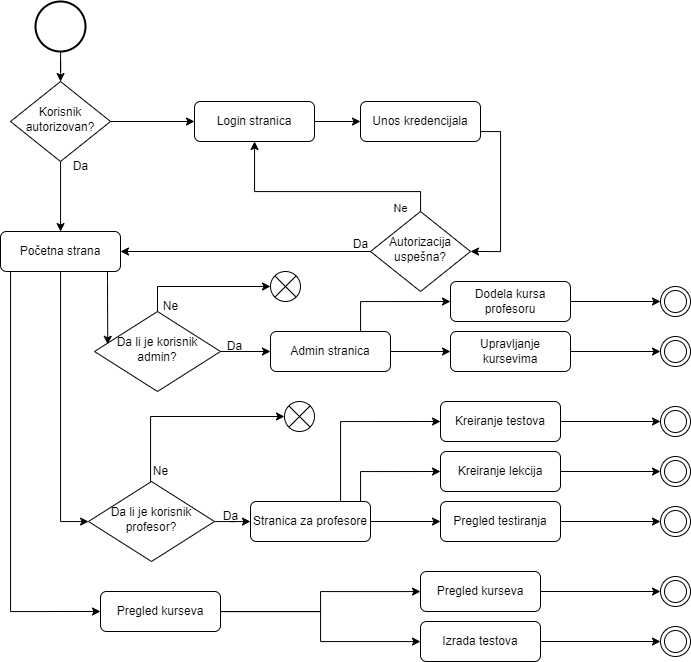
## Dijagram slučajeva korišćenja

Neautorizovan korisnik će videti stranicu koja prikazuje sve kurseve u sistemu i dugme za registraciju i prijavljivanje u sistem. Nakon što korisnik kreira nalog, moći će da se prijavi i pregleda sve kurseve na koje je prijavljen, izmeni kurseve na kojima je predavač, radi testove. Pored osnovnih funkcionalnosti postoje i druge poput pretrage kurseva, detaljan pregled ocena studenata, kreiranje testova i druge. Dijagram slučajeva korišćenja aplikacije, gde su prikazane sve akcije koje korisnik može da izvrši se mogu videti na slici 2.



Slika 2 Dijagram slučajeva korišćenja aplikacije

## 3.2. Dijagram aktivnosti

Prilikom otvaranja aplikacije, proverava se da li je korisnik ulogovan i ukoliko jeste prebacuje se na početnu stranu sa kursevima, dok ukoliko nije, prebacuje se na prozor za prijavljivanje. Nakon uspešne prijave, interfejs prikazuje listu kurseva i dugme za pristup istima, dajući korisniku mogućnost da bira koji kurs želi da sluša, dok ukoliko korisnik ima admin ulogu prikazaće mu se i mogućnost odlaska na admin stranicu, isto važi i za stranicu za predavače ukoliko je korisnik predavač. Nakon što korisnik odabere koju vrstu akcije želi da ostvari otvara mu se mogućnost da izvrši istu, ukoliko ima tu mogućnost. Dijagram aktivnosti aplikacije je predstavljen na slici 3.

Slika 3 Dijagram aktivnosti aplikacije

# Opis implementacije aplikacije

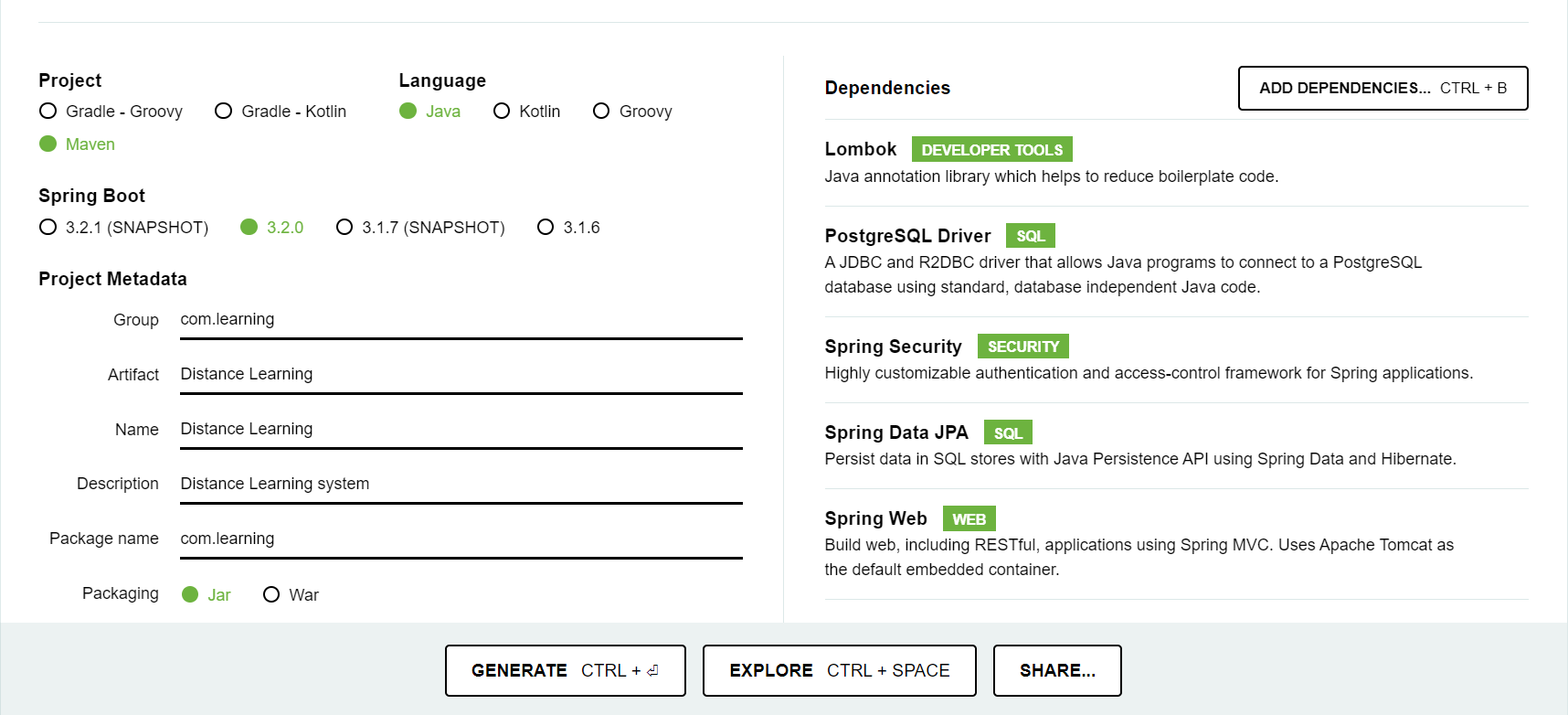
Aplikacija je napisana korišćenjem sledećih alata:

* IntelliJ IDEA - *Integrated Development Environment* (*IDE*) za razvoj bekend aplikacija [11],
* Visual Studio Code – *IDE* za razvoj frontend aplikacija i
* *Angular* *CLI* - za inicijalizaciju, razvoj, kompajliranje i serviranje *Angular* aplikacija.

## Inicijalizacija aplikacije

4.1.1 Backend inicijalizacija

Kako bismo započeli razvoj aplikacije na bekendu potrebno je da generišemo projekat. *Spring* *framework* je poznat po tome što je njegove projekte lako generisati, sve što je potrebno da se uradi jeste da se na *start.spring.io* *web* stranici konfigurišu alati za razvoj projekta poput *dependency* menadžera, programski jezik, verzija *Spring* projekta, verzija jezika i biblioteke koje želimo da koristimo u projektu i *Spring* će nam automatski generisati sam projekat u zip formatu.



Slika 4 Generisanje projekta uz pomoć start.spring.io web sajta

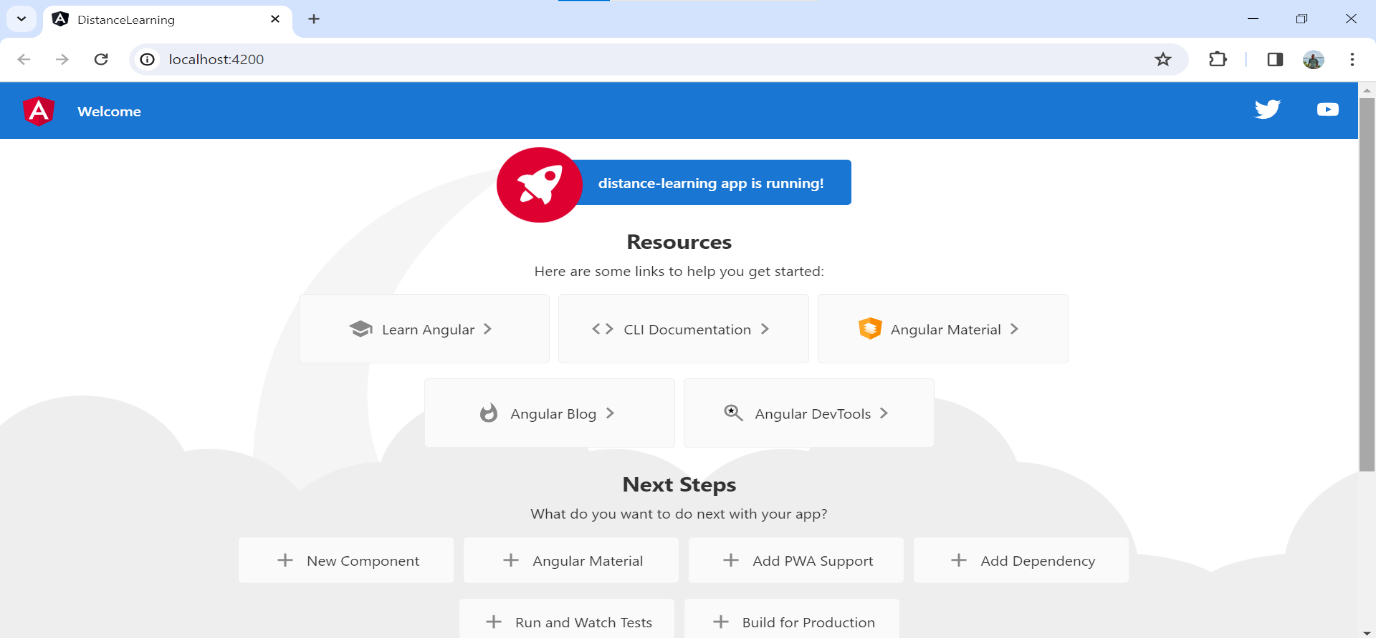
Nakon klika na ***Generate*** dugme, preuzimamo direktorijum ***Distance Learning*** koji sadrži sve fajlove potrebne za razvijanje. Kada otvorimo *IntelliJ* okruženje i izaberemo naš projekat okruženje će samo prepoznati koja verzija Jave nam je potrebna i Maven će sam pokrenuti proces preuzimanja biblioteka sa interneta. Nakon preuzimanja svih bibllioteka i instalacije Java verzije okruženje će biti spremno za pokretanje projekta.

Sledeće što nam je potrebno jeste kreiranje baze i povezivanje sa istom. Za kreiranje baze koristićemo ***PostgreSQL*** alat uz pomoć komandne linije. Sve što je potrebno jeste da se prijavimo sa *postgres* nalogom uz pomoć komande ***psql -U postgres*** i kreiramo bazu komandom ***CREATE DATABASE learning***. Komandom ***\l*** proveravamo sve postojeće baze i tu će se pojaviti i naša *learning* baza. Nakon kreiranja, u ***application.properties*** fajlu našeg *Spring* projekta unosimo putanju do naše baze, kredencijale i verziju drajvera *PostgreSQL*, kao što je prikazano na listingu 13. Nakon podešavanja okruženja i pokretanja programa, isti će osluškivati sve pozive ka adresi *http://localhost:8080* i biće povezan sa bazom podataka preko porta 5432.

*# Database Configuration*spring.datasource.url=jdbc:postgresql://localhost:5432/learning  
spring.datasource.username=postgres  
spring.datasource.password=root  
spring.datasource.driver-class-name=org.postgresql.Driver  
spring.server.port=8080

Listing 13 Konfiguracija baze podataka u application.properties fajlu

4.1.2 Frontend inicijalizacija

Za inicijalizaciju aplikacije koristi se *Angular CLI* komanda ***ng new*** iza koje sledi naziv projekta, zatim se u direktorijumu projekta prvo instaliraju biblioteke ***npm install*** komandom i sa ***ng serve*** komandom projekat se kompajlira i pokreće na portu 4200 koji će servirati podatke iz projekta. Nakon ovih komandi u *browseru* na adresi [*htps://localhost:4200*](https://localhost:4200) će se pojaviti osnovna *Angular* aplikacija prikazana na slici 6. Postupak pokretanje aplikacije prikazan je na listingu 14.

***ng new*** *distance-learning-frontend*

***cd distance-learning-frontend***

***npm install***

***ng serve***

Listing 14 Kreiranje i pokretanje aplikacija uz pomoć Angular CLI

Slika 5 Osnovna Angular aplikacjia u browseru

Koristeći ***npm*** komandu ***install***, instaliraćemo dodatne biblioteke koje će se koristiti u aplikaciji:

* *rx-js* – biblioteka koja omogućava asinhrone pozive koja omogućava lakšu komunikaciju sa bekendsistemom,
* *ngx-material-timepicker* – biblioteka koja omogućava korišćenje interfejsa za odabir datuma i vremena i
* *chart.js* – biblioteka koja omogućava korišćenje komponenata i alata za pregledanje podataka pomoću dijagrama.

Ako pogledamo *index.html* (listing 15) našeg frontend projekta videćemo da je to jednostavan ***HTML5***dokument koji sadrži dva taga koja ne postoje u *HTML5* specifikaciji: ***<base>*** i ***<app-root>***. Ova dva taga su deo *Angular* platforme. ***<base>***kazuje ruteru koja mu je osnovna putanja da nad njom nadograđuje putanje aplikacije. ***<app-root>*** označava mesto gde će se *Angular* aplikacija učitati na stranici.

<!doctype html>  
<**html** lang=**"en"**>  
  <**head**>  
    <**meta** charset=**"utf-8"**>  
    <**title**>DistanceLearning</**title**>  
    <**base** href=**"/"**>  
    <**meta** name=**"viewport"** content=**"width=device-width, initial-scale=1"**>  
    <**link** rel=**"icon"** type=**"image/x-icon"** href=**"favicon.ico"**>  
  </**head**>  
  <**body**>  
    <**app-root**></**app-root**>  
  </**body**>  
</**html**>

Listing 15 Prikaz **index.html** dokumenta koji sadrži i učitava Angular aplikaciju

Glavni modul aplikacije je ***AppModule*** (listing 16) i nalazi se u *src/app* direktorijumu. U modul se učitava ***BrowserModule*** koji inicijalizuje aplikaciju da se prikazuje u *browser*-u, zatim se deklariše glavna komponenta, ***AppComponent***, koja sadrži šablon za glavni prikaz aplikacije.

@NgModule({  
 declarations: [ AppComponent ],  
 imports: [ BrowserModule ],  
 providers: [],  
 bootstrap: [ AppComponent ]  
})  
**export** **class** AppModule {  
}

Listing 16 AppModule – početni modul Angular aplikacije

Da bi se izbeglo ponovno učitavanje aplikacije nakon svake promene osnovnog prikaza stranice, *Angular* u sebi sadrži mehanizam za promenu *URL* adrese i prikaza bez ponovnog učitavanja cele *HTML* stranice. Taj mehanizam se naziva rutiranje (*router*) i nalazi se u biblioteci ***@angular/router*** od *Angular* verzije 4. Ruter može koristiti ili *hash* *URL* notaciju (na kraju glavne *URL* adrese doda simbol *#*, a zatim iza toga gradi svoju *URL* strukturu) ili *History API* [12] podržan u *HTML5* standardu.

Da bi se ovo izvelo, u modulu ***AppRoutingModule*** (listing 17), koji ćemo dodati u glavni ***AppModule***, inicijalizovaćemo niz ruta i dodati instancu ***Router*** modula u njemu. *Angular* dozvoljava da se dodavanje modula u ***imports*** konfiguracioni niz modula obavi uz prosleđivanje konfiguracionih parametara za taj modul. Statičkoj metodi ***forRoot()*** modula ***Router*** prosleđujemo niz tipa ***Routes***, kojim definišemo rute aplikacije, gde definišemo koje se komponente aplikacije mapiraju na određene putanje. Na kraju ćemo instancu ***RouterModule*** modula dodati u ***exports*** konfiguracioni niz kako bi glavni modul mogao da ih iskoristi.

**const** routes: Routes = [  
 { **path**: **'login'**, **component**: LoginComponent },  
 { **path**: **'admin'**, **component**: AdminComponent },

{ **path**: **'home'**, **component**: HomeComponent },  
   
 *// otherwise redirect to home* { **path**: **'\*\*'**, **redirectTo**: **'home'** }  
];  
  
@NgModule({  
 **exports**: [ RouterModule ],  
 **imports**: [ RouterModule.*forRoot*(routes) ]  
})  
**export class** AppRoutingModule {  
}

Listing 17 Primer definisanja ruta

Ukoliko se korisnik nađe na */login* adresi prikazaće mu se ***Login*** komponenta koja prikazuje dugme za autorizaciju korisnika. Sve ostale adrese će se prebaciti na adresu */home* čija komponenta sadrži glavni prikaz aplikacije.

Da bi ruter znao gde će učitavati komponente koje mu se prosleđuju, za svaku navedenu adresu potrebno je da u glavnoj komponenti modula u kome se učitava ruter, u ovom slučaju je to ***AppComponent***, dodati ***<router-outlet>*** tag. Ovaj tag će biti zamenjen tagom komponente koja se učitava. Sve što je potrebno da se doda u *HTML* fajl kako bi komponenta učitala druge komponente iz rutera prikazano je na listingu 18.

<**router-outlet**></**router-outlet**>

Listing 18 Prikaz komponente koja je odabrana rutiranjem

## Korisnici i uloge

Implementacija korisnika i uloga u našem sistemu je bitna stavka kako bismo identifikovali različite korisnike i kako bismo određenim korisnicima ograničili funkcionalnosti aplikacije gde je to potrebno. Svaki korisnik će imati mogućnost da sluša kurseve, testira u okviru lekcija i dobija ocene za njih i potrebno mu je omogućiti prijavu u sistem sa svojim kredencijalima. Sa druge strane, ne želimo svim korisnicima da omogućimo funkcionalnost izmene i kreiranja kurseva i druge funkcionalnosti, pa je potrebno da implementiramo i uloge u sistemu.

Naš sistem će podržavati dve uloge, regularnog korisnika i admin korisnika. Ove dve uloge možemo predstaviti uz pomoć ***ERole*** *enuma* koji će sadržati dve vrednosti, *ROLE\_REGULAR* i *ROLE\_ADMIN*. **Enum** je specijalni tip podataka u Javi koji se koristi za definisanje kolekcije konstantnih vrednosti. Kako bismo uloge sačuvali u bazi potrebno je da kreiramo klasu ***Role*** i označimo je sa ***Entity*** anotacijom pomoću koje će se kreirati tabela *Role* u našoj bazi koja će za kolone imati polja ove klase. Ova klasa će sadržati samo *enum* kao polje koji je takođe i primarni ključ tabele, stoga ga označavamo sa ***Id*** anotacijom i označićemo ga sa ***Enumerated*** anotacijom, jer želimo da nam se u bazi uloga čuva u tekstualnom formatu, a ne pomoću automatsko generisanih identifikatora, kao što je prikazano na listingu 19.

@Entity  
public class Role {  
  
 @Id  
 @Enumerated(EnumType.*STRING*)  
 private ERole role;  
}

Listing 19 Entitet role koja se čuva u bazi

Podaci koji su nam potrebni za korisnika su njegova imejl adresa, koja će služiti kao primarni ključ tabele, lozinka, njegovo ime i prezime, kao i njegova uloga. Kako bismo korisnike mogli da sačuvamo u bazi, potrebno je da klasu anotiramo sa ***Entity***. Ulogu ćemo anotirati sa ***ManyToOne*** kako bismo naznačili da više korisnika mogu biti povezani ka jednoj ulozi u bazi. Ova anotacija će kolonu *role* u našoj *User* tabeli označiti kao strani ključ ka tabeli *Role*, čime se održava integritet podataka, garantujući da će *User* biti povezan sa ulogom koja je validna. Naš korisnik će implementirati *Spring* interfejs ***UserDetails*** koji sadrži metodu ***getAuthorities*** koja će vraćati ulogu korisnika i on će nam kasnije pomoći da automatski autentifikujemo korisnika. Na listingu 20 je prikazana implementacija modela korisnika u Javi.

@Entity  
public class AppUser implements UserDetails {  
 @Id  
 private String email;  
 private String firstName;  
 private String lastName;  
 private String password;  
  
 @ManyToOne  
 private Role role;  
  
 @Override  
 public Collection<? extends GrantedAuthority> getAuthorities() {  
 SimpleGrantedAuthority authority = new SimpleGrantedAuthority(role.getRole().name());  
 return Collections.*singletonList*(authority);  
 }  
}

Listing 20 Model korisnika u Java aplikaciji

Nakon kreiranja entiteta, potrebno je da kreiramo ***AppUserRepository*** i ***RoleRepository*** interfejse koji nasleđuju *JpaRepository* interfejs i anotiramo ih sa ***Repository*** anotacijom kako bismo omogućili osnovne *CRUD* funkcionalnosti u bazi nad *User* i *Role* entitetima.

Nakon repozitorijuma kreiraćemo i servis ***AppUserService*** koji će obavljati svu logiku našeg korisnika. Za početak, anotiraćemo ga sa ***Service*** i implementirati *Spring* interfejs ***UserDetailsService*** i njegovu metodu ***loadUserByUsername*** koja će pronaći korisnika preko imejl adrese i kasnije nam pomoći oko autorizacije korisnika (listing 21).

@Service  
@AllArgsConstructor  
public class AppUserService implements UserDetailsService {  
  
 private final AppUserRepository appUserRepository;  
  
 @Override  
 public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {  
 return appUserRepository.findById(email).orElseThrow(() -> new UsernameNotFoundException(email));  
 }  
}

Listing 21 Implementacija AppUser servisa

## Registracija i prijavljivanje

Zbog raznih problema koji mogu nastati ako lozinke čuvamo kao čist tekst u bazi, bitno je da te probleme sprečimo na vreme. Ovo postižemo tako što ćemo lozinke enkriptovati prilikom čuvanja u bazi [13]. Sve što je potrebno jeste da injektujemo implementaciju jednog od ***PasswordEncodera***, npr. ***BCryptPasswordEncoder***. Kako bismo omogućili da *Spring* automatski obavi proveru kredencijala, konfigurisaćemo i ***AuthenticationManager***. *AuthenticationManager* autentifikuje korisnike uz pomoć ***DaoAuthenticationProvider*** koji uz pomoć *Dependency injection-a* pronalazi *AppUser* servis koji pronalazi korisnika i proverava njegove kredencijale. ***Configuration*** anotacija označava da će se u klasi nalaziti ***Bean*** objekti koje *Spring* treba da učita u svoj kontejner pre početka rada programa (listing 22).

@Configuration  
public class AppConfig {  
  
 @Bean  
 public AuthenticationManager authenticationManager(UserDetailsService userDetailsService) {  
 DaoAuthenticationProvider provider = new DaoAuthenticationProvider();  
 provider.setUserDetailsService(userDetailsService);  
 provider.setPasswordEncoder(*passwordEncoder*());  
 return new ProviderManager(provider);  
 }  
  
 @Bean  
 public static PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
}

Listing 22 Konfiguracija autentifikacije

Kako bismo omogućili da naš server prima *HTTP* zahteve za registraciju i prijavljivanje od frontenda potrebno je da kreiramo *Auth* kontroler. Kontroler anotiramo sa ***RestController*** anotacijom kako bismo omogućili obrađivanje *HTTP* zahteva u okviru klase i sa anotacijom ***RequestMapping*** kako bi naznačili /*auth* kao osnovnu putanju kontrolera.

Metoda ***authenticate*** jeste akcija koju želimo da se izvrši kada klijenti žele da se prijave u sistem, a ***registerUser*** kada žele da se registruju. Anotiraćemo ih sa ***PostMapping*** anotacijom koja označava da se ove metode pozivaju sa *HTTP POST* zahtevom ka */auth/login* ili */auth/registration* ruti. *Authenticate* metoda očekuje ***TokenRequest*** objekat kao parametar koji je prosleđen kroz telo poruke i zato ga označavamo sa ***RequestBody*** anotacijom, on sadrži imejl i lozinku koju mu je prosledio klijent, dok *registerUser* metoda očekuje ***RegistrationRequest*** koji sadrži podatke za registraciju: ime, prezime, imejl i lozinku. Ove metode pozivaju *Auth* servis koji izvršava logiku i koji je injektovan uz pomoć *depency* *injection-a* i vraćaju odgovarajuću *HTTP* poruku. Implementacija kontrolera je prikazana na listingu 23.

@RestController  
@RequestMapping("/auth")  
@AllArgsConstructor  
public class AuthController {  
  
 private final AuthService authService;  
  
 @PostMapping("/login")  
 public ResponseEntity<TokenResponse> authenticate(@RequestBody TokenRequest tokenRequest) {  
 return new ResponseEntity<>(authService.login(tokenRequest), HttpStatus.*OK*);  
 }  
  
 @PostMapping("/registration")  
 public ResponseEntity<AppUser> registerUser(@RequestBody RegistrationRequest request) {  
 return new ResponseEntity<>(authService.register(request, ERole.*ROLE\_REGULAR*), HttpStatus.*CREATED*);  
 }  
}

Listing 23 Implementacija kontrolera za prijavu i registraciju

*Auth* servis će unutar login metode uz pomoć injektovanog *authenticationManagera* izvršiti ***authenticate*** metodu za prosleđene kredencijale i vratiti ***Authentication*** objekat ukoliko su kredencijali ispravni ili će vratiti grešku ukoliko takvi kredencijali ne postoje u bazi. Nakon toga, potrebno je da generišemo *JSON Web* token za korisnika i vratiti mu token i osnovne podatke njegovog naloga, kao što je prikano na listingu 24.

public TokenResponse login(TokenRequest request) {  
 Authentication authentication = authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(request.email, request.password));  
 SecurityContextHolder.*getContext*().setAuthentication(authentication);  
 return appUserService.generateToken(request.email);  
}

Listing 24 Metoda servisa koja obrađuje logiku prijave na sistem

*JSON Web Token* se generiše na osnovu imejla korisnika. Kada imamo imejl korisnika, pronađemo istog u bazi i generišemo token koji će sadržati ulogu korisnika koju ćemo kasnije koristiti prilikom slanja podataka. Sve što je potrebno jeste da ga potpišemo privatnim ključem i dodelimo mu vreme trajanja. Implementacija generisanja tokena prikazana je na listingu 25.

public TokenResponse generateToken(String email) {  
  
 AppUser appUser = appUserRepository.findById(email).orElseThrow(() -> new UsernameNotFoundException(email));  
  
 Map<String, Object> claims = new HashMap<>();  
 claims.put("role", user.getRole().getRole().name());  
  
 TokenResponse tokenResponse = new TokenResponse();  
 tokenResponse.token = Jwts.*builder*()  
 .setClaims(claims)  
 .setSubject(user.getEmail())  
 .setIssuedAt(Date.*from*(Instant.*now*()))  
 .setExpiration(Date.*from*(Instant.*now*().plus(300L, ChronoUnit.*MINUTES*)))  
 .signWith(getKey())  
 .compact();  
  
 tokenResponse.user = appUser;  
 return tokenResponse;  
}

Listing 25 Generisanje JSON Web tokena i potpisivanje privatnim ključem

Kako bismo u potpunosti omogućili komunikaciju između bekend i frontend sistema, potrebno je da podesimo ***WebSecurityConfig*** na našoj bekend aplikaciji. *Spring* ne dozvoljava komunikaciju između programa sa različitim domenima, samim tim nam je potrebno da manuelno omogućimo komunikaciju između naša dva programa tako što ćemo postaviti *URL* na kom je naš frontend pokrenut kao jedan od dozvoljenih izvora za razmenu poruka, kao što je prikazano na listingu 26. Takođe, omogućujemo da pozivi za registraciju i prijavljivanje mogu biti pozvani bez postojeće autorizacije, dok će za sve ostale pozive biti potrebna prijava kako bi se pozvali.

@Configuration  
@EnableMethodSecurity  
@EnableWebSecurity  
public class WebSecurityConfig {  
  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
  
 http .cors(cors -> cors  
 .configurationSource(request -> {  
 CorsConfiguration config = new CorsConfiguration();  
 config.setAllowedOrigins(List.*of*("http://localhost:4200"));  
 config.setAllowedMethods(List.*of*("\*"));  
 config.setAllowedHeaders(List.*of*("\*"));  
 return config;  
 }))  
 .authorizeHttpRequests(authorize -> {  
 authorize  
 .requestMatchers("/auth/\*\*").permitAll()  
 .anyRequest().authenticated();  
 });  
  
 return http.build();  
 }  
}

Listing 26 Omogućavanje komunikacije između aplikacija i konfiguracija autorizacije

Za kreiranje forme za registraciju na frontendu potrebna nam je komponenta koja će izvršavati ovu funkcionalnost. Uz pomoć *Angular CLI* i komande ***ng generate component register***kreiramo ***Register*** komponentu koja će nam pružiti funkcionalnost za registrovanje korisnika. Polja koja su nam potrebna za registrovanje jesu ime, prezime, imejl i lozinka. Kako bismo uspostavili vezu između *HTML* elemenata i polja u skript fajlu komponente, koristićemo *two-way data binding* na poljima uz pomoć ***ngModel*** direktive*,* koja je prethodno objašnjena.

Takođe, kreiramo i ***Login*** komponentu koja će nam pružiti funkcionalnost za prijavljivanje na sistem. Za popunjavanje forme za prijavljivanje potrebna su nam polja imejl i lozinka koje ćemo takođe povezati uz pomoć *two-way data binding-a.*

Kako bi naše komponente registrovale i prijavile korisnike potrebno im je da komuniciraju sa bekend serverom. Kreiraćemo servis uz pomoć komande ***ng generate service auth*** koji će služiti za slanje i primanje *HTTP* poruka sa bekenda. Poruke se šalju uz pomoć ***HttpClienta*** i njihova povratna vrednost je ***Observable,*** koja je prethodno objašnjena. Povratnu vrednost dobijamo *HTTP POST* zahtevom ka bekendu i kao argumente prosledimo bekend putanju kojom pozivamo metodu za autorizaciju i telo našeg zahteva, kao što je prikazano na listingu 27.

export class AuthService {

  isLoggedIn = false;

  constructor(private http: HttpClient) { }

  login(email: string, password: string): Observable<any> {

    return this.http.post(AUTH\_API + 'auth/login', { email, password });

  }

  register(firstName: string, lastName: string, email: string, password: string): Observable<any> {

    return this.http.post(AUTH\_API + 'auth/registration, { firstName, lastName, email, password });    
 }

}

Listing 27 Servis za autentifikaciju korisnika

Prilikom klika na dugme za prijavljivanje, pozvaće se ***login*** metoda *Login* komponente koja iščekuje krajnju vrednost poslatog zahteva. Ukoliko je vrednost vraćena bez greške, izvršiće se kod koji se nalazi u ***next*** bloku i tada zapisujemo da je korisnik ulogovan, čuvamo njegov token i njegove osnovne podatke unutar *sessionStorage[14]* u *browseru* i rutiramo ga na početnu stranu, dok ukoliko je vraćena greška prilikom autorizacije na bekendu, ispisujemo grešku o neispravnim kredencijalima koja se nalazi u *error* bloku. Skript fajl komponenta za prijavljivanje u sistem prikazan je u okviru listinga 28.

export class LoginComponent {

  email: string = '';

  password: string = '';

  constructor(private authService: AuthService, private router: Router) { }

  login() {

    this.authService.login(this.email, this.password).subscribe({

      next: (data) => {

         this.authService.isLoggedIn = true

         window.sessionStorage.setItem('auth-token', data.token)

         window.sessionStorage.setItem('auth-user', JSON.stringify(data.user))

         this.router.navigate(["/home"])

      },

      error: () => {

          window.alert("Invalid credentials!")

      }

    });

  }

}

Listing 28 Skript fajl Login komponente

Nakon prijavljivanja u sistem, naša *browser* sesija će čuvati *JW* *Token* koji će nam pomoći da se autentifikujemo prilikom svakog slanja zahteva ka bekend serveru. Sve što nam je potrebno jeste da kreiramo ***AuthInterceptor*** klasu koja će implementirati *Angular HTTP* *Interceptor* interfejs koji izvršava metodu *intercept* prilikom svakog slanja *HTTP* poruka. Našu poruku ćemo samo dopuniti tokenom, ukoliko token postoji u našem sistemu, kao što je prikazano na listingu 29.

const TOKEN\_HEADER\_KEY = 'Authorization';

@Injectable()

export class AuthInterceptor implements HttpInterceptor {

  constructor(private token: TokenService) { }

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    let authReq = req;

    const token = this.token.getToken();

    if (token != null) {

      authReq = req.clone({ headers: req.headers.set(TOKEN\_HEADER\_KEY, 'Bearer ' + token) });

    }

    return next.handle(authReq);

  }

}

export const authInterceptorProviders = [

  { provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptor, multi: true }

];

Listing 29 Klasa pomoću koje proširujemo HTTP zahteve JW tokenom

## Kreiranje kurseva

Osnovni podaci koje kurs sadrži jesu ime kursa, fotografija, opis i predavač kursa. Naziv kursa će biti primarni ključ tabele, dok će fotografija i opis kursa biti opcioni podaci. Uz pomoć anotacije ***Entity***kreiramo tabelu *Course* u našoj bazi. Polje koje označava predavača ćemo označiti sa **ManyToOne** anotacijom kako bismo predstavili vezu više-prema-jedan kursa ka predavaču i kako bismo kreirali strani ključ u tabeli *Course* koji će pokazivati ka entitetu našeg predavača u tabeli *User*.

@Entity  
public class Course {  
  
 @Id  
 private String name;  
  
 private String pictureURL;  
  
 private String description;  
  
 @ManyToOne  
 private AppUser teacher;  
}

Listing 30 Klasa koja prikazuje entitet kursa u tabeli Course

Na našoj frontent aplikaciji potrebno je omogućiti da samo admini imaju pravo upravljanja kursevima i uz pomoć tokena kojeg dobijamo prilikom prijave ćemo proveriti da li korisnik zapravo ima admin ulogu i ukoliko ima daćemo mu opciju za pristup admin prozoru.

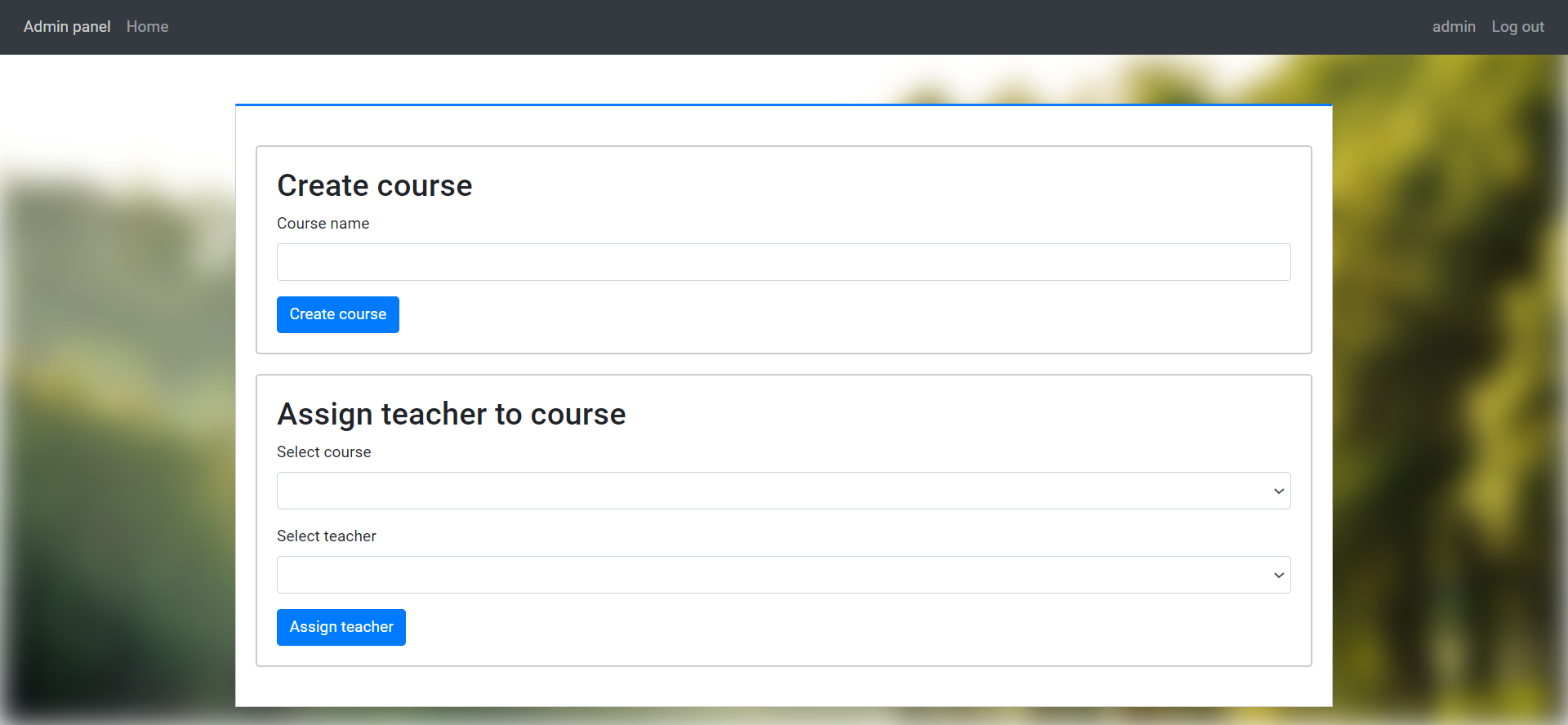
Admin prozor će biti omogućen u okviru navigacione linije pa ćemo zato njegov pristup omogućiti kroz *HTML* fajl koji je uvek prikazan na našoj aplikaciji, ***app.component.html***. Kako bi se ovaj element prikazivao samo kada je potrebno, koristićemo *Angular* ***\*ngIf***direktivu koja će proveriti varijablu *Auth* servisa, koja je injektovana kroz *app.component.ts* fajl, koja će proveriti da li je korisnik prijavljen i ako jeste, uz pomoć tokena proveriti da li trenutni korisnik ima admin ulogu, kao što je prikazano na listingu 31. Klikom na dugme, prebacujemo se na rutu */admin* na kojoj ćemo kreirati novu *Angular* komponentu koja služi za prikazivanje funkcionalnosti za koje admini imaju pristup.

<li \*ngIf="authService.isLoggedIn && isLoggedUserAdmin()">

    <a routerLink="admin">Admin panel</a>

</li>

Listing 31 Dugme za pristup admin prozoru

Admin komponenta sadrži sve funkcionalnosti koje admin korisnici mogu da obavljaju. Trenutno su to kreiranje kurseva i dodela predavača kursu (slika 6). Na našem frontend projektu ovu funkcionalnost ćemo odraditi tako što ćemo čuvati listu kurseva i listu korisnika koje ćemo dovući preko našeg bekend sistema, kako bismo mogli da dodelimo nekom korisniku kurs. Podatke ćemo povezati preko *two-way data binding-a*.

Slika 6 Prikaz admin prozora

Klikom na dugme ***Create course*** će se, ukoliko polje nije prazno, pozvati *Course* servis koji će u komunikaciji sa serverom vratiti entitet kursa ukoliko je isti kreiran ili praznu vrednost ukoliko nije moguće kreirati kurs ako kurs sa tim imenom već postoji. U okviru listinga 32 prikazan je kontroler koji prima zahtev za kreiranjem kursa i prima ime kursa kao telo poruke, dok je u okviru listinga 33 prikazan servis koji izvršava logiku za kreiranje kursa i proverava da li kurs sa tim imenom već postoji trenutno u sistemu.

@PostMapping("/create")  
public ResponseEntity<Course> createCourse(@RequestBody String courseName) {  
 return new ResponseEntity<>(courseService.createCourse(courseName), HttpStatus.*CREATED*);  
}

Listing 32 Metoda kontrolera koja obrađuje zahtev za kreiranjem kursa

public Course createCourse(String courseName) {  
 courseRepository.findById(courseName).ifPresent(c -> {  
 throw new CourseAlreadyExistsException(courseName);  
 });  
  
 Course course = new Course();  
 course.setName(courseName);  
 return courseRepository.save(course);  
}

Listing 33 Metoda servisa koja izvršava logiku za kreiranje kursa

Nakon odabira kursa i predavača iz padajuće liste sa slike 6, klikom na dugme ***Assign teacher*** poslaćemo zahtev serveru koji sadrži naziv izabranog kursa i predavača kojem želimo da dodelimo kurs. U okviru listinga 34 prikazana je logika servera za dodeljivanje predavača kursu.

public void changeTeacher(CourseTeacherRequest request) {  
 Course course = courseRepository.findById(request.courseName);   
 AppUser newTeacher = userRepository.findById(request.newTeacherEmail)   
 course.setTeacher(newTeacher);  
}

Listing 34 Metoda servisa koja izvršava logiku za menjanje predavača kursa

## Upravljanje kursevima

Svakom korisniku koji se uspešno prijavi u sistem i koji predaje bar jedan kurs, biće ponuđeno dugme kroz navigacionu liniju za otvaranje prozora o upravljanju kursevima na kojima on predaje na isti način kao i admin korisnicima u vezi admin prozora, preko *app.component.html* fajla i zvaće se ***Manage courses***. Klikom na ovo dugme otvoriće se nova komponenta ***manage-courses*** čija je funkcionalnost da prikaže sve kurseve na kojima korisnik predaje i omogući otvaranje novog prozora za izmenu kursa klikom na jedan od ponuđenih kurseva.

Kako bismo omogućili ovu funkcionalnost, potrebno je da povučemo sve kurseve iz baze koje korisnik predaje i ubaciti ih u listu ***coursesITeach***, ovo ćemo implementirati slanjem *HTTP GET* zahteva ka serveru koji će filtrirati kurseve i vratiti samo one kod kojih je predavač postavljen na imejl iz zahteva. Imena kurseva ćemo prikazati uz pomoć ***\*ngFor*** direktive u *HTML* fajlu komponente. Klikom na ime jednog od kurseva pozvaće se ***editCourse*** metoda koja za parametar prima kurs na koji smo kliknuli. Ova metoda će nas rutirati ka novoj komponenti sa detaljima kursa i toj komponenti ćemo kroz *History state API* proslediti objekat ovog kursa, kao što je prikazano u okviru listinga 35.

editCourse(course: Course) {

    this.router.navigate(['/editCourse/', course.name], {state: {course: course}});

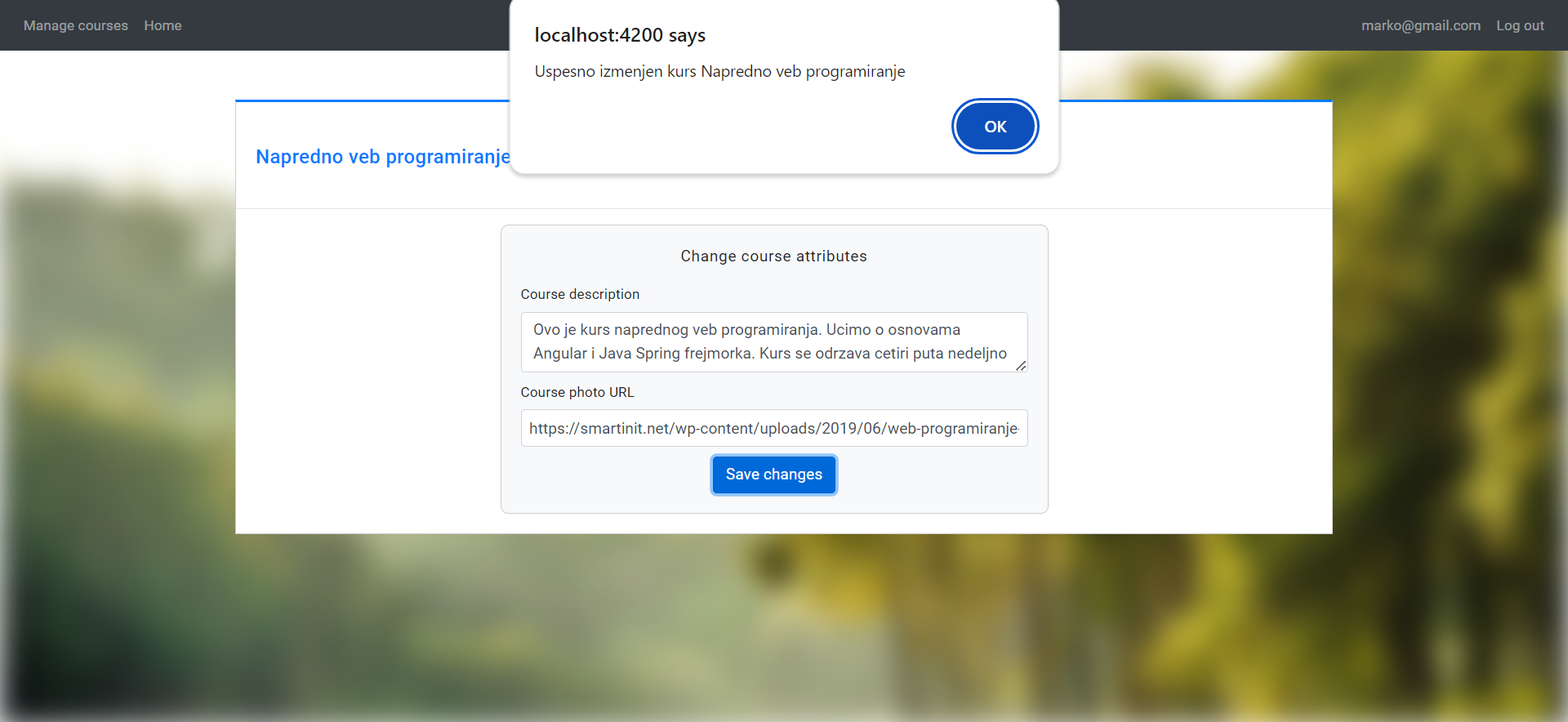
}

Listing 35 Metoda koja rutira i prosleđuje objekat komponenti

Komponenta ***course-details,*** na koju smo rutirali, sadrži dva tekstualna polja, jedno koje predstavlja opis kursa i drugo koje predstavlja *URL* fotografije kursa, ova dva podatka možemo menjati. Klikom na dugme ***Save changes*** poslaćemo *HTTP PUT* zahtev bekendu za izmenu podataka koji sadrži naziv kursa, opis i fotografiju kursa. Bekend će ovaj poziv obraditi i u servisu pronaći kurs prosleđenim imenom i postaviti nove vrednosti za ova dva podatka, kao što je prikazano na listingu 36. Na slici 7 možemo videti kako izgleda prozor za izmenu detalja kursa i uspešna poruka nakon izmene podataka.

public Course changeData(CourseChangeDataRequest request) {  
 Course course = courseRepository.findByName(request.courseName);  
 course.setDescription(request.description);  
 course.setPictureURL(request.pictureURL);  
 return course;  
}

Listing 36 Logika servisa za izmenu podataka kursa



Slika 7 Prikaz komponenta za izmenu podataka o kursu i poruka o uspešnoj izmeni

## Kreiranje lekcija

Nakon kreiranja kurseva i postavljanje osnovnih podataka o njima, možemo ih proširiti lekcijama. Lekcije će se sastojati od naziva, *URL* videa koji će prikazivati lekciju i tekstualnog opisa lekcije. Pored ovih osnovnih podataka potreban nam je i automatsko generisani identifikator i kurs kome će se lekcija dodeliti. Na listingu 37 prikazan je entitet lekcije u Javi.

@Entity   
public class Lecture {  
  
 @Id  
 @GeneratedValue  
 Long id;  
 private String title;  
 private String videoURL;  
 private String data;  
  
 @ManyToOne  
 @JsonIgnore  
 private Course course;  
}

Listing 37 Entitet lekcije u Javi

Polje kurs obeležavamo sa anotacijom ***ManyToOne*** koja označava vezu više-prema-jedan između lekcije i kursa. Ovo znači da će naša tabela *Lecture* sadržati strani ključ *course* koji pokazuje na entitet kursa za koji je lekcija napravljena. Anotacija ***JsonIgnore*** označava da ne želimo da povlačimo podatke o polju koje je anotirano prilikom povlačenja podataka o njegovoj klasi.

Kako bismo došli do komponente koja ima funkcionalnost kreiranja lekcija potrebno je da preko prozora za predavače odaberemo kurs kojem želimo da dodamo lekciju i klikom na dugme ***Create lecture*** rutiramo do nove komponente ***create-lecture***. Naša aplikacija će znati kom kursu želimo da kreiramo lekciju tako što će prilikom rutiranja ka adresi komponente za kreiranje lekcija dodati objekat kursa kao prosleđeno polje kroz već objašnjeni *History State API* *Angulara*, kao što je prikazano na listingu 38.

redirectToCreateLectures() {

  this.router.navigate(['/createLecture', this.course.name], {state: {course: this.course}});

}

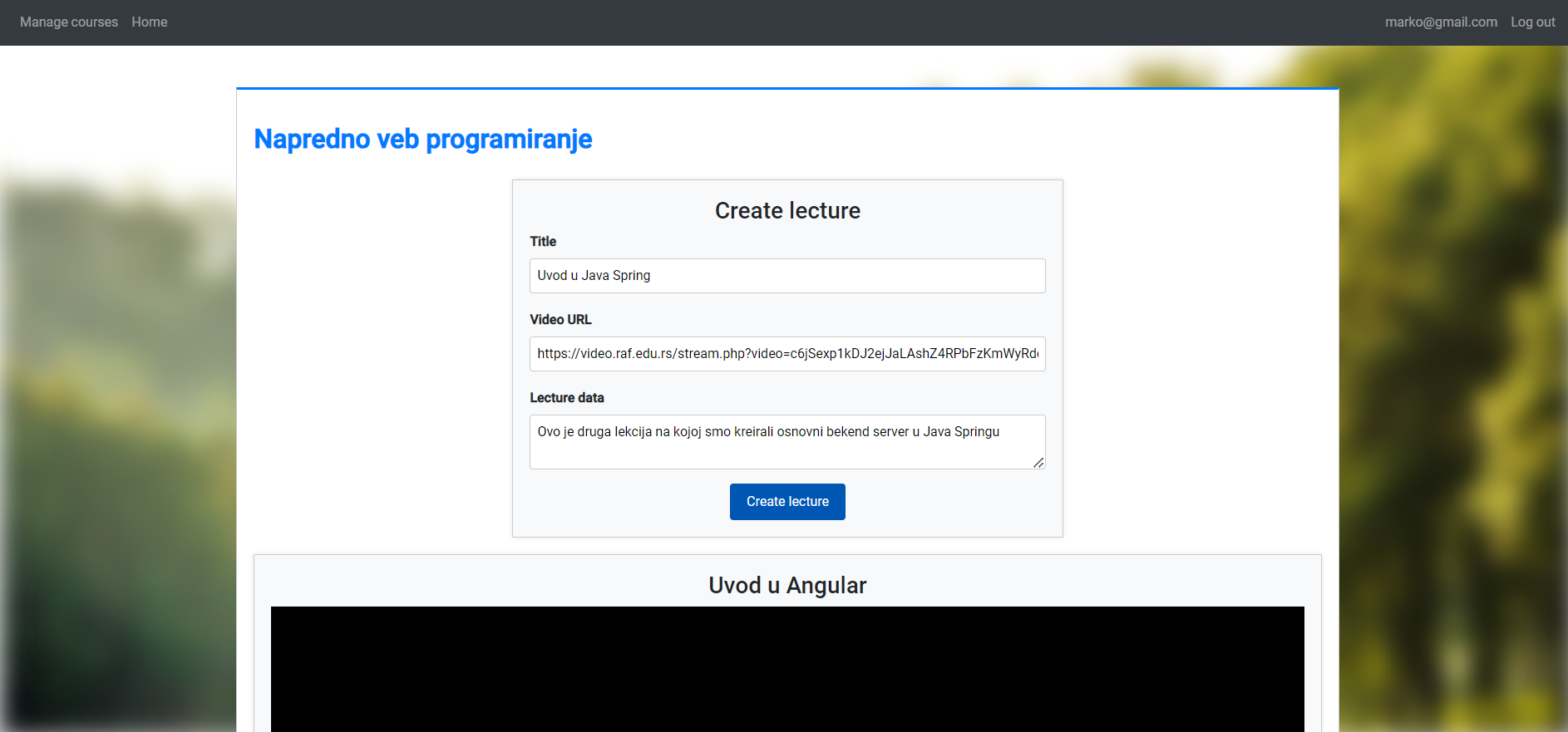
Listing 38 Rutiranje uz prosleđivanje objekta sa History State API

Komponenta za kreiranje lekcija sadrži tri polja: naziv lekcije, *URL* videa lekcije i tekstualni podatak o lekciji koja funkcionišu uz pomoć *two-way data bindinga*. Slika 8 sadrži prikaz ove komponente. Klikom na dugme ***Create lecture*** u okviru prozora, poslaćemo *HTTP POST* zahtev ka našem bekendu sa podacima koja su upisana u tri polja i naziv kursa. Bekend će uz pomoć ***LectureControllera*** pročitati naš zahtev i uz pomoć servisa obraditi logiku za čuvanje lekcije u tabeli *Lecture,* što je prikazano na listingu 39.

public Lecture createLecture(LectureCreateRequest request) {  
 Course course = courseRepository.findById(request.courseName);

Lecture lecture = new Lecture();  
 lecture.setTitle(request.title);  
 lecture.setVideoURL(request.videoURL);  
 lecture.setData(request.data);  
 lecture.setCourse(course);  
 return lectureRepository.save(lecture);  
}

Listing 39 Logika za čuvanje lekcija u bazi



Slika 8 Prikaz komponente za kreiranje lekcija

## Prijavljivanje na kurs i pretraga

Kako bismo zapamtili koji korisnik pohađa koji kurs potrebna su nam dva podatka: imejl korisnika i naziv kursa koji pohađa. Napravićemo novu tabelu u našoj bazi sa nazivom ***Enrollment*** koja će sadržiti ova dva podatka.

@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class Enrollment {  
  
 @EmbeddedId  
 private EnrollmentPK id;  
  
 @ManyToOne  
 @MapsId("course")  
 private Course course;  
  
 @ManyToOne  
 @MapsId("student")  
 private AppUser student;  
}

@Data  
@Embeddable  
public class EnrollmentPK implements Serializable {  
  
 @ManyToOne  
 private Course course;  
  
 @ManyToOne  
 private AppUser student;  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 EnrollmentPK that = (EnrollmentPK) o;  
 return Objects.*equals*(course.getName(), that.course.getName()) && Objects.*equals*(student.getEmail(), that.student.getEmail());  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(course, student);  
 }  
}

Listing 40 Klase koje predstavljaju Enrollment entitete sa kompozitnim ključem

Entitet ***Enrollment*** u tabeli sadrži imejl studenta i naziv kursa za koji se prijavio. Obzirom na to da nam je ovaj podatak direktno vezan za oba polja, napravićemo da bilo koja dva reda u ovoj bazi ne sadrže istog studenta i isti test, jer bi onda tabela bila nevalidna. Ovo ćemo postići tako što ćemo napraviti **kompozitni primarni ključ**. ***EnrollmentPK*** klasa će predstavljati naš kompozitni ključ i u njoj postoje dva polja, *course* i *student*. Oba polja predstavljaju vezu više-ka-jedan, koju smo prethodno objasnili, stoga ćemo ih anotirati sa ***ManyToOne*** anotacijom u obe klase. Klasu kompozitnog ključa anotiramo sa ***Embeddable***, što označava da sva polja u ovoj klasi čine jedan gradivni blok koji može da se umetne u neku drugu klasu. Takođe, potrebno je da klasa implementira ***Serializable*** i njene dve metode, ***equals i hashCode***, kako bismo znali kako se razlikuju dva objekta ove klase. Nakon toga, sve što je potrebno kako bismo koristili *EnrollmentPK* klasu kao naš kompozitni ključ u *Enrollment* entitetima jeste da je naznačimo anotacijom ***EmbeddedId*** i da polja mapiramo ka našem kompozitnom ključu uz pomoć ***MapsId*** anotacije, kao što je prikazano na listingu 40.

Kako bi se korisnik prijavio na kurs, potrebno je da izlistamo sve kurseve na početnoj strani naše aplikacije i omogućimo mu da klikom na dugme ***Enroll course*** započne slušanje kursa. Ovu funkcionalnost implementiramo tako što ćemo povući sve postojeće kurseve iz baze i prikazažemo uz pomoć *home.components.html* fajla i *Angular* direktiva (listing 41). Nakon uspešnog učitavanja kurseva, dobijeni rezultat možemo videti na slici 9.

<h3>Available courses</h3>

<div \*ngFor="let course of courses">

    <img \*ngIf="course.pictureURL" [src]="course.pictureURL">

    <h3>{{ course.name }}</h3>

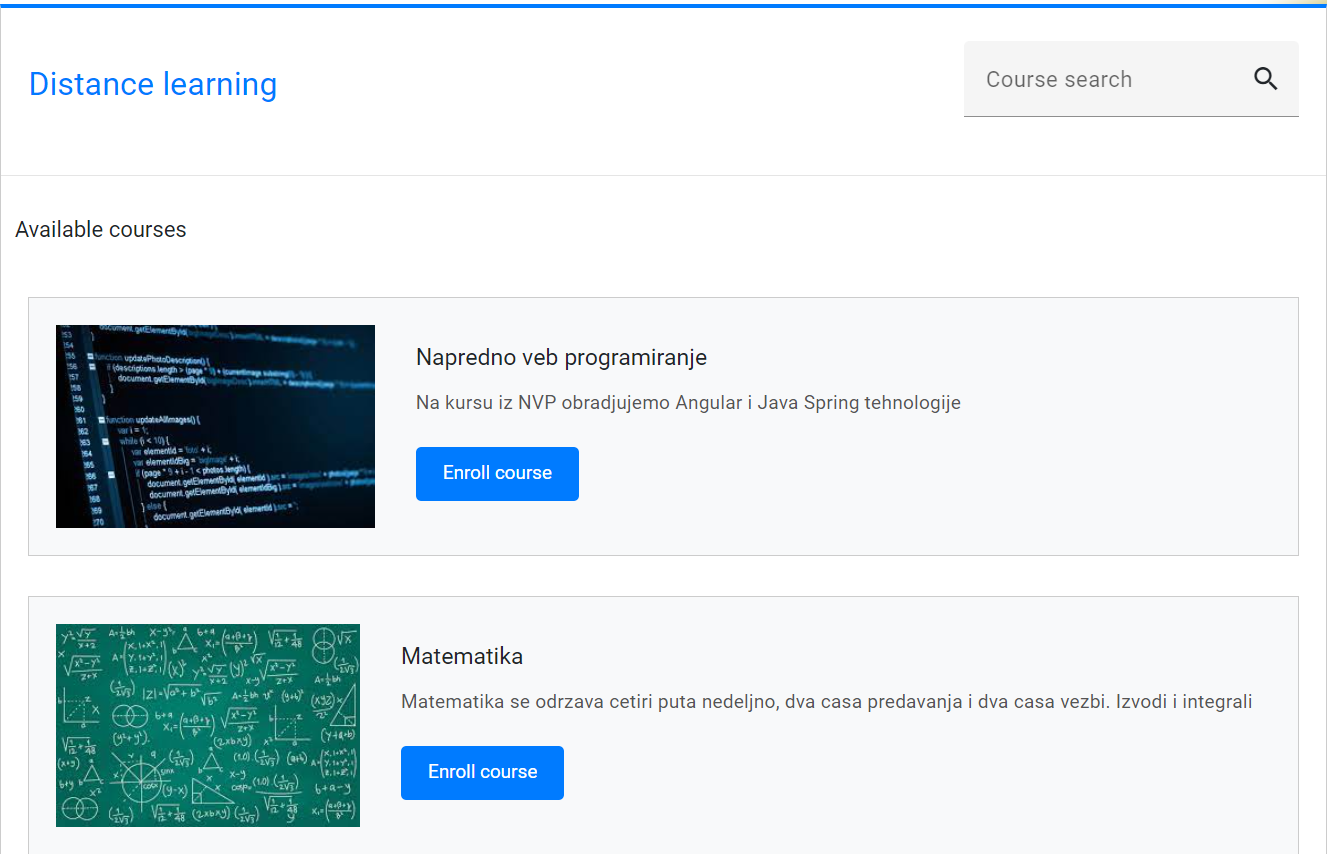
    <p \*ngIf="course.teacher.name != null">Professor: {{ course.teacher.name }}</p>

    <p \*ngIf="course.description">{{ course.description }}</p>

    <button (click)="enrollCourse(course)">Enroll course</button>

</div>

Listing 41 Prikaz svih kurseva na početnoj strani pomoću ngFor direktive



Slika 9 Prikaz svih kurseva na početnoj strani

Klikom na ***Enroll course*** poziva se *HTTP PUT* zahtev ka našem bekendu koji će u svom servisu proveriti da li u bazi postoje kurs na koji korisnik želi da se prijavi i sam korisnik i ako postoje, sačuvaće i samu prijavu.

public Enrollment enroll(EnrollRequest request) {  
 Course course = courseRepository.findById(request.courseName);   
 AppUser student = appUserRepository.findById(request.email);  
  
 EnrollmentPK pk = new EnrollmentPK();  
 pk.setCourse(course);  
 pk.setStudent(student);  
  
 Enrollment enrollment = new Enrollment();  
 enrollment.setId(pk);  
  
 return enrollmentRepository.save(enrollment);  
}

Listing 42 Metoda servisa za čuvanje prijava na kurseve

Pretraga kurseva je implementirana na tekstualnom polju sa slike 9 pomoću ***(ngModelChange)*** *Angular* direktive koja se poziva prilikom svake izmene polja nad elementom za koji je vezana. Ova direktiva će za svaku promenu pozvati ***onSearchChange*** funkciju iz ***home.component.ts*** fajla koja će filtrirati sve kurseve koji ne sadrže tekst iz polja u svom imenu, kao što je prikazano na listingu 43.

onSearchChange() {

    this.coursesFiltered = this.courseService.courses;

    this.coursesFiltered.forEach(course => {

      if(!course.name.includes(this.searchTerm)) {

        this.coursesFiltered = this.coursesFiltered.  
 filter(item => item.name != course.name)

}})

Listing 43 Metoda koja se poziva prilikom svake izmene u Course Search polju

## Kreiranje testova

Kako naša aplikacija omogućava kreiranje i izradu testova, potrebno je da napravimo model koji će čuvati testove i pitanja u bazi. Lekcije mogu imati testove, ali ne moraju. Svaki test je moguće otvoriti sat vremena od početka otvaranja testa i sadrži pet pitanja, dok svako pitanje ima jedan tačan i tri netačna odgovora.

@Entity  
public class Test {  
  
 @Id  
 @GeneratedValue  
 private Long id;  
  
 @Temporal(TemporalType.*TIMESTAMP*)  
 private Date testStartDate;  
  
 @OneToOne  
 @JsonIgnore  
 private Lecture lecture;  
  
 @OneToMany(mappedBy = "test", cascade = CascadeType.*ALL*)  
 private List<Question> questions = new ArrayList<>();  
}

Listing 44 Entitet testa sa poljima

Entitet testa u našoj aplikaciji sadrži automatsko generisani identitet, vreme starta testiranja, lekciju za koju je test napravljen i listu pitanja. ***Temporal*** anotacija sa postavljenim *TemporalType.TIMESTAMP* atributom nad *Date* poljem označava da će se ovo polje čuvati u koloni tabela pomoću ***SQL Timestamp*** objekta. Polje lekcija ćemo označiti sa ***OneToOne*** anotacijom kako bismo naznačili da lekcija može imati samo jedan test i da test može biti uvezan samo za jednu lekciju. Takođe, u modelu lekcije ćemo dodati test polje, kako bi lekcija znala za postojanje entiteta testa. Polje sa listom pitanja označavamo sa ***OneToMany*** anotacijom kako bismo naznačili da svaki test ima listu pitanja za koju je vezan. ***mappedBy*** atribut notacije označava da će se mapiranje ovog polja odvijati u test polju u *Question* klasi, što znači da se pitanja neće čuvati u *Test* tabeli, već će se identifikator testa čuvati u *Questions* tabeli koju kreiramo pomoću *Question* klase, prikazane na listingu 44.

@Entity  
public class Question {  
  
 @Id  
 @GeneratedValue  
 public Long id;  
  
 @ManyToOne  
 public Test test;  
  
 public String question;  
 public String rightAnswer;  
 public String wrongAnswer1;  
 public String wrongAnswer2;  
 public String wrongAnswer3;  
}

Listing 45 Entitet pitanja sa poljima

Model pitanja koji se čuva u bazi ima svoj identifikator, test koji će se mapirati u *Questions* tabeli, tačan odgovor na pitanje i tri netačna odgovora.

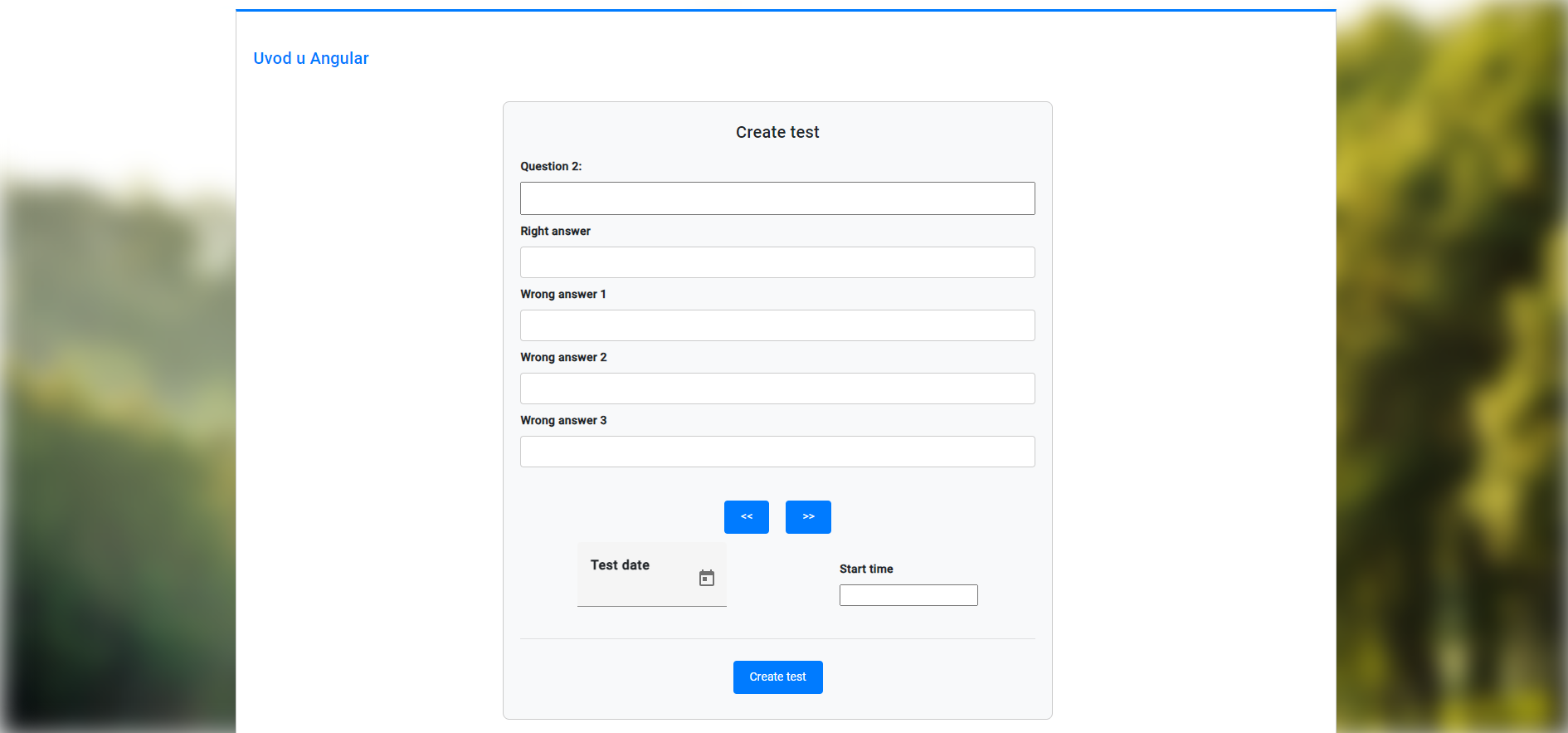
Kako bismo kreirali test, potrebno je biti ulogovan kao profesor nekog od kursa i pomoću prozora za predavače odabrati kurs kojeg želimo da izmenimo. Odabirom kursa otvoriće se prozor na kome možemo odabrati koju funkcionalnost želimo da izvršimo nad kursom. Klikom na ***Create test*** želimo da nam se prikaže komponenta za odabir lekcije nad kojom želimo da kreiramo test. Potrebno je da je generišemo komponentu preko *Angular* *CLI* komande ***ng generate component choose-lecture*** i dodati istu u *routing.module.ts* fajl (listing 46). Ova komponenta će prikazati sve lekcije prethodno izabranog kursa i klikom na neku od lekcija rutiraćemo se ka novoj komponenti, koju takođe kreiramo uz pomoć *Angular* *CLI* komande ***ng generate component create-test***, koja ima funkcionalnost kreiranja testa i kojoj prosleđujemo ***lecture*** objekat kroz *History State*.

  { path: 'chooseLecture/:name', component: ChooseLectureComponent },

  { path: 'createTest/:lectureID', component: CreateTestComponent }

Listing 46 Dopuna route liste u routing.module.ts fajlu novim putanjama

Da bismo uspešno kreirali test potrebno je da popunimo polje pitanje, napišemo tačan odgovor i tri netačna odgovora i klikom na strelicu prelazimo na sledeće pitanje i popunjavamo i njega, ove korake radimo za sva pet pitanja. Nakon pitanja, potrebno je da izaberemo datum i vreme početka testa i kliknemo na dugme ***Create test***. Nakon klika, naš skript fajl će uz pomoć *ngModel* direktive proveriti da li su sva polja popunjena i ukoliko jesu, poslaće servisu identifikator lekcije, listu pitanja i datum i vreme početka izrade testa za koje će servis napravi *HTTP POST* zahtev sa svim ovim podacima umetnutim u telo poruke koju prosleđuje bekendu, kako bi on kreirao test u bazi. Na slici 10 prikazan je izgled komponente za kreiranje testova.



Slika 10 Prikaz prozora za kreiranje testova

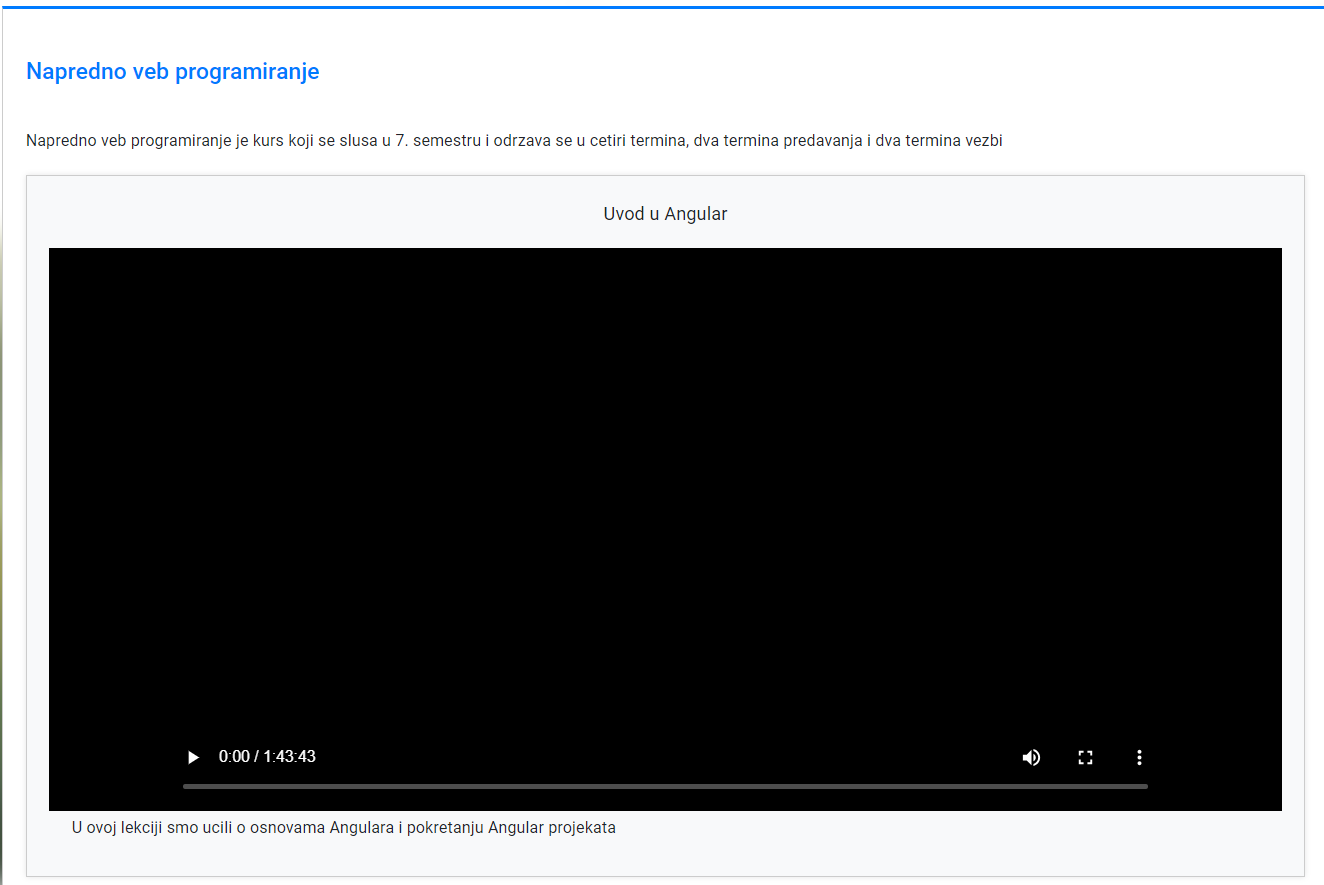
Kako bismo test uspešno sačuvali u našoj bazi prvo pronalazimo lekciju za koju je test kreiran. Nakon toga ćemo testu postaviti vreme početka i ID lekcije i takođe ćemo postaviti identifikator testa u entitetu lekcije. Nakon čuvanja ova dva entiteta u bazi, možemo svakom pitanju postaviti novonastali identifikator testa kao strani ključ i sačuvati i pitanja u *questions* tabelu, kao što je prikazano na listingu 47.

public Test createTest(TestCreateRequest request) {  
 Lecture lecture = lectureRepository.findById(request.getLectureID());  
 Test test = new Test();  
 test.setTestStartDate(request.getStartDate());  
 test.setLecture(lecture);  
 lecture.setTest(test);  
 testRepository.save(test);  
 lectureRepository.save(lecture);  
 request.getQuestions().forEach(question -> question.setTest(test));  
 questionRepository.saveAll(request.getQuestions());  
 return test;  
}

Listing 47 Logika čuvanja testa i pitanja u bazi podataka

## Pregled lekcija

Nakon prijavljivanja na kurs iz poglavlja potrebno je korisniku omogućiti i sam pristup lekcijama kursa. Na početnoj strani, umesto dugmeta za prijavu, pojaviće se dugme ***Enter course*** ukoliko se korisnik već prijavio na kurs. Nakon klika na dugme aplikacija će rutirati ka novoj putanji koja vodi do nove komponente. Komponentu ćemo nazvati ***course-page*** i kreiramo je uz pomoć *Angular CLI* komande *ng generate component course-page*. Za novu putanju koja vodi do ove komponente ćemo odabrati */course* i novu putanju ćemo registrovati u ***app-routing.module.ts*** fajlu. Pored samog rutiranja poslaćemo i podatak o kursu kojeg želimo da učitamo uz pomoć *History State API-ja* kojeg smo prethodno objasnili.

Nakon otvaranja prozora želimo da prikažemo osnovne informacije o kursu poput naziva i opisa kursa i želimo da prikažemo sve kreirane lekcije kursa. Kako podatke o kursu već imamo kroz *History State API* funkcionalnost, potrebne su nam još lekcije. Lekcije ćemo dovući uz pomoć *HTTP GET* zahteva ka bekendu koji sadrži naziv kursa kao parametar, a kao odgovor iščekuje listu lekcija. Nakon što bekend uspešno vrati listu lekcjia kao odgovor, *Angular* će sačuvati sve lekcije u listu i prikazati ih korišćenjem *Angular* **\*ngFor** direktive. Prikaz jedne lekcije možemo videti na slici 11.

Slika 11 Prikaz lekcije u okviru prozora kursa

## Izrada testova

Nakon što je test kreiran u bazi, korisnicima koji su prijavljeni na kurs za čiju je lekciju napravljen test bi trebalo omogućiti funkcionalnost za ocenjivanje. Sve što je potrebno jeste da u okviru prozora na kom je prikazan kurs klikom na dugme***Start test*** ispod lekcije započnu testiranje.

Potrebno je da proširimo bazu ***TestScore*** tabelom koja će čuvati podatke da li je korisnik već pohađao test i koju je ocenu dobio. Entitet *TestScore* u će u tabeli sadržati imejl studenta, identifikator testa koji je pohađao i ocenu. Obzirom na to da nam je ovaj podatak vezan za studenta i test, napravićemo da bilo koja dva reda u ovoj bazi ne sadrže istog studenta i isti test, jer bi onda tabela bila nevalidna. Ovo ćemo postići tako što ćemo napraviti kompozitni primarni ključ, baš kao što smo to uradili i za *Enrollment* entitete prikazane na listingu 40. U okviru listinga 48 prikazan je entitet *TestScore* modela implementiran u Javi.

@Entity  
public class TestScore {  
  
 @EmbeddedId  
 private TestScorePK id;  
  
 @ManyToOne  
 @MapsId(“student”)  
 AppUser student;  
  
 @ManyToOne  
 @MapsId(“test”)  
 Test test;  
  
 int score = 0;  
}

Listing 48 Entitet TestScore implementiran u Javi

Kako bismo na našoj lekciji prikazali odgovarajuće stanje u vezi sa tim da li je student već polagao test i ako jeste, kojom je ocenom položio, postoje više situacija:

* test još uvek nije počeo,
* test je počeo, a student nije započeo testiranje,
* student je završio testiranje ili
* test se završio, a student se nije testirao.

Listing 49 nam prikazuje na koji način ćemo obuhvatiti ove situacije, ali nam nedostaju podaci o tome da li je student odradio test i koju je ocenu dobio, pa ćemo morati da proširimo naš *API* za dohvatanje lekcije kako bi dobili i ove nedostajuće podatke.

<div \*ngIf="lecture.test != null">

    <p \*ngIf="beforeTest(lecture)">Test start: {{lecture.test.testStartDate}}</p>

    <button \*ngIf="duringTest(lecture) && testScores[i] == null" (click)="startTest(lecture)">Start test</button>

    <p \*ngIf="testScores[i] != null">Test completed {{ testScores[i] }}/5</p>

    <p \*ngIf="afterTest(lecture) && testScores[i] == null">Test has been closed </p>

   </div>

Listing 49 Stanja u vezi prikaza dugmeta za početak testiranja

Kako bismo dobili podatke o tome da li je student položio test i kojom ocenom, *API* za dovlačenje lekcija ćemo proširiti uz pomoć ***Request parametra extended*** koji označava da li su nam prilikom poziva potrebni i dodatni podaci o studentovim prethodnim testiranjima ili su nam potrebni samo osnovni podaci o lekciji. ***RequestParam*** jeste anotacija koju koristimo kako bismo čitali *request* parametre *HTTP* zahteva, koja će nam koristiti prilikom implementacije prikazanoj na listingu 50. Nakon dodavanja parametra na bekendu, *HTTP* zahtev možemo slati zajedno sa parametrom kroz *URL*, npr. *GET* http://distance-learning.com /api/course/Matematika?extended=true koji predstavlja *HTTP GET* zahtev koji za *request* parametar *extended* ima postavljenu vrednost *true*.

@GetMapping("/{courseName}")  
public ResponseEntity getLecturesByCourseName(@PathVariable String courseName, @RequestParam(defaultValue = "false") boolean extended) {  
 if(extended) {  
 return new ResponseEntity<> (lectureService.getLecturesExtByCourseName(courseName), HttpStatus.*OK*);  
 }  
 return new ResponseEntity<> (lectureService.getLecturesByCourseName(courseName), HttpStatus.*OK*);  
}

Listing 50 Bekend proširenje API-ja za primanje Request parametara prilikom poziva

Sledeće što je potrebno jeste da u servisu za lekcije na bekendu implementiramo funkciju za vraćanje dodatnih podataka. Ukoliko lekcija sadrži test, proverićemo da li je student taj test polagao i ako jeste, vratićemo tu lekciju zajedno sa ocenom koju je dobio, dok ukoliko nije, vratićemo samo osnovne podatke o lekciji. Podaci o oceni i polaganju nalaze se u ***TestScore*** entitetima u bazi. Logika za obradu ovih podataka prikazana je na listingu 51.

public List<LectureExt> getLecturesExtByCourseName(String courseName) {  
 Course course = courseRepository.findById(courseName).orElseThrow(() -> new CourseNotFoundException("Course with name " + courseName + " not found!"));  
 List<Lecture> lectures = course.getLectures();  
 List<LectureExt> response = new ArrayList<>();  
  
 for(Lecture lecture : lectures) {  
 if(lecture.getTest() != null) {  
 Optional<TestScore> ts = scoreRepository.  
 findByStudentEmailAndTestId(jwtUtils.getCurrentUsername(), lecture.getTest().getId());  
 if(ts.isPresent()) response.add(new LectureExt(lecture, ts.get()));  
 else response.add(new LectureExt(lecture, null));  
 }  
 }  
 return response;  
}

Listing 51 Logika vraćanja podataka o lekcijama i postignutim ocenama

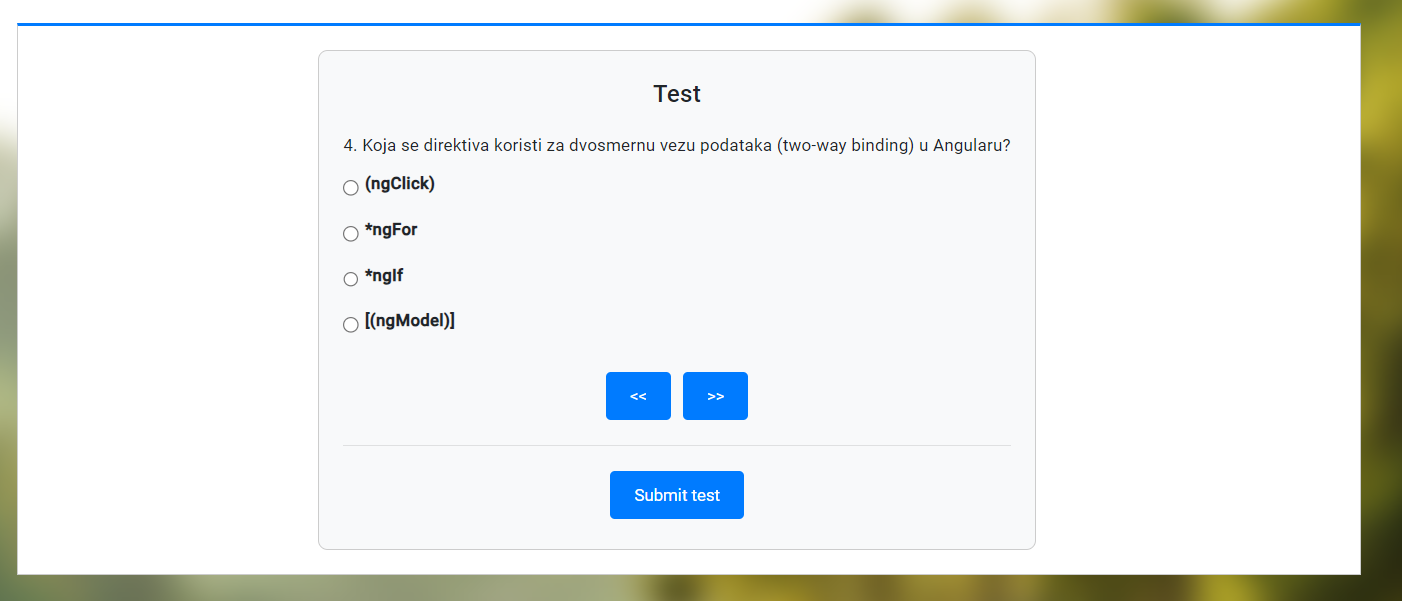
Nakon implementacije novog načina dohvatanja lekcija, izmenićemo na frontendu *HTTP GET* poziv i uz pomoć parametra ćemo dohvatiti i dodatne podatke o lekciji i studentovog polaganja testa (listing 52). Nakon ove izmene, frontend ima sve potrebne informacije za prikaz svih mogućih situacija sa izradom testa iz listinga 49.

getLectures(course: string, extended: boolean): Observable<any> {

    return this.http.get(AUTH\_API + `lecture/${course}` + "?extended=" + extended);

}

Listing 52 Proširenje API poziva za dodatni parametar extended

Za potrebe izrade testa kreiraćemo novu komponentu ***test-page*** koja će prikazati prozor za izradu testa. Korisniku će biti ponuđena pitanja, odgovori će biti izmešani i biće mu dostupno dugme ***Submit test***. Nakon klika na ovo dugme, korisnik potvrđuje da je završio test, ispisaće mu se broj ostvarenih poena i taj podatak ćemo sačuvati u bazi i prebacićemo korisnika na početnu stranu aplikacije. Na slici 12 prikazan je prozor komponente *test-page.*

Slika 12 Prozor komponente za izradu testa

Da bi ova komponenta funkcionisala, od podataka su joj potrebni pitanja i identifikator lekcije za koju se polaže test. Ove podatke prosleđujemo uz pomoć *History State* funkcionalnosti u *Angularu* prilikom rutiranja na *test-page* komponentu.

Za prikaz svih pitanja koristićemo *Angular* direktivu **\**ngFor*** pomoću koje prolazimo kroz sva pitanja iz liste pitanja i pratićemo indeks tog elementa u listi. Svako od tih pitanja ima četiri odgovora, pa će nam biti potrebno da još jednom iskoristimo **\*ngFor** direktivu i pratimo indeks elemenata iz liste odgovora. Nakon toga nam je potreban *radio* *button* za svaki odgovor, koji se zapravo oslikava na promenljivu ***selectedAnswer*** tog pitanja pomoću *[(ngModel)]* directive. Uz pomoć *Angular* ***[name]*** direktive dinamički možemo kreirati *name* polja *input* elemenata. Polje *name* ćemo koristiti kako bismo odgovore na pitanja spojili u celinu i omogućiti samo odabir jednog odgovora kao tačnog, zbog toga polje *name* postavljamo na vrednost ***question\_i***, gde je *i* indeks pitanja. Direktiva ***[id]*** takođe služi za dinamičko generisanje identifikatora elemenata pomoću polja. U našem primeru želimo da postignemo da svaki odgovor za svako pitanje ima poseban identifkator i zbog toga ćemo koristiti oba indeksa u identifikatoru i postavićemo njegovu vrednost na ***answer\_i\_j***, gde su *i* i *j* indeksi pitanja i odgovora. Labela nam služi za ispisivanje jednog od ponuđenih odgovora i ***[for]*** direktiva nam omogućava dinamičko generisanje *for* polja labele koristeći indekse pitanja i odgovora. Uz pomoć ovog atributa, omogućeno je da se klikom na labelu automatski selektuje i *radio* dugme koje ima isti identifikator kao *for* polje kliknute labele, kao što je prikazano na listingu 53.

<div \*ngFor="let question of questions; let i = index”

  <p>{{ i + 1 }}. {{ question.question }}</p>

  <div \*ngFor="let answer of question.answers; let j = index">

    <input type="radio" [name]="'question\_' + i" [id]="'answer\_' + i + '\_' + j" [(ngModel)]="question.selectedAnswer">

    <label [for]="'answer\_' + i + '\_' + j">{{ answer }}</label>

  </div>

</div>

Listing 53 HTML fajl komponente koja je zadužena za prikaz pitanja i ponuđenih odgovora

Nakon odgovora na sva pitanja testa, izabrani odgovori su mapirani na ***selectedAnswer*** polje svakog pitanja i uz pomoć njega možemo oceniti studenta nakon klika na ***Submit test*** dugme. Metoda za ocenjivanje studenta i slanje zahteva prikazana je na listingu 54. Nakon provere koliko korisnik ima tačnih odgovora, ovaj podatak šaljemo na bekend *HTTP PUT* zahtevom ka ruti */lecture/submitScore* uz identifikator lekcije i broja poena koji je ostvario.

submitTest() {

    this.score = 0;

    this.questions.forEach((question) => {

      if(question.selectedAnswer == question.rightAnswer) {

        this.score += 1;

      }

    });

    this.testService.submitScore(this.lectureID, this.score).subscribe({

      next: () => {

        window.alert("Test finished, your score is: " + this.score);

        this.router.navigate(['/home']);

      },

      error: (err) => {

        console.log(err);

      }

    });

}

Listing 54 Metoda test-page komponente za izračunavanje broja poena

Nakon što bekend obradi pristigli zahtev, pozivamo *TestScore* servis koji je zadužen za čuvanje ovog podatka u bazi. Servis će pronaći studenta kojem treba upisati ocenu tako što će pročitati vrednost iz primljenog *JW Tokena* i lekciju koju će pročitati iz tela poruka. Na osnovu ovih podataka, servis kreira primarni ključ *TestScore* entiteta i dodatno mu dodeljuje broj poena koje je student ostvario na testu koje će preuzeti iz tela poruke. Logika ove metode prikazana je na listingu 55.

public void submitScore(SubmitScoreRequest request) {  
 AppUser student = userRepository.  
 findById(jwtUtils.getCurrentUsername()));  
 Lecture lecture = lectureRepository.findById(request.getLectureID);  
 TestScore testScore = new TestScore();  
 TestScorePK pk = new TestScorePK();  
 pk.setStudent(student);  
 pk.setTest(lecture.getTest());  
 testScore.setId(pk);  
 testScore.setScore(request.getScore());  
 testScoreRepository.save(testScore);  
}

Listing 55 Metoda servisa zadužena za čuvanje broja poena u bazi

## Pregled testiranja

Obzirom na to da trenutno u našoj bazi imamo upisane ocene studenata na testovima, potrebno je te podatke i prikazati u smislenim formama. Predavačima će biti omogućen prikaz ocena jedne lekcije kroz više formi.

Prva forma jeste tabela u kojoj su redovi studenti, a kolone lekcije, pa će centar tabele biti popunjen svim ocenama studenata koji su pohađali te lekcije. Druga jeste prikaz po studentima. Predavaču će biti ponuđeno da izabere studenta čije ocene želi da pregleda i nakon klika na studenta prikazaće se ***Line Chart*** koji prikazuje ocene studenta kroz vremensku linija od prvog ka poslednjem testu. Treća forma za prikaz jeste prikaz po lekciji, profesor će izabrati za koju lekciju želi da pregleda ocene, nakon toga prikazaće mu se ***Bar Chart*** koji će prikazati koliko studenta je ostvarilo određen broj poena.

U aplikaciji ćemo kreirati novu komponentu ***test-data***. Do ove komponente ćemo se rutirati kroz prozor koji vide predavači i zatim kroz odabir kursa kom želimo da pristupimo i klikom na dugme za prikaz rezultata testova. Komponenti prosleđujemo *course* objekat kroz *History State* kako bi komponenta sadržala osnovne podatke o kursu čije ocene je potrebno da prikaže. Komponenta sadrži tri dugmeta pomoću kojih predavač može da izabere formu koju želi prikazati. Ova komponenta takođe sadrži i sve tri forme za prikaz ocena u sebi i uz pomoć ***\*ngIf*** direktive nad elementama koje prikazuju komponente će odlućiti koju formu je potrebno prikazati, kao što je prikazano na listingu 56.

<button (click)="showComp = 'Data'">Data</button>

<button (click)="showComp = 'Student'">Filter by student</button>

<button (click)="showComp = 'Lecture'">Filter by lecture</button>

<app-all-test-data \*ngIf="showComp === 'Data'"></app-all-test-data>

<app-per-student-data \*ngIf="showComp === 'Student'"></app-per-student-data>

<app-per-lecture-data \*ngIf="showComp === 'Lecture'"></app-per-lecture-data>

Listing 56 HTML fajl test-data komponente

Kako bismo dobili podatke o ocenama u konstruktoru skript fajla *test-page* komponente ćemo pozvati ***TestPageService*** koji će nam na osnovu imena kursa, kojeg smo dobili kroz *History State*, dovući sve potrebne podatke o ocenama sa *HTTP GET* pozivom ka bekendu ka putanji */test/scores/{courseName}*. Bekend će vratiti listu entiteta, od kojih će svaki sadržati: naziv lekcije, imejl studenta i ocenu koju je dobio na testu, kao što je prikazano na listingu 57. Na ovaj način filtriranje po lekciji i po studentu će nam postati mnogo lakše.

public List<CourseScoringData> getScoringData(String courseName) {  
 List<CourseScoringData> response = new ArrayList<>();  
 Course course = courseRepository.findById(courseName);  
  
 for(Lecture lecture : course.getLectures()) {  
 for(Enrollment enrollment : course.getEnrollments()) {  
 AppUser student = enrollment.getStudent();  
  
 CourseScoringData record = new CourseScoringData();  
 record.setLectureTitle(lecture.getTitle());  
 record.setStudentEmail(student.getEmail());  
  
 if(lecture.getTest() != null) {  
 testScoreRepository

.findByStudentEmailAndTestId(student.getEmail(), lecture.getTest().getId())

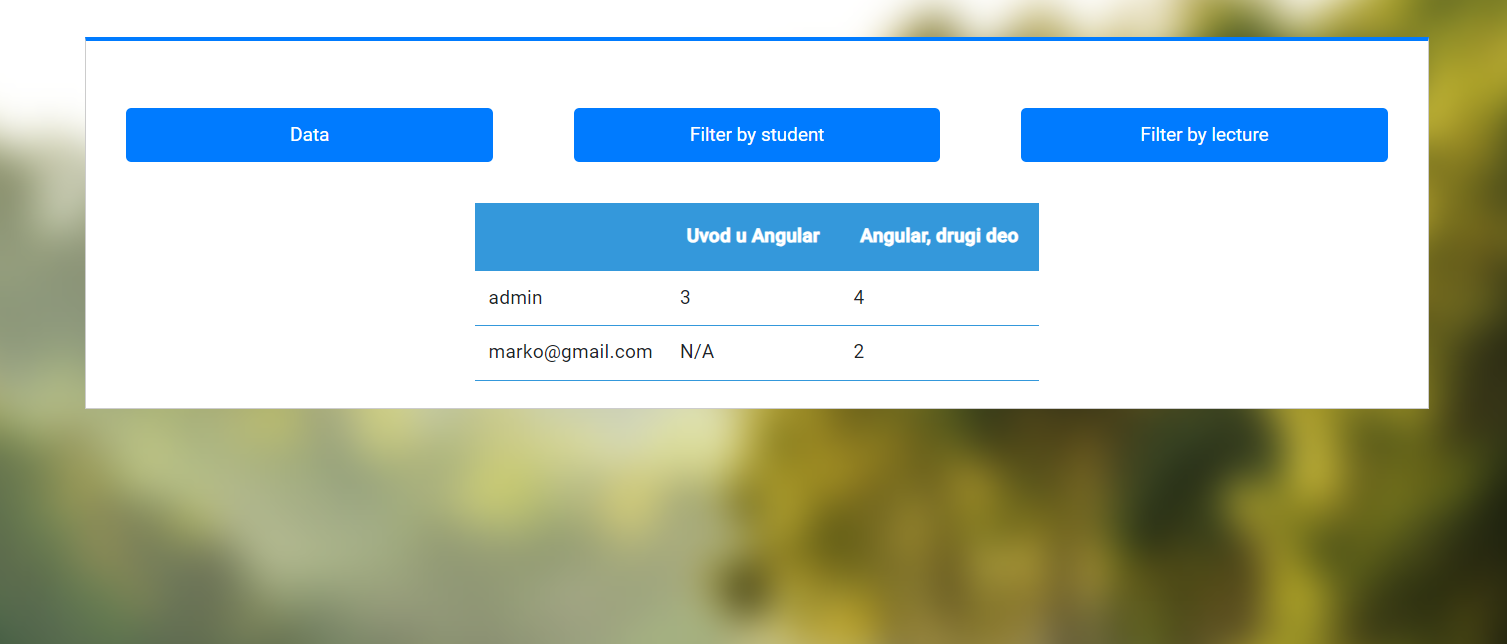
.ifPresent(testScore -> record.setScore(testScore.getScore()));  
 }  
 response.add(record);  
 }  
 }  
  
 return response;  
}

Listing 57 Metoda servisa koja vraća podatke o studentima i njihovim ocenama na kursu

Nakon dobijanja podataka na frontendu iste ćemo čuvati u poljima u ***TestDataServisu***. U *data* polju čuvamo celu listu primljenih entiteta, dok ćemo u poljima *students* čuvati set koji se sastoji od svih studenata i l*ectures* koji će se sastojati od svih lekcija tog kursa, kako bismo imali redove i kolone naših formi bez ponavljanja.

### Tabela svih podataka

Komponentu za prikaz tabele ćemo nazvati ***all-test-data***. Ova komponenta će u svom *HTML* fajlu sadržati samo *table* element koji ćemo popuniti u skript fajlu komponente. Prvi red ove tabele popunjavamo nazivima lekcija koje čuvamo u *lecture* setu u *TestDataService*, dok za ostale redove koristimo petlju koja prolazi kroz sve studente iz seta. Nakon kreiranja prvog reda gde će se prikazati nazivi lekcija kursa, u prvim kolonama ostalih redova ispisaćemo imejlove studenata. Ostale ćelije tabele popunićemo ocenama koje student iz tog reda ima iz lekcija prikazanim u tim kolonama. Ocenu pronalazimo linearnim prolaskom kroz entitete *data* polja servisa, poređujući imejl studenta i naziv lekcije koju tražimo. Na listingu 58 prikazana je metoda za kreiranje i popunjavanje ove tabele, dok je na slici 13 prikazan izgled ove tabele.



Slika 13 Prikaz tabele studenata i njegovih ocena u kursu

generateTableBody(table: HTMLTableElement) {

    let tbody = table.createTBody();

    for (const student of this.testDataService.students) {

      let row = tbody.insertRow();

      row.insertCell(0).innerHTML = student;

      for (let i = 1; i <= this.testDataService.lectures.size; i++) {

        let lectureTitle = this.getTableCell(0, i);

        let studentEmail = this.getTableCell(row.rowIndex, 0);

        let score = this.findScoreForLecture(lectureTitle, studentEmail);

        row.insertCell(i).innerHTML = score;

      }

    }

}

Listing 58 Metoda kojom popunjavamo tabelu ocenama studenata

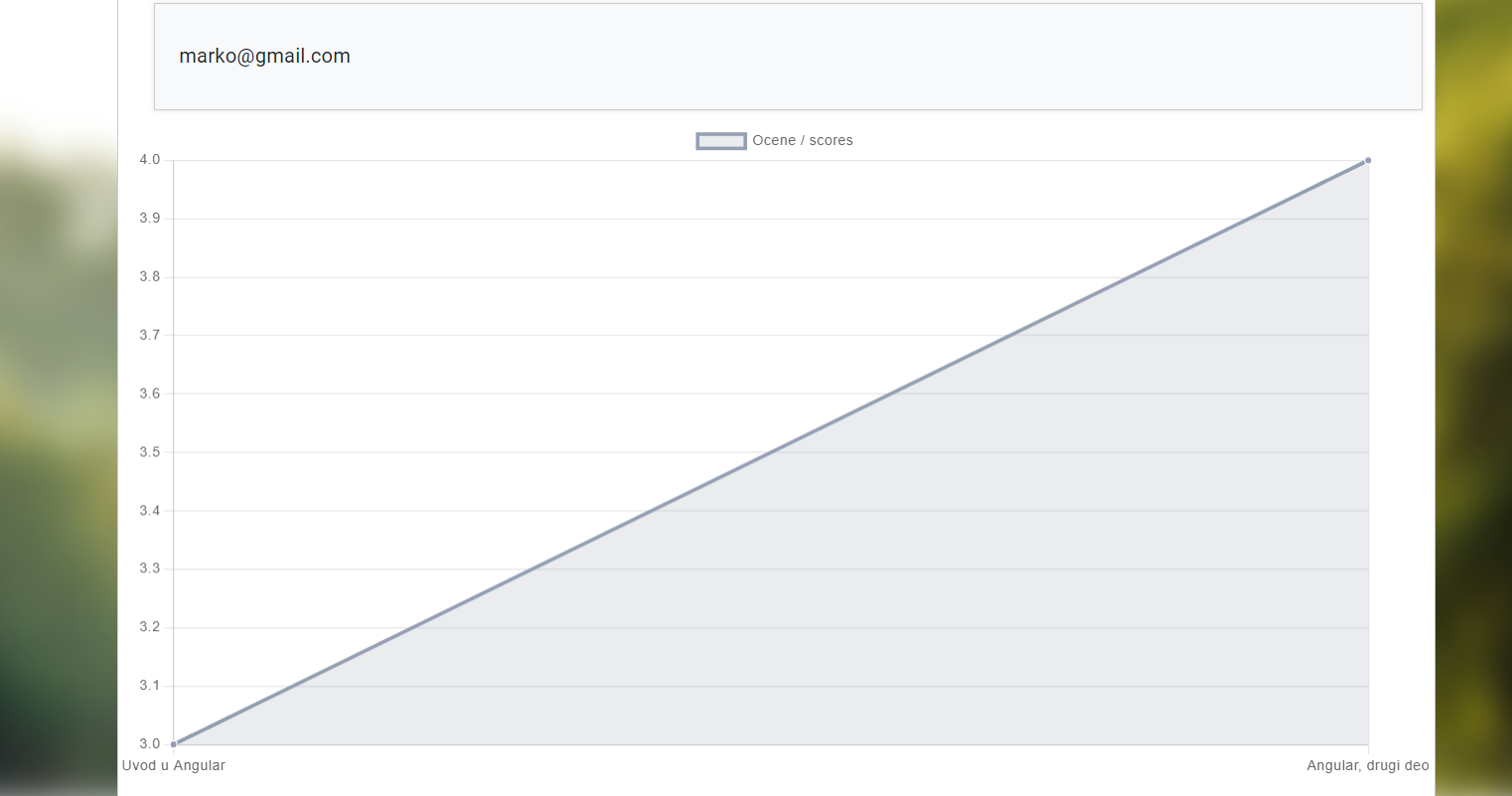
### Prikaz ocena po studentima

Jedan od načina prikazivanja ocena jeste filtriranjem prema studentima. Odabirom na studenta iz liste prikazaće se *Line Chart* koji će prikazati sve studentove ocene od početka pohađanja kursa.

Sve što je potrebno kako bismo prikazali dijagram na našoj stranici jeste da u *HTML* fajlu komponente napravimo *canvas* element uz ***baseChart*** direktivu *ng2-charts* biblioteke koja služi za povezivanje *chart.js* biblioteke i *canvas* elemenata. Direktive ***[data]*** i ***[type]*** su takođe direktive biblioteke koje povezuje podatke iz skript fajla komponente o podacima i tipu *charta* na *canvas* element. Primer kreiranja dijagramau *HTML* fajlu prikazan je na listingu 59.

<canvas baseChart [data]="lineChartData" [type]="lineChartType"></canvas>

Listing 59 HTML element pomoću kojeg prikazujemo chart iz ng2-charts biblioteke

U skript fajlu komponente ***ChartType*** polje koje je vezano za *canvas* element postavićemo na ***line*** vrednost jer želimo da naš dijagram bude linijski, dok ćemo za polje ***lineChartData*** potrebne podatke dobiti kroz ***TestDataService*** polje *data*, tako što ćemo za početak filtrirati samo podatke o izabranom studentu preko imejla, a nakon toga ćemo za podatke dijagrama uzeti ocene studenta iz filtrirane liste, a za labele sve lekcije. Proces filtriranja podataka i određivanja labela prikazan je kroz listing 60. Na slici 14 prikazan je krajnji rezultat prikazivanja ovog *charta*.

Slika 14 Prikaz filtriran prema studentu

public lineChartType: ChartType = 'line';

public lineChartData!: ChartConfiguration['data'];

  showStudentData(email: string) {

    let studentScores = this.testDataService.data.filter((lecture: any) => lecture.studentEmail === email);

    this.lineChartData = {

      datasets: [

        {

          data: studentScores.map(lecture => lecture.score),

          label: 'Ocene / scores',

        }

      ],

      labels: studentScores.map(lecture => lecture.lectureTitle),

    };

}

Listing 60 Metoda kojom postavljamo sve potrebne podatke charta

### Prikaz po lekciji

Prikaz po lekciji pomaže predavačima da izvuku bitne informacije o testu. Pomoću ***Bar Charta*** omogućeno nam je da saznamo koliko su zapravo studenti uspešno savladali test.

Za prikaz ove komponente potrebno nam je identično *canvas* polje kao kod prethodnog dijagrama, dok u skript fajlu imamo nekoliko izmena. Za početak, umesto *ChartType* polja koje je bilo postavljeno na *line*, ovog puta postavljamo na ***bar***, zato što želimo prikaz pomoću ***BarCharta***. Takođe, labele koje su nam potrebne za ovaj dijagramjesu predefinisani poeni, tj lista brojeva od nula do pet, dok nam je za podatke koje želimo da koristimo potrebna dodatna kalkulacija.

Za izračuvanje broja studenata po broju poena koristićemo *data* polje *TestData* servisa koje ćemo filtrirati i ostaviti samo entitete sa nazivom lekcije. Nakon toga kreiramo praznu listu sa šest elemenata koja pokazuje broj pojavljivanja ocena, redom od nula do pet. Prolazom kroz sve elemente filtrirane liste, povećavamo element liste za jedan za svako pojavljivanje broja poena u listi entiteta, kao što je prikazano na listingu 61. Ovo su podaci koje ćemo koristiti u našem *ChartData* prikazu. Nakon dobijenih podataka, prikaz će biti popunjen podacima, kao što možemo i videti na slici 15.

studentsPerPointsData(lectureTitle: string) {

  let testData = this.testDataService.data

.filter((data: any) => data.lectureTitle === lectureTitle);

  let data = [0, 0, 0, 0, 0, 0];

  for(const test of testData) {

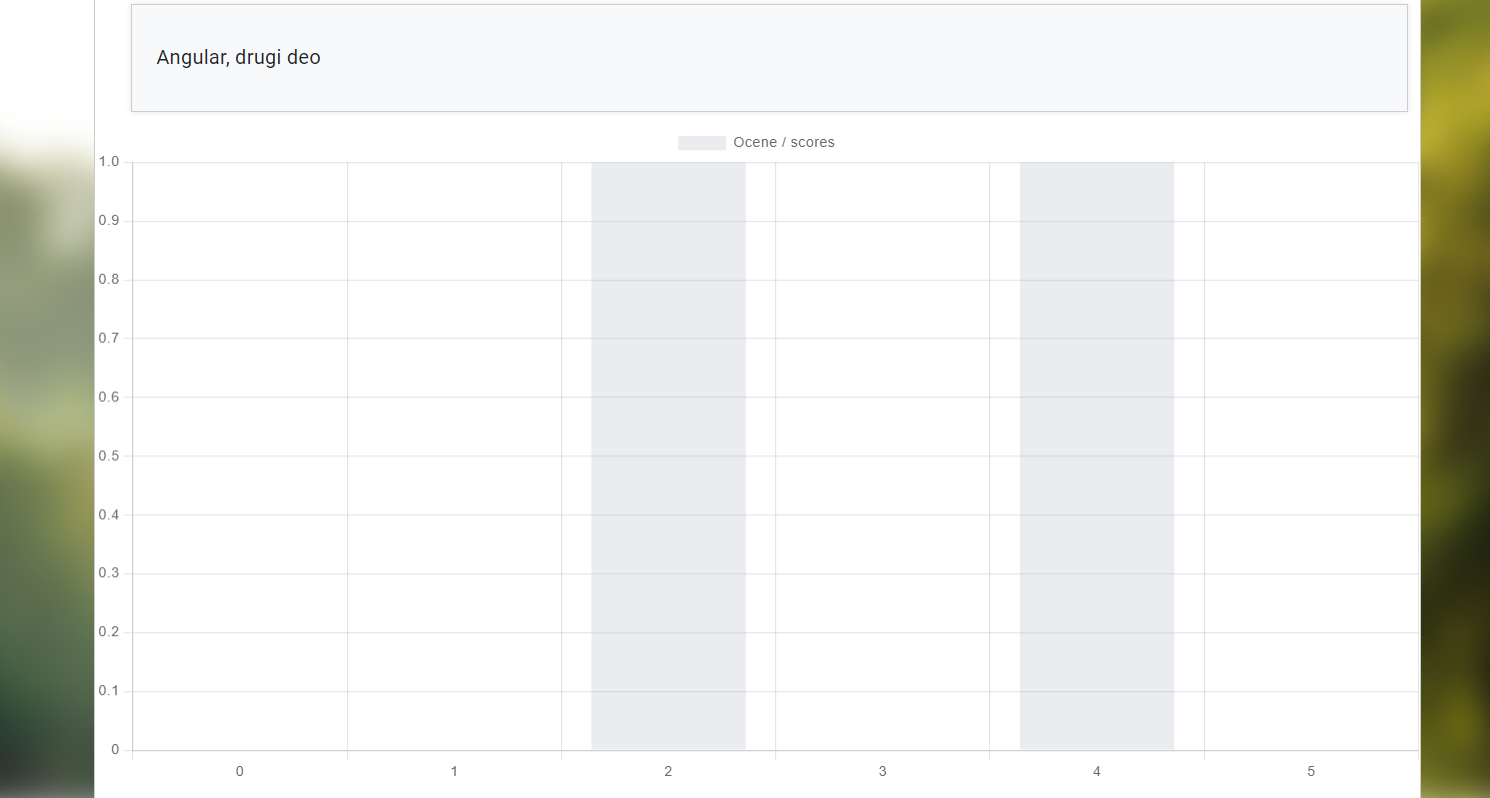
    data[test.score]++;

  }

  return data;

}

Listing 61 Prikaz kalkulacije potrebne za formiranje podataka charta



Slika 15 Prikaz dijagrama filtriranom po lekcijama

5. Zaključak

Aplikacija učenja na daljinu pomaže predavačima i studentima da evidentiraju i prate nastavu koja sa održava onlajn. Iako trenutno pruža samo osnovne funkcionalnosti za rad, njen dalji razvoj može doprineti da aplikacija postane još lakše za praćenje nastave. Neki od razvoja koji bi ovo omogućili jesu imejl notifikacije pre testa, obaveštenja predavača svojim studentima, prezentacije ili dokumenti koji pokrivaju gradivo iz lekcija, odabir jezika za prikaz funkcionalnsoti kao i još mnogo drugih opcija.

Iako aplikacija deluje kao da sadrži samo osnovne funkcionalnosti za rad, ona se ipak sastoji od više sistema, koji mogu raditi pojedinačno, ali koji zajedno komuniciraju kako bi omogućili svim korisnicima praćenje lekcija i ocenjivanje u okviru istih.

6. Literatura

1. <https://sh.wikipedia.org/wiki/Java_(programski_jezik)>
2. <https://www.geeksforgeeks.org/garbage-collection-java/>
3. <https://sh.wikipedia.org/wiki/Java_(virtualna_ma%C5%A1ina)>
4. <https://en.wikipedia.org/wiki/Create,_read,_update_and_delete>
5. <https://sh.wikipedia.org/wiki/Relacijska_baza_podataka>
6. <https://www.json.org/json-en.html>
7. <https://www.mongodb.com/basics/acid-transactions>
8. <https://en.wikipedia.org/wiki/TypeScript>
9. <https://en.wikipedia.org/wiki/AngularJS>
10. <https://www.codeblog.rs/clanci/asinhrono_programiranje_u_javascriptu>
11. <https://en.wikipedia.org/wiki/Integrated_development_environment>
12. <https://www.w3schools.com/jsref/api_history.asp>
13. <https://sr.wikipedia.org/sr/Enkripcija>
14. <https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage>