Android aplikacija za zadavanje zadataka

Štefančić, Kristijan

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:498621

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-11-27**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni diplomski studij računarstva

ANDROID APLIKACIJA ZA ZADAVANJE ZADATAKA

Diplomski rad

Kristijan Štefančić

Osijek, 2018.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za obranu diplomskog rada

Osijek, 23.09.2018.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu diplomskog rada

Ime i prezime studenta:	Kristijan Štefančić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D 853 R, 10.10.2017.
OIB studenta:	23722946048
Mentor:	Izv. prof. dr. sc. Krešimir Nenadić
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Alfonzo Baumgartner
Član Povjerenstva:	Doc.dr.sc. Tomislav Keser
Naslov diplomskog rada:	Android aplikacija za zadavanje zadataka
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Kratko opisati mogućnost korištenja Android platforme i poslužiteljskih tehnologija za zadavanje zadataka učenicima. Učenici se trebaju registrirati, preuzeti zadatke iz proizvoljnog predmeta i dostaviti rješenja na poslužitelj. Nastavnik treba imati mogućnost zadavanja zadatka za svoje predmete i točnog rješenja. Poslužiteljski dio bilježi bodove točnih odgovora i pravi rang listu. Obaviti testiranje cijelog sustava i prikazati rezultate u radu.
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 2 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 2 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	23.09.2018.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 09.10.2018.

Ime i prezime studenta:	Kristijan Štefančić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D 853 R, 10.10.2017.
Ephorus podudaranje [%]:	4%

Ovom izjavom izjavljujem da je rad pod nazivom**: Android aplikacija za zadavanje zadataka**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Krešimir Nenadić

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA

Ja, Kristijan Štefančić, OIB: 23722946048, student/ica na studiju: Diplomski sveučilišni studij Računarstvo, dajem suglasnost Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek da pohrani i javno objavi moj **diplomski rad**:

Android aplikacija za zadavanje zadataka

u javno dostupnom fakultetskom, sveučilišnom i nacionalnom repozitoriju.

Osijek, 09.10.2018.

_____

potpis

SADRŽAJ

1. UVOD

Pametni su telefoni unazad nekoliko godina postali dio naše svakodnevnice. Pametnim telefonima služe se sve generacije, a ponajviše djeca. Problemi s kojima se susreću roditelji su ti da djeca često zapostavljaju školske, a i ostale obveze upravo zbog pametnih telefona. Cilj aplikacije za zadavanje zadataka je da djeca osim za igru, pametne telefone koriste za usvajanje novih znanja i ponavljanje naučenog gradiva na njima zanimljiviji i privlačniji način.

Aplikacija za zadavanje zadataka je zapravo sustav koji učenicima omogućuje interaktivno učenje i vježbanje zadataka iz različitih školskih predmeta. Aplikacija se sastoji od dva glavna dijela. Poslužiteljska aplikacija koristeći *Spring Framework* omogućuje udaljen pristup podacima koji su pohranjeni na poslužitelju implementacijom REST (eng. *Representational State Transfer*) servisa, a Android aplikacija omogućuje korisnicima manipulaciju te korištenje podataka dobivenih s poslužitelja. Poslužiteljska aplikacija se također brine i o autorizaciji korisnika, ograničavanju pristupa podacima te upravlja bazom podataka. Android aplikacija na temelju tipa korisnika (učenik ili učitelj) dinamički stvara sučelje. Učitelji će imati mogućnost zadavanja zadataka za svoj predmet, pregled svih objavljenih zadataka vezanih za predmete koje predaju te uvid u rang liste učenika. Učenici će moći rješavati zadatke zadane od strane učitelja, vidjeti rang liste učenika na temelju različitih filtera (škola, grad, županija...) i svoju poziciju na ukupnoj rang listi svih učenika.

U ovom diplomskom radu je opisan način izrade aplikacije za zadavanje zadataka, korištene tehnologije na poslužiteljskoj i klijentskoj strani s primjerima i objašnjenjem koda te načinom testiranja i uputama za korištenje aplikacije.

1.1. Zadatak završnog rada

Kratko opisati mogućnost korištenja Android platforme i poslužiteljskih tehnologija za zadavanje zadataka učenicima. Učenici se trebaju registrirati, preuzeti zadatke iz proizvoljnog predmeta i dostaviti rješenja na poslužitelj. Učitelj treba imati mogućnost zadavanja zadatka za svoje predmete i točnog rješenja. Poslužiteljski dio bilježi bodove točnih odgovora i pravi rang listu. Obaviti testiranje cijelog sustava i prikazati rezultate u radu.

2. OPIS KORIŠTENIH TEHNOLOGIJA

Tehnologije korištene za izradu sustava za zadavanje zadataka su *MySQL* baza podataka (eng. *database*) koja služi kao spremnik podataka, *Spring Framework* se koristi kao glavna tehnologija za izradu poslužiteljske aplikacije te Android Studio razvojno okruženje za izradu mobilne aplikacije.

2.1. *Spring Framework*

Spring Framework je aplikacijski okvir (eng. *application framework*) koji se ponaša kao glavni spremnik inverzije kontrole (eng. *Inversion of Control (IoC) container*) za Java platformu.[1] Osnovne značajke (eng. *features*) ovog aplikacijskog okvira mogu se koristiti u svim Java aplikacijama, ali se najčešće koriste kao alat za izradu web aplikacija na temelju Java EE (*Enterprise Edition*) platforme. *Spring Framework* pruža širok spektar mogućnosti:

- Aspektno-orijentirano programiranje (eng. *Aspect-oriented programming*)
- Ubrizgavanje ovisnosti (eng. *dependency injection*)
- Autentifikaciju i autorizaciju korisnika pomoću *Spring Security* projekta
- Pristup i obrada podataka
- Slanje i primanje poruka
- MVC (eng. *Model-View-Controller*) programska paradigma
- Stvaranje web servisa i njihovo upravljanje - SOAP (eng. *Simple Object Access Protocol*) i REST (eng. *Representational State Transfer*) servisi
- Upravljanje transakcijama
- Testiranje

2.1.1. *Spring Boot*

U poslužiteljskoj aplikaciji sustava za zadavanje zadataka koristi se *Spring Boot* implementacija *Spring Frameworka* koja omogućuje jednostavno stvaranje samostalnih Java aplikacija koje je dovoljno pokrenuti na bilo kojem računalu. Takve Java aplikacije posjeduju

ugrađeni poslužitelj (eng. *embedded server*) i potrebno je vrlo malo *Spring* konfiguracije za njihovo stvaranje i ispravan rad. Najvažnije mogućnosti koje pruža *Spring Boot* su:

- stvaranje samostalnih (eng. *stand alone*) aplikacija
- ugrađeni *Tomcat, Jetty* ili *Undertow* poslužitelj (bez potrebe .*war* datoteka)
- pruža pojednostavljene početne (eng. *starter*) ovisnosti za jednostavniju konfiguraciju
- automatski konfigurira *Spring* i vanjske biblioteke (eng. *third party libraries*)
- ne postoji generiranje koda i nije obavezno pisati XML konfiguraciju [2]

2.1.2. *Spring Data* (JPA)

Za pristupanje *MySQL* bazi podataka na poslužitelju, konfiguraciju JDBC konektora, mapiranje entiteta i tablica koristi se *Spring Data* modul. On omogućava jednostavno korištenje tehnologija za pristupanje podacima, relacijskih i nerelacijskih baza podataka, okvira za mapiranje objekata i servisa temeljenih na oblaku (eng. *cloud services*). Osnovne mogućnosti koje omogućuje *Spring Data* modul su:

- apstraktno mapiranje objekata (entiteta) i tablica baze podataka unutar repozitorija
- dinamičko kreiranje SQL upita na temelju imena metoda repozitorija
- jednostavna integracija s ostalim *Spring* modulima pomoću Java konfiguracije i XML imenika
- napredna integracija sa *SpringMVC* kontrolerima [3]

2.1.3. *Spring Security*

Spring Security je vrlo moćan i visoko prilagodljiv programski okvir za autentifikaciju i kontrolu pristupa. Postao je temeljni standard za osiguravanje aplikacija temeljenih na *Spring Fraweworku*. Osnovne značajke ovog programskog okvira su:

- sveobuhvatna i proširiva podrška za autentifikaciju i autorizaciju
- zaštita od napada poput „fiksacije sesije“ (eng. *session fixation*), „*clickjacking*“, zahtjeva za lažne stranice (eng. *cross site request forgery*)

- integracija sa *servlet* API-jima
- moguća integracija sa *Spring Web MVC* modulom [4]

2.2. Android MVC uzorak izrade aplikacije

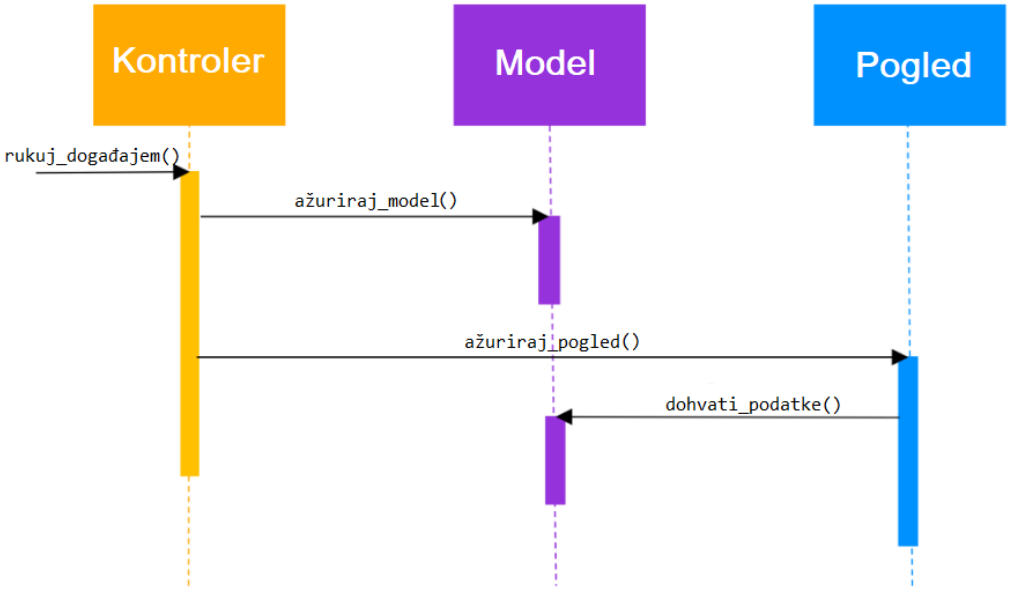
U današnje vrijeme logika korisničkog sučelja prilikom izrade Android aplikacija se mijenja češće nego poslovna logika pa je iz tog razloga, kao i kod poslužitelja, osmišljen MVC programski uzorak za odvajanje poslovne logike od korisničkog sučelja:

- Model – sloj podataka koji je odgovoran za upravljanje poslovnom logikom i rukovanjem mrežnim API-jima
- Pogled (eng. *View*) – sloj korisničkog sučelja tj. reprezentacija podataka iz modela
- Kontroler (eng. *Controller*) – sloj na kojem se odrađuje poslovna logika

Postoje dva osnovna tipa MVC programskog uzorka:

1. Pasivni model – kontroler je samo klasa koja manipulira modelom na temelju korisničkih akcija. Nakon ažuriranja ili promjene podataka unutar modela, kontroler mora obavijestiti pogled da se i on mora ažurirati te će u tom trenutku pogled zatražiti podatke od modela.
2. Aktivni model – ukoliko kontroler nije jedina klasa koja mijenja podatke u modelu, mora postojati alternativni način za obavještavanje pogleda (i drugih klasa) o promjeni modela. Ovo se postiže programskim uzorkom „promatrač“ (eng. *Observer pattern*). Model sadrži kolekciju promatrača koji prate promjene u podacima, a pogledi implementiraju promatračka sučelja i na taj način i oni promatraju promjene u modelu. [5]

Android aplikacija za zadavanje zadataka je napravljena na temelju pasivnog modela. Aktivnosti i fragmenti se ponašaju kao kontroleri, oni manipuliraju modelima te obavještavaju različite poglede o promjenama podataka.



Sl. 2.1. *Dijagram toka pasivnog MVC modela*

3. ANDROID APLIKACIJA ZA ZADAVANJE ZADATAKA

Android aplikacija za zadavanje zadataka zapravo je sustav koji radi na principu klijent-poslužitelj. Poslužitelja predstavlja *Spring Boot* aplikacija koja se može izvršavati na bilo kojem računalu s instaliranom Javom, ona prilikom pokretanja konfigurira i pokreće ugrađeni server (eng. *embedded server*) kojem je moguće pristupati udaljeno. *Spring Boot* aplikacija se temelji na MVC programskoj paradigmi, a komunikacija s klijentom ostvaruje se pomoću REST servisa, ona se također brine o autorizaciji i autentifikaciji korisnika, pruža servise za komunikaciju s repozitorijima (eng. *repository*) koji upravljaju podacima u bazi podataka. Klijenti poslužitelju pristupaju pomoću Android aplikacije koja korisnicima omogućava registraciju i prijavu u sustav. Nakon prijave u sustav, na temelju tipa korisnika dinamički se izrađuje korisničko sučelje te se definiraju dozvoljene funkcionalnosti koje korisnik može upotrebljavati:

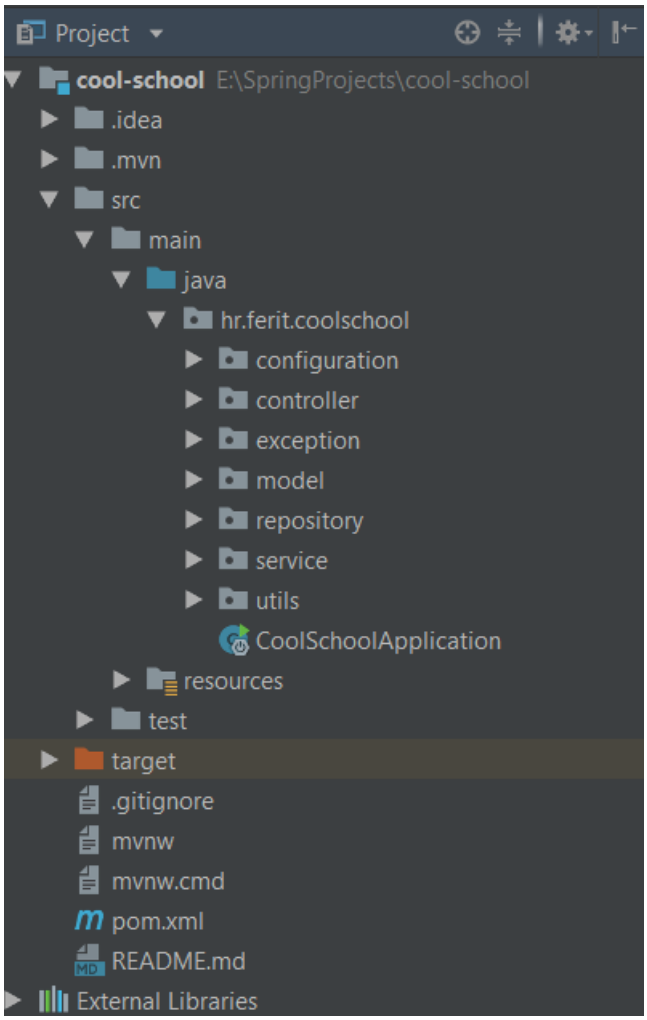
- Svi korisnici mogu s poslužitelja preuzeti rang liste učenika na temelju odabranih filtera te uređivati informacije na svom profilu
- Učitelji mogu kreirati zadatke u obliku kvizova te ih slanjem na poslužitelja spremiti u bazu podataka
- Učenici mogu rješavati kvizove za bilo koji predmet te tako ocijeniti svoje znanje

3.1. Poslužiteljska aplikacija

Poslužiteljska (*Spring Boot*) aplikacija ponaša se kao servis koji omogućuje udaljen pristup, manipulaciju podacima, njihovo skladištenje i dohvaćanje iz baze podataka. *Spring Boot* aplikacija temelji se na nekoliko modula (paketa) ili podsustava:

- Konfiguracija (eng. *Configuration*) – Java klase, *.xml* i *.properties* datoteke unutar kojih se nalaze naredbe za konfiguraciju različitih komponenti i modula (baze podataka, ugrađeni poslužitelj, sigurnost, autorizacija korisnika, logova (eng. *logs*) itd.) prije i tijekom izvođenja aplikacije
- Modeli (eng. *Models*) – Java klase koje služe za reprezentaciju podataka kojima aplikacija manipulira

- Kontroleri (eng. *Controllers*) – Java klase unutar kojih se definiraju HTTP zahtjevi za pristup servisima
- Servisi (eng. *Services*) – Java klase unutar kojih se implementira poslovna logika (eng. *business logic*) koja će se primjenjivati u manipulaciji podacima
- Repozitoriji (eng. *Repositories*) – Java sučelja koja uz pomoć JPA (eng. *Java Persistance API*) omogućavaju operacije umetanja, ažuriranja i brisanja podataka unutar baze te njihovo dohvaćanje



Sl. 3.1. *Struktura projekta poslužiteljske aplikacije*

3.1.1. Sigurnost (autentifikacija i autorizacija)

Za sigurnost podataka brine se *Spring Security* koji implementira autorizaciju korisnika te prilikom svakog HTTP zahtjeva provjerava je li korisnik autoriziran za pristupanje traženom servisu na temelju konfiguracije za autorizaciju zahtjeva. Cjelokupna konfiguracija autorizacije i autentifikacije nalazi se u klasi „*SecurityConfig*“ koja nasljeđuje klasu „*WebSecurityConfigurerAdapter*“. Postoji nekoliko načina konfiguracije, a jedna od najčešće korištenih je konfiguracija u kojoj se nadjačavaju (eng. *override*) metode „*configure(AuthenticationManagerBuilder auth)*“ i „*configure(HttpSecurity http)*“ iz naslijeđene klase. U prvoj metodi se definira instanca servisa unutar kojeg se na temelju korisničkog imena i lozinke, dobivenih iz zahtjeva za prijavu, korisnik može autentificirati i preuzeti svoje podatke iz baze podataka. Uz instancu servisa za autentifikaciju moguće je predati instancu klase za enkripciju zaporke. U drugoj metodi konfigurira se autorizacija za pojedine HTTP zahtjeve i putanje.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .antMatchers(
                    ...antPatterns: "/login",
                    "/api/users/registration",
                    "/api/schools").permitAll()
            .antMatchers( ...antPatterns: "/api/quiz-results/**/submit/**").hasAuthority(Role.ROLE_STUDENT.toString())
            .antMatchers(HttpMethod.POST,  ...antPatterns: "/api/quizzes").hasAuthority(Role.ROLE_TEACHER.toString())
            .antMatchers(HttpMethod.PUT,  ...antPatterns: "/api/quizzes/**").hasAuthority(Role.ROLE_TEACHER.toString())
            .antMatchers(HttpMethod.DELETE,  ...antPatterns: "/api/quizzes/**").hasAuthority(Role.ROLE_TEACHER.toString())
            .antMatchers(HttpMethod.POST,  ...antPatterns: "/api/schools").hasAuthority(Role.ROLE_ADMIN.toString())
            .antMatchers(HttpMethod.PUT,  ...antPatterns: "/api/schools/**").hasAuthority(Role.ROLE_ADMIN.toString())
            .antMatchers(HttpMethod.DELETE,  ...antPatterns: "/api/schools/**").hasAuthority(Role.ROLE_ADMIN.toString())
            .antMatchers(HttpMethod.DELETE,  ...antPatterns: "/api/users/**").hasAuthority(Role.ROLE_ADMIN.toString())
            .anyRequest().authenticated()
```

Sl. 3.2. *„configure()“ metoda za konfiguraciju autorizacije*

Kao što je vidljivo na slici 3.2. metoda „*antMatchers()*“, koja kao parametre prima ili krajnje točke (eng. *endpoints*) u obliku teksta ili tip HTTP zahtjeva(POST, PUT, DELETE, GET, PATCH...) i krajnju točku, u kombinaciji s metodom „*hasAuthority()*“, koja kao parametar prima autoritet (eng. *authority*), stvaraju konfiguraciju za autorizaciju specifičnih zahtjeva. Metodom „*permitAll()*“ se dopuštaju zahtjevi na krajnje točke definirane u prethodnoj metodi „*antMatchers()*“. Dodavanjem „*anyRequest().authenticated()*“ konfigurira se da je za bilo koju drugu krajnju točku, koja nije navedena u konfiguraciji, potrebna autentifikacija korisnika, ali se autorizacija ne provjerava.

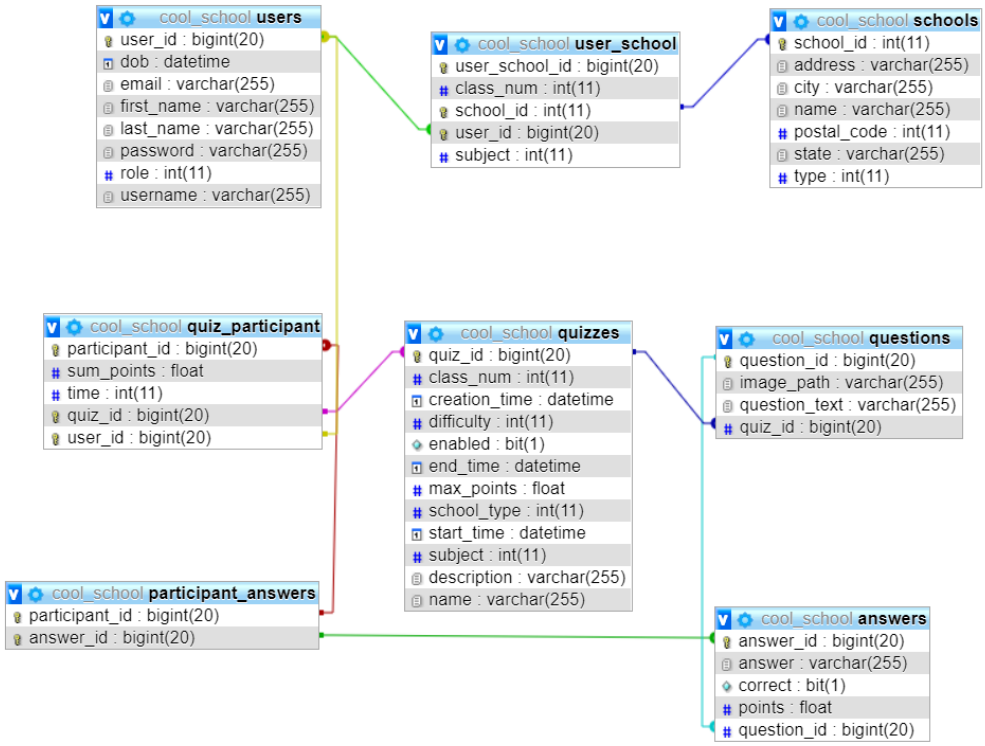
3.1.2. Baza podataka i entiteti

Spring Boot i JPA omogućuju kreiranje i mapiranje entiteta (eng. *entity mapping*) tj. Java klasa s tablicama baze podataka. Dovoljno je kreirati modele i svojstvima (eng. property) pridružiti određene anotacije kako bi JPA na temelju njih mogao kreirati i mapirati tablice baze podataka. Sama baza podataka se sastoji od pet glavnih tablica:

1. „*users*“ – sadrži osnovne podatke o korisnicima (korisničko ime, lozinku, ime, prezime itd.). Svaki korisnik može imati samo jednu ulogu (eng. *role*) („*ROLE_STUDENT*“ ili „*ROLE_TEACHER*“), a može pohađati ili predavati u više škola.
2. „*schools*“ – sadrži osnovne podatke o školama (adresa, ime, tip itd.), a svaka škola može imati više korisnika (učitelja/ica ili učenika/ca).
3. „*quizzes*“ – tablica koja predstavlja određenu grupu pitanja odnosno kviz. Svaki kviz ima određenu težinu, ukupan broj bodova, predmet na koji se odnosi (fizika, matematika, hrvatski...), tip škole na koji se odnosi (osnovna škola, srednja škola ili fakultet) te razred za koji je kviz preporučen. Ova tablica sadrži i osnove podatke o kvizovima (naslov i opis kviza, datum početka i završetka itd.)
4. „*questions*“ – sadrži podatke o pitanjima, a svako pitanje treba imati više odgovora od kojih je samo jedan točan.
5. „*answers*“ – sadrži podatke o odgovoru (tekst odgovora, je li odgovor točan u skupini odgovora za određeno pitanje te broj bodova koji nosi točan odgovor).

Uz glavne tablice zbog veze više naprema više dodane su još tri tablice:

1. „*user_schools*“ – tablica koja predstavlja vezu više naprema više između škola i korisnika. Podatak „*class_num*“ označava razred koji učenik/ca (korisnik s ulogom „*ROLE_STUDENT*“) pohađa u školi, a „*subject*“ označava predmet koji učitelj/ica (korisnik s ulogom „*ROLE_TEACHER*“) predaje u školi.
2. „*quiz_participant*“ – tablica koja predstavlja sudionika u kvizu jer svaki kviz može biti riješen od više korisnika. Kvizove mogu rješavati samo korisnici s ulogom „*ROLE_STUDENT*". Ova tablica sadrži podatke o vremenu rješavanja kviza i broju skupljenih bodova na kvizu.
3. „*quiz_participant_answers*“ – tablica za spremanje predanih odgovora korisnika



Sl. 3.3. *Tablice unutar baze podataka i njihove relacije*

Kao što je već spomenuto, za mapiranje entiteta s tablicama baze podataka ključne su anotacije koje se pridružuju samim entitetima i njihovim svojstvima. Kako bi neka klasa bila predstavljena kao entitet potrebno je iznad definicije klase dodati anotaciju „*@Entity*“, a uz nju za konfiguraciju imena tablice u bazi ili jedinstvenog ključa (eng. *unique constraint*) mogu se dodati i anotacije „*@Table*“ i „*@UniqueConstraint*“ kao što je prikazano na slici 3.4.

```
@Entity
@Table(name = "quiz_participant", uniqueConstraints = {
        @UniqueConstraint(columnNames = {"quiz_id", "user_id"})
})
public class QuizParticipant {
```

Sl. 3.4. *Primjer definiranja entiteta*

Za definiranje primarnog ključa (eng. *primary key*) tablice potrebno je iznad svojstva koje će se promatrati kao primarni ključ postaviti anotaciju „*@Id*“, a za automatsko generiranje identifikacijskog broja (*ID*-a) koristi se anotacija „*@GeneratedValue*“. Za definiranje relacija između entiteta, a samim time i tablica unutar baze podataka koriste se sljedeće anotacije:

1. „*@OneToOne*“ – označava relaciju jedan naprema jedan (1-1). Potrebno je u oba entiteta definirati ovu anotaciju za svojstvo koje predstavlja pripadajući entitet.
2. „*@ManyToOne*“ – označava relaciju više naprema jedan (N-1). Ukoliko se u entitetu svojstvo definira s ovom anotacijom tada to svojstvo predstavlja entitet sa strane jedan (1), a u entitet na strani jedan (1) potrebno je dodati anotaciju „*@OneToMany*“ na svojstvo koje predstavlja listu entiteta sa strane više (N). Uz ovu anotaciju potrebno je dodati i anotaciju „*@JoinColumn*“ koja kao parametar prima ime svojstva na temelju kojeg se stvara relacija.
3. „*@OneToMany*“ – označava relaciju jedan naprema više (1-N). Ukoliko se u entitetu svojstvo definira s ovom anotacijom tada je to svojstvo lista entiteta sa strane više (N), a u entitet na strani više (N) potrebno je dodati anotaciju „*@ManyToOne*“ na svojstvo koje predstavlja entitet sa strane jedan (1).
4. „*@ManyToMany*“ – označava relaciju više naprema više (N-N). Potrebno je dodati anotaciju na svojstva u oba entiteta. Uz ovu anotaciju potrebno je dodati i anotaciju „*@JoinTable*“ u kojoj se konfigurira relacija za povezivanje dva entiteta. Parametar „*name*“ predstavlja ime tablice koja povezuje entitete, a „*joinColumns*“ i „*inverseJoinColumns*“ predstavljaju imena svojstava koji će biti strani ključevi (eng. *foreign keys*) u novoj tablici.

```java
@Entity
@Table(name = "quiz_participant", uniqueConstraints = {
        @UniqueConstraint(columnNames = {"quiz_id", "user_id"})
})
public class QuizParticipant {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long participantId;
    private Float sumPoints;
    //time in seconds
    private Integer time;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;

    @ManyToOne
    @JoinColumn(name = "quiz_id")
    private Quiz quiz;

    @ManyToMany(cascade = CascadeType.MERGE)
    @JoinTable(
            name = "participant_answers",
            joinColumns = {@JoinColumn(name = "participant_id")},
            inverseJoinColumns = {@JoinColumn(name = "answer_id")}
    )
    private Set<Answer> participantAnswers;
```

Sl. 3.5. *Primjer entiteta s relacijama N-1 i N-N*

Postoji slučaj kada se ručno mora stvoriti entitet koji predstavlja tablicu za povezivanje entiteta s više naprema više (N-N) relacijom, a to je kada su u toj tablici uz primarne i strane ključeve potrebna dodatna svojstva. Primjer takvog slučaja upravo je entitet prikazan na slici 3.5. - „*QuizParticipant*“, koji predstavlja vezu između korisnika i kviza. U osnovi ovaj entitet ne bi ni postojao, bilo bi dovoljno u „*User*“ i „*Quizzes*“ entitete dodati „*@ManyToMany*“ anotaciju, ali zbog postojanja dodatnih svojstava („*sum_points*“ i „*time*“) ručno je stvoren ovaj entitet s relacijama više naprema jedan (N-1) prema korisnicima i kvizovima. Isto situacija se javlja i za entitet „*UserSchool*“ koji uz ključeve sadrži podatke o razredu koji korisnik pohađa ili predmet koji korisnik predaje.

3.1.3. Repozitoriji

Kako je spomenuto u uvodnom dijelu poglavlja, repozitoriji su sučelja izvedena iz „*JpaRepository*“ sučelja i služe za komunikaciju s bazom podataka. U „*JpaRepository*“ sučelju definirane su glavne metode za manipulaciju bazom podataka:

- „*save(T entity)*“ – služi za umetanje novih ili ažuriranje postojećih podataka (metoda vraća novi ili ažurirani objekt)
- „*findAllBy[Property1AndProperty2...](T... properties)*“ – služi za dobivanje liste objekata na temelju predanih svojstava, npr. metoda definirana u „*SchoolRepository*“ - *findAllByCity(String city)* će vratiti listu škola koje se nalaze u gradu predanom kao parametar u metodi.
- „*findBy[Property1AndProperty2...](T... properties)*“ – služi za dobivanje jednog objekta koji odgovara predanim svojstvima (može biti i jedno svojstvo). Za ovu metodu je važno da je rezultat upita jedan objekt (ili *NULL*), jer se inače javi greška.
- „*delete(T entity)*“ – služi za brisanje entiteta iz baze podataka, uz nju se mogu koristiti i metode „*deleteById(T entityId)“* te delete metode tipa gore navedenih „*find...*“ metoda – „*deleteAllByProperties()*“ i „*deleteByProperties()*“.

Ukoliko je potrebno koristiti naprednije tj. upite koji nisu definirani u „*JpaRepository*“ sučelju, dovoljno je iznad metode u repozitoriju dodati anotaciju „*@Query*“ unutar koje se kao parametar predaje SQL upit. Ako upit služi za mijenjanje podataka unutar baze potrebno je dodati i anotaciju „*@Modifying*“.

```java
public interface UserSchoolRepository extends JpaRepository<UserSchool, Long> {

    List<UserSchool> findAllBySchoolSchoolIdAndUserRole(Integer schoolId, Role role);

    List<UserSchool> findAllBySchoolSchoolId(Integer schoolId);

    @Modifying
    @Query("DELETE FROM UserSchool WHERE user_id = :userId")
    void deleteAllByUserUserId(@Param("userId") Long userId);
}
```

Sl. 3.6. *Primjer repozitorija – „UserSchoolRepository“*

Na slici 3.6. je prikazano „*UserSchoolRepository*“ sučelje koje nasljeđuje „*JpaRepository<UserSchool, Long>*“ sučelje, „*UserSchool*“ klasa koja je predana u sučelje predstavlja entitet kojim se služi ovaj repozitorij, a „*Long*“ klasa predstavlja tip primarnog ključa. Metoda „*findAllBySchoolSchoolIdAndUserRole()*“ se pomoću JPA interpretira kao upit koji glasi ovako: „*SELECT * FROM user_school us, users u, schools s WHERE us.user_id = u.user_id AND us.school_id = s.school_id AND s.school_id = {schoolId_value} AND u.role = {role_value}*“. Metoda „*deleteAllByUserUserId()*“ izvršava upit predan u „*@Query*“ anotaciji s time da se parametar unutar upita „*:userId*“ za vrijeme izvršavanja zamijeni s vrijednosti predanog „*userId*“ parametra u metodi (da bi se to omogućilo potrebno je definirati anotaciju „*@Param("ime_parametra_u_upitu")*“. Postoji specifičan slučaj kod kojega rezultat upita nije entitet iz baze podataka, već mješavina svojstava različitih entiteta, tada se koriste tzv. izvorni upiti (eng. *native queries*). Rezultat se mapira na temelju anotacija definiranih u klasi koja će predstavljati rezultat upita. Primjer takvog načina rada je upit za rang listu učenika po osvojenim bodovima, kreirana je klasa „*Rank*“ unutar koje je definiran izvorni upit te način mapiranja rezultata, također je kreiran repozitorij unutar kojeg se nalazi metoda za pozivanje izvornog upita kao što je prikazano na slikama 3.7. i 3.8.

```java
@SqlResultSetMapping(
        name = "findAllRankings",
        classes = {
                @ConstructorResult(
                        targetClass = Rank.class,
                        columns = {
                                @ColumnResult(name = "username", type = String.class),
                                @ColumnResult(name = "points", type = float.class),
                                @ColumnResult(name = "schoolName", type = String.class),
                                @ColumnResult(name = "classNum", type = int.class)
                        })
        })
@NamedNativeQuery(name = "findRankings",
        query = "SELECT u.username username, sum(qp.sum_points) points, " +
                "us.class_num classNum, s.name schoolName " +
                "FROM users u, quiz_participant qp, user_school us, schools s, quizzes q " +
                "WHERE qp.user_id = u.user_id AND us.user_id = u.user_id " +
                "AND q.quiz_id = qp.quiz_id AND s.school_id = us.school_id " +
                "AND q.quiz_id = COALESCE(:quizId,q.quiz_id) " +
                "AND q.subject = COALESCE(:subject, q.subject) " +
                "AND s.school_id = COALESCE(:schoolId, s.school_id) " +
                "AND s.city = COALESCE(:city, s.city) " +
                "AND s.state = COALESCE(:state, s.state) " +
                "AND s.type = COALESCE(:schoolType, s.type) " +
                "GROUP BY u.username ORDER BY points desc ",
        resultClass = Rank.class,
        resultSetMapping = "findAllRankings"
)
@Entity
public class Rank {

    @Id
    private Long id;
    private String username;
    private float points;
    private String schoolName;
    private int classNum;
```

Sl. 3.7. *„Rank“ klasa s definiranim izvornim upitom i načinom mapiranja rezultata upita*

```java
public interface RankingRepository extends JpaRepository<Rank, Long> {

    @Query(nativeQuery = true, name = "findRankings")
    List<Rank> findRankings(
            @Param("city") String city,
            @Param("quizId") Long quizId,
            @Param("schoolId") Integer schoolId,
            @Param("state") String state,
            @Param("subject") Integer subject,
            @Param("schoolType") Integer schoolType
    );
}
```

Sl. 3.8. *„RankingRepository“* sučelje s metodom za izvođenje definiranog izvornog upita

3.1.4. Servisi

Servisi su Java klase unutar kojih bi se trebala nalaziti sva poslovna logika za manipulaciju podacima. Svaki servis potrebno je označiti anotacijom „*@Service*“ kako bi ga *Spring Boot* mogao definirati kao *bean* tj. objekt kreiran od strane *Spring Framework* spremnika za inverziju kontrole. *Spring Framework* nudi vrlo jednostavnu implementaciju upravljanja transakcijama unutar projekta (ukoliko se koriste Spring komponente poput JPA), dovoljno je dodati anotaciju „*@Transactional*“ iznad servisa. Servisi tijekom i nakon manipulacije podacima najčešće koriste repozitorije kako bi dohvatili podatke iz baze ili ih spremili u bazu. *Spring Framework* definicijom *beanova* omogućava ubrizgavanje ovisnosti, tj. svaki *bean* predstavlja jednu instancu objekta koji se može ubrizgavati u druge klase čime se smanjuje mogućnost zauzimanja velike količine memorije. Ubrizgavanje *beana* u klasu radi se pomoću anotacije „*@Autowired*“ iznad svojstva koje predstavlja ubrizgani *bean*.
Prilikom konfiguracije autorizacije je spomenut servis koji na temelju primljenih podataka (korisničkog imena i lozinke) iz zahtjeva za autentifikaciju, dohvaća podatke o korisniku iz baze te ih vraća klijentu. Spomenuti servis prikazan je na slici 3.9.

```
@Transactional
@Service
public class UserServiceImpl implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Optional<User> user = this.userRepository.findByUsername(username);
        if (!user.isPresent()) {
            throw new UsernameNotFoundException("Korisnik ne postoji");
        }

        return new org.springframework.security.core.userdetails.User(
                user.get().getUsername(),
                user.get().getPassword(),
                Collections.singletonList(new SimpleGrantedAuthority(user.get().getRole().toString()))
        );
    }
```

Sl. 3.9. *„UserServiceImpl*“ klasa s metodom „*loadUserByUsername()*"

Metoda „*loadUserByUsername()*“ predstavlja implementaciju istoimene metode unutar *Spring Framework* sučelja „*UserDetailsService*“. Metoda pozivom „*UserRepository*“ metode „*findByUsername()*“ dohvaća podatke o korisniku iz baze podataka na temelju njegovog korisničkog imena. Za daljnju autorizaciju potreban je „*UserDetails*“ objekt koji se kreira od

podataka dobivenih iz baze podataka. Svaki „*UserDetails*“ objekt ima definirano korisničko ime, lozinku i kolekciju (eng. *collection*) *„SimpleGrantedAuthority*“ objekata unutar kojih se najčešće nalaze korisničke uloge i/ili dopuštenja (eng. *permissions*).

Prije skoro svake operacije dohvaćanja podataka iz baze ili spremanja podataka u bazu potrebno je izvršiti određene radnje nad podacima. Npr. pri dodavanju novog korisnika, unutar servisa se može provjeriti je li korisničko ime već zauzeto i ako jest servis baca iznimku (eng. *exception*) čiju poruku korisnik na klijentskoj strani može vidjeti. Uz provjeru korisničkog imena potrebno je upravljati dodavanjem škole, koju korisnik pohađa ili u kojoj se školuje, u bazu podataka. Cijela metoda za spremanje korisnika prikazana je na slici 3.10.

```
@Override
public User save(User user) {
    if (!this.userRepository.findByUsername(user.getUsername()).isPresent()) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        User savedUser = this.userRepository.save(user);
        if (user.getUserSchools().size() > 0) {
            user.getUserSchools().forEach(
                    us -> {
                        us.setUser(savedUser);
                        us.setSchool(this.schoolRepository.findById(us.getSchool().getSchoolId())
                                .orElseThrow(() -> new ResourceNotFoundException("Tražena škola ne postoji")));
                        this.userSchoolRepository.save(us);
                    }
            );
        }
        return this.userRepository.findById(savedUser.getUserId())
                .orElseThrow(() -> new RuntimeException("Nešto je pošlo po zlu"));
    }
    throw new UserAlreadyExistsException("Korisnik s odabranim korisničkim imenom već postoji");
}
```

Sl. 3.10. *„save()“ metoda unutar „UserServiceImpl“ klase*

Unutar servisa „*QuizServiceImpl*“ nalaze se metode za dohvaćanje kvizova na temelju predanih filtera, metoda za spremanje novog kviza te ažuriranje starog. Prilikom dodavanja i ažuriranja kviza bitno je napraviti provjeru strukture kviza, a trebaju vrijediti sljedeća pravila:

1. Ukupan broj bodova kviza mora odgovarati sumi bodova za pojedine točne odgovore unutar kviza
2. Svako pitanje u kvizu mora imati samo jedan točan odgovor

Metoda koja se brine da ta pravila vrijede je prikazana na slici 3.11.

```
private float checkPointsAndAnswers(Set<Question> questions) {
    float sumOfPoints = 0;
    for (Question question : questions) {
        int correctAnswers = 0;
        for (Answer answer : question.getAnswers()) {
            if (answer.isCorrect()) correctAnswers++;
            sumOfPoints += answer.getPoints();
        }
        if (correctAnswers > 1) {
            throw new QuizException("U pitanju: \"" + question.getQuestionText() + "\" postoji previše " +
                    "točnih odgovora, svako pitanje može imati samo jedan točan odgovor");
        }
        if (correctAnswers < 1) {
            throw new QuizException("Za pitanje: \"" + question.getQuestionText() + "\" ne postoji točan odgovor");
        }
    }
    return sumOfPoints;
}
```

Sl. 3.11. *Metoda „checkPointsAndAnswers()“ unutar „QuizServiceImpl“ klase*

„*QuizSolutionServiceImpl*“ je servis unutar kojega se provjeravaju, boduju te spremaju rješenja učenika. Za izvedbu navedenih operacija odgovorne su dvije metode unutar servisa:

1. „*submitQuiz()*“ – metoda koja kao parametre prima identifikacijski broj (ID) kviza, vrijeme rješavanja kviza i listu objekata klase „*QuizSolution*“ od kojih svaki sadrži pitanje iz kviza te odgovor koji je korisnik ponudio. Unutar ove metode provjerava se postojanje kviza na temelju predanog ID-a te se računa ostvareni broj bodova na temelju sljedeće formule:

$$ukupni\_bodovi = \sum_{1}^{broj\_pitanja}(bodovi\_za\_odgovor) \cdot (10 \cdot broj\_pitanja \cdot težina\_kviza - vrijeme\_rješavanja)$$

2. „*calculatePoints()*“ – metoda koja za svaki korisnički odgovor provjerava je li točan, brine se o ostvarenom broju bodova te u „*QuizSolution*“ objekt dodaje točan odgovor kako bi korisnik znao gdje je pogriješio. Također provjerava postojanje predanih pitanja unutar kviza te postojanje predanih odgovora unutar tih pitanja.

Nakon predaje kviza i izvršavanja gore navedenih metoda, servis vraća objekt klase „*QuizReport*“ koji se sastoji od liste objekata klase „*QuizSolutions*“ (s predanim odgovorima, točnim odgovorima i ostvarenim brojem bodova u svakom pitanju), ukupnog broja ostvarenih bodova te vremena rješavanja kviza. Rad metoda prikazan je na slici 3.12.

```java
@Override
public QuizReport submitQuiz(Long quizId, List<QuizSolution> quizSolutions, long timeToFinish) {
    Quiz quiz = this.quizRepository.findById(quizId)
            .orElseThrow(() -> new ResourceNotFoundException("Kviz za koji ste predali odgovore ne postoji"));

    String username = SecurityContextHolder.getContext().getAuthentication().getName();
    User user = this.userRepository.findByUsername(username)
            .orElseThrow(() -> new ResourceNotFoundException("Korisnički račun ne postoji"));

    float pointsSum = calculatePoints(quizSolutions, quiz) *
            (quiz.getQuestions().size() * 10 * quiz.getDifficulty() - timeToFinish);
    QuizParticipant quizResults = new QuizParticipant(pointsSum, (int) timeToFinish, user, quiz);
    quizSolutions.forEach(qs -> qs.getGivenAnswer().setQuestion(qs.getQuestion()));
    Set<Answer> givenAnswers = quizSolutions.stream().map(QuizSolution::getGivenAnswer).collect(Collectors.toSet());
    quizResults.setParticipantAnswers(givenAnswers);

    quizParticipantRepository.save(quizResults);

    return new QuizReport(quizSolutions, pointsSum, (int) timeToFinish);
}

private float calculatePoints(List<QuizSolution> quizSolutions, Quiz quiz) {
    float points = 0;
    for (QuizSolution quizSolution : quizSolutions) {
        Question question = quiz.getQuestions().stream()
                .filter(q -> q.getQuestionId().equals(quizSolution.getQuestion().getQuestionId())).findFirst()
                .orElseThrow(() -> new ResourceNotFoundException(
                        String.format("Pitanje \"%s\" nije pronađeno", quizSolution.getQuestion().getQuestionText())
                ));
        Answer correct = question.getAnswers().stream().filter(Answer::isCorrect).findFirst()
                .orElseThrow(() -> new ResourceNotFoundException(
                        String.format("Ne može se pronaći točan odgovor na pitanje \"%s\"", question.getQuestionText())
                ));
        quizSolution.getGivenAnswer().setQuestion(question);
        quizSolution.checkAnswer(correct);
        points += quizSolution.getPoints();
    }
    return points;
}
```

Sl. 3.12. *Metode za predaju rješenja kviza*

3.1.5. Kontroleri

Kontroleri su Java klase koje obrađuju HTTP zahtjeve te na temelju njih izvršavaju zatraženu radnju (metodu). *Spring Framework* omogućuje korištenje dvije vrste kontrolera:

1. Osnovni kontroleri – rade s pogledima (eng. *views*) tj. *.jsp* ili *.html* datotekama. Na temelju HTTP zahtjeva kontroler odabire metodu koja će se izvršiti te će rezultate izvođenja vratiti u obliku pogleda unutar kojega će ti podaci biti prikazani. Kako bi *Spring Framework* otkrio da je klasa kontroler potrebno je dodati anotaciju „*@Controller*“.
2. REST kontroleri – rade s podacima predstavljenim u JSON ili XML formatu. Ovi kontroleri se koriste za stvaranje REST servisa koji za prikaz različitih objekata koriste neki od prijenosnih formata za razliku od običnih kontrolera koji za prikaz informacija

koriste poglede. Kako bi klasa bila definirana kao REST kontroler potrebno je dodati anotaciju „*@RestController*“. Prilikom svakog zahtjeva svi podaci se iz JSON ili XML formata mapiraju u odgovarajuće objekte, a prilikom vraćanja odgovora se objekt koji je predan kao odgovor pretvara nazad u JSON ili XML.

Za svaku metodu unutar kontrolera potrebno je definirati tip HTTP zahtjeva (*PUT, POST, GET, DELETE, PATCH*...) te krajnju točku za pristup metodi. Sve ovo se definira pomoću sljedećih anotacija:

- „*@RequestMapping(value = "/putanja/do/metode", method = {HTTP zahtjev})* – ovom anotacijom se može definirati bilo koji tip HTTP zahtjeva za navedenu krajnju točku. Dakle ako parametar „*method*“ ima vrijednost „*RequestMethod.GET*“ slanjem isključivo *GET* zahtjeva na gore navedenu krajnju točku će kontroler biti u mogućnosti izvršiti metodu kojoj ta anotacija pripada.
- „*@GetMapping("putanja/do/metode")*“ – anotacija koja je skraćena od „*@RequestMapping*“ anotacije – nije potrebno definirati tip zahtjeva jer je on definiran u samom nazivu anotacije (*GET*). Ovi zahtjevi se najčešće koriste u svrhu dobivanja podataka na temelju predanih parametara.
- „*@PostMapping("putanja/do/metode")*“ – koristi se za *POST* zahtjeve, odnosno najčešće se ovakvi zahtjevi koriste u svrhu dodavanja podataka u bazu.
- „*@PutMapping("putanja/do/metode")*“ – koristi se za *PUT* zahtjeve, ovaj tip se poziva u svrhu ažuriranja podataka unutar baze.
- „*@DeleteMapping("putanja/do/metode")*“ – DELETE zahtjevi u svrhu brisanja podataka iz baze.

Svaka metoda unutar kontrolera može primati određene podatke i to u nekoliko oblika:

1. „*@PathVariable*“ – varijabla putanje – ukoliko se unutar krajnje točke definirane unutar anotacije, postavi tekst oblika „dio-putanje/{varijabla_putanje}“ tada je varijabla putanje jednaka vrijednosti postavljenoj na njezinom mjestu, npr. ako se šalje zahtjev na putanju „dio-putanje/24“, tada će varijabla putanje biti jednaka 24. Varijabla putanje se najčešće koristi za definiranje ID-a nekog objekta („quizzes/15“ – označava kviz koji ima ID = 15)
2. „*@RequestParam*“ – parametar zahtjeva je vrijednost predana u putanji odvojena upitnikom (?). Ukoliko se nakon određene putanje postavi tekst oblika „?ime=Ivan&prezime=Ivić“ tada se iz te putanje mogu dobiti vrijednosti parametara zahtjeva „ime“ i „prezime“ i koristiti unutar pozvane metode.

3. „*@RequestBody*“ – predstavlja tijelo poruke tj. zahtjeva, definiranjem ove anotacije predani objekt u JSON ili XML formatu, unutar tijela zahtjeva, se automatski mapira u objekt definirane klase.

```
@RestController
@RequestMapping("api/quiz-results")
public class QuizResultController {

    @Autowired
    private QuizSolutionService quizSolutionService;

    @PostMapping("{quizId}/submit")
    public ResponseEntity<?> submitQuizResult(
            @PathVariable("quizId") Long quizId,
            @RequestBody List<QuizSolution> quizSolutions,
            @RequestParam(value = "timeToFinish") long timeToFinish
    ){
        QuizReport quizReport = this.quizSolutionService.submitQuiz(quizId, quizSolutions, timeToFinish);
        return ResponseEntity.ok(quizReport);
    }
}
```

Sl.3.13. *„QuizResultController“ klasa*

Na slici 3.13. prikazan je izgled REST kontrolera za predavanje rješenja kviza. Iznad same klase je uz anotaciju „*@RestController*“ dodana i „*@RequestMapping*“ anotacija koja označava korijensku putanju za sve metode definirane unutar ovog kontrolera, dakle u ovom slučaju, slanjem *POST* zahtjeva na putanju „.../api/quiz-results/5/submit?timeToFinish=60“ zajedno s listom „*QuizSolution*“ objekata poziva se metoda „*submitQuizResult*“ unutar koje će varijabla „*quizId*“ imati vrijednost 5, a „*timeToFinish*“ vrijednost 60.

```java
@PostMapping
public ResponseEntity<?> insertQuiz(
        @RequestBody @Valid Quiz quiz
) {
    return ResponseEntity.ok(this.quizService.save(quiz));
}

@GetMapping("{id}")
public ResponseEntity<?> quizById(
        @PathVariable("id") Long id
) {
    return ResponseEntity.ok(this.quizRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Ne postoji kviz koji ima id: " + id)));
}

@PutMapping("{id}")
public ResponseEntity<?> updateQuiz(
        @PathVariable("id") Long id,
        @RequestBody @Valid Quiz quiz
) {
    return ResponseEntity.ok(this.quizService.update(quiz, id));
}

@DeleteMapping("{id}")
public ResponseEntity deleteQuiz(
        @PathVariable("id") Long quizId
) {
    this.quizRepository.findById(quizId)
            .orElseThrow(() -> new ResourceNotFoundException("Ne postoji traženi kviz"));
    this.quizRepository.deleteById(quizId);
    return new ResponseEntity(HttpStatus.NO_CONTENT);
}
```

Sl 3.14. *Metode za POST, GET, PUT i DELETE zahtjeve unutar „QuizController“ klase*

3.2. Android aplikacija

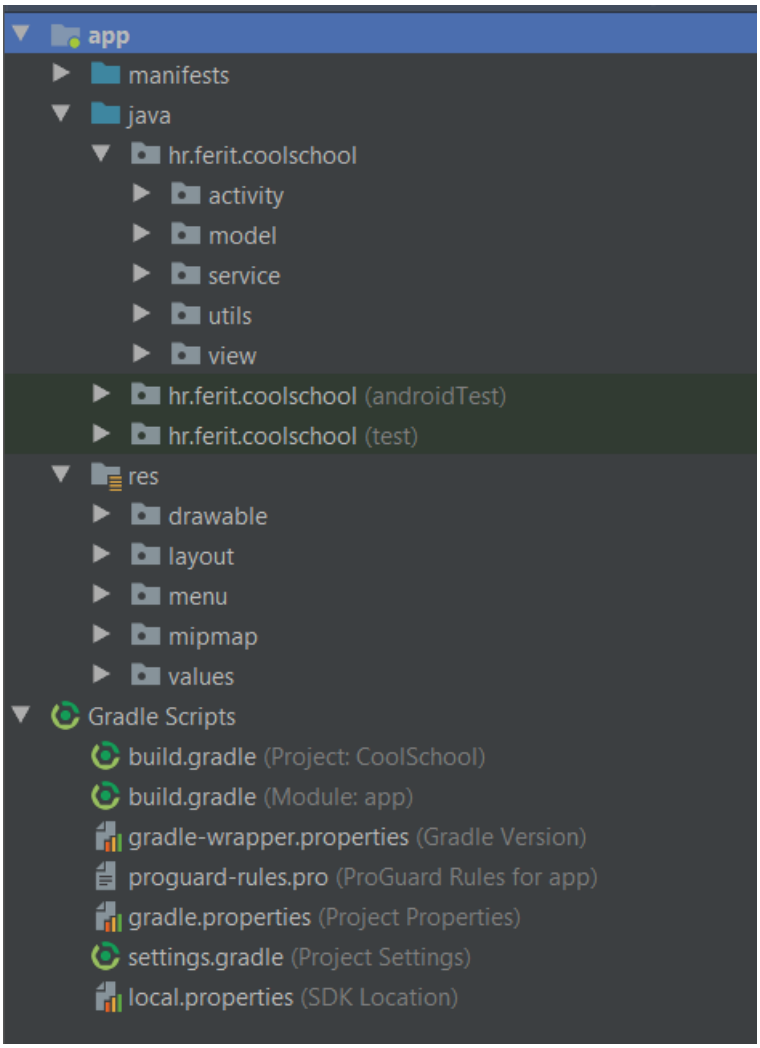
Klijentska strana sustava za zadavanje zadataka je Android aplikacija koja je napravljena kao razumljivo (eng. *user-friendly*) i jednostavno okruženje za manipulaciju podacima s poslužitelja. Pokretanjem aplikacije pokreće se aktivnost „*SplashScreenActivity*“ koja konfigurira animaciju slike na zaslonu te provjerava postoji li prijavljeni korisnik na uređaju. Ukoliko postoji pokreće se glavna kartična aktivnost (eng. *tabbed activity*) „*DashboardActivity*“ koja se sastoji od tri kartice s fragmentima:

1. **„***RankingFragment*“ – fragment koji s poslužitelja preuzima rang listu korisnika na temelju odabranih filtera te je prikazuje na zaslonu
2. „*QuizFragment*“ – fragment koji s poslužitelja dohvaća listu postojećih kvizova s opcijom filtriranja te učenicima omogućava rješavanje kvizova, a učiteljima dodavanje novih kvizova.

3. „*SettingsFragment*“ – fragment za mijenjanje korisničkih podataka poput imena, prezimena, korisničkog imena i lozinke te podataka o školama.

Ako ne postoji prijavljeni korisnik na uređaju pokreće se aktivnost „*LoginActivity*“ koja šalje podatke za autentifikaciju korisnika na poslužitelj te prikazuje odgovor poslužitelja korisniku.

Aplikacija se sastoji od nekoliko modula (paketa): postoje modeli („*models*“) koji odgovaraju modelima s poslužiteljske strane, aktivnosti („*activity*“) i fragmenti („*fragment*“), servisi („*service*“) koji predstavljaju sučelja za slanje zahtjeva na poslužitelj, pomoćne klase („*utils*“) te pogledi („*views*“) odnosno različiti adapteri za „*RecyclerView*“ komponentu.



Sl. 3.15. *Osnovna struktura Android projekta*

3.2.1. *Retrofit*

Retrofit je biblioteka koja predstavlja REST klijent za Android, Javu i Kotlin. Biblioteka pruža snažan programski okvir (eng. *framework*) za autentifikaciju i interakciju s REST servisima te slanje zahtjeva koristeći „*OkHttp*“ klijent. Olakšava preuzimanje i slanje podataka u JSON i/ili XML formatima na način da se preuzeti podaci mapiraju u tzv. *POJO* (eng. *Plain Old Java Object*) klase koje moraju biti definirane za svaki resurs u odgovoru na zahtjev. [6] Za korištenje retrofita potrebno je dodati ovisnosti u „*build.gradle*“ datoteku na način prikazan na slici 3.16.

```
dependencies {
    implementation 'com.github.ViksaaSkool:AwesomeSplash:v1.0.0'
    implementation 'com.squareup.retrofit2:retrofit:2.4.0'
    implementation 'com.squareup.retrofit2:converter-jackson:2.3.0'
}
```

Sl. 3.16. *Ovisnosti za Retrofit biblioteku i mapiranje objekata*

Nakon uspješne sinkronizacije projekta moguće je koristiti sve mogućnosti *Retrofita*. Potrebno je kreirati klasu koja će predstavljati jedinstvenu instancu (eng. *singleton*) *Retrofit* objekta, u ovom projektu ta klasa se naziva „*RetrofitImpl*“. Unutar te klase definira se IP adresa za pristup poslužitelju, instance servisa koji će se koristiti za komunikaciju s poslužiteljem te statička metoda za kreiranje *Retrofit* instance zajedno s definiranjem HTTP klijenta („*OkHttp*“) te klase za pretvaranje *POJO* objekata u JSON/XML format i obrnuto.

3.2.2. „*SplashScreenActivity*“

Početna aktivnost koja se izvodi prilikom pokretanja aplikacije i služi za konfiguraciju animacije na zaslonu i odabira sljedeće aktivnosti, na temelju toga postoji li prijavljeni korisnik na uređaju ili ne. Za konfiguraciju animacije potrebno je nadjačati metodu „*initSplash()*“ klase „*AwesomeSplash*“, unutar te metode moguće je postaviti resurs (sliku) koja će se prikazivati na zaslonu, tip animacije i trajanje animacije. Za nastavak izvođenja aplikacije potrebno je u nadjačanoj metodi „*animationsFinished()*“ provjeriti postoji li spremljeni tj. prijavljeni korisnik na uređaju. Podaci o svakom korisniku koji se uspješno prijavi spremaju se u zajedničke postavke uređaja koristeći „*SharedPreferences*“ klasu pa se tako u metodi

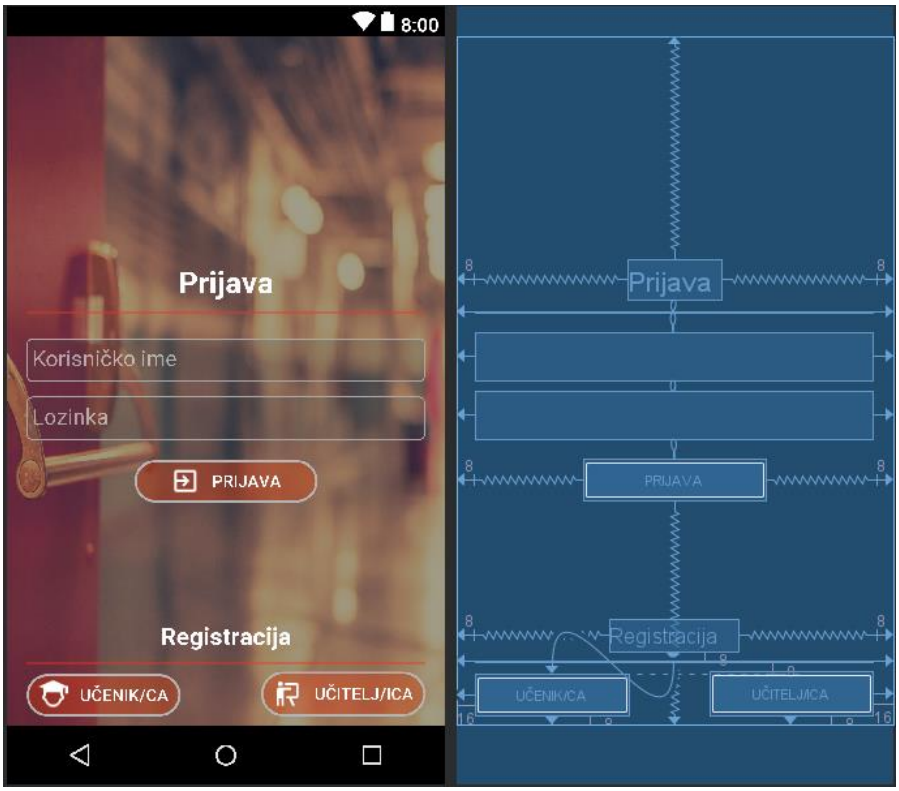
„*animationsFinished()*“ provjerava postoji li spremljeni korisnik u zajedničkim postavkama. Ukoliko postoji ponovnom prijavom u sustav, koristeći *Retrofit* poziv za prijavu korisnika, osvježava se sesija na poslužitelju tj. poslužitelj u zaglavlju odgovora šalje kolačić (eng. *cookie*) koji predstavlja ID sesije, a on je potreban za autorizaciju gotovo svih zahtjeva (ovisi o konfiguraciji na poslužitelju). Kolačić se sprema u zajedničke postavke uređaja kako bi mu se moglo pristupiti bilo kada i iz bilo kojeg dijela aplikacije. Nakon toga se kreira *Intent* za pokretanje aktivnosti „*DashboardActivity*“. Ako ne postoji spremljeni korisnik u dijeljenim postavkama kreira se *Intent* za pokretanje aktivnosti „*LoginActivity*“.

```
public class SplashScreenActivity extends AwesomeSplash {

    private SharedPrefsHelper mSharedPrefs;

    @Override
    public void initSplash(ConfigSplash configSplash) {

        configSplash.setBackgroundColor(R.color.white);
        configSplash.setAnimCircularRevealDuration(1000);
        configSplash.setRevealFlagX(Flags.REVEAL_RIGHT);
        configSplash.setRevealFlagY(Flags.REVEAL_BOTTOM);

        configSplash.setLogoSplash(R.drawable.logo);
        configSplash.setAnimLogoSplashDuration(1000);
        configSplash.setAnimLogoSplashTechnique(Techniques.Tada);

        configSplash.setOriginalHeight(400);
        configSplash.setOriginalWidth(400);
        configSplash.setAnimPathStrokeDrawingDuration(5);
        configSplash.setPathSplashStrokeSize(1);
        configSplash.setPathSplashStrokeColor(R.color.colorPrimary);
        configSplash.setAnimPathFillingDuration(1000);
        configSplash.setPathSplashFillColor(R.color.colorPrimary);
    }

    @Override
    public void animationsFinished() {
        mSharedPrefs = new SharedPrefsHelper( mContext: this);
        if (mSharedPrefs.getAuthenticatedUserInfo() == null) {
            startLoginActivity();
        } else {
            obtainCookieByLogin(mSharedPrefs.getAuthenticatedUserInfo());
        }
    }
```

Sl. 3.17. *„SplashScreenActivity“ klasa*

3.2.3. *„LoginActivity“*

Ova aktivnost služi za prijavu u sustav, tj. autentifikaciju korisnika. Na zaslonu su vidljiva dva polja za unos (polje za unos korisničkog imena i polje za unos zaporke) te dugme za prijavu, dugme za registraciju učenika/ce i dugme za registraciju učitelja/ice. Korisničko sučelje je kreirano pomoću *ConstraintLayout*a koji omogućuje povezivanje kontrola korisničkog sučelja pomoću različitih ograničenja (eng. *constraints*).



Sl. 3.18. *Korisničko sučelje aktivnosti „LoginActivity“*

Pritiskom na dugme „Prijava“ pokreće se metoda „*login()*“ koja stvara zahtjev za poslužitelja pomoću *Retrofita*. Svaki tip zahtjeva mora biti definiran u servisima čije se instance nalaze u „*RetrofitImpl*“ klasi i oni moraju odgovarati postavljenim krajnjim točkama na poslužitelju. Za prijavu korisnika u sustav na poslužitelju je, u *Spring Security* modulu, ukomponirana krajnja točka „*/login*“ koja prima *POST* zahtjev s parametrima „*username*“ i „*password*“ (korisničko ime i lozinka). Kako bi se taj zahtjev mogao kreirati unutar „*UserService*“ sučelja definiran je upravo odgovarajući zahtjev prikazan na slici 3.19.

```
public interface UserService {

    @FormUrlEncoded
    @POST("/login")
    Call<ResponseBody> login(@Field("username") String username, @Field("password") String password);
```

Sl. 3.19. *Metoda koja predstavlja zahtjev za prijavu korisnika*

Metoda „*login()*“ unutar „*UserService*“ sučelja označava pripremljeni zahtjev za poslužitelja. Anotacija „*@POST*“ označava da se radi o *POST* zahtjevu na krajnju točku predanu kao parametar, a anotacija „*@FormUrlEncoded*“ označava da će unutar zahtjeva postojati parovi ključ-vrijednost (eng. *key-value pairs*) – u ovom slučaju će to biti korisničko ime i lozinka. Svaka metoda definirana u ovakvom sučelju mora imati povrati tip „*Call<T>*“ gdje T označava klasu čiji će objekt biti vraćen kao odgovor s poslužitelja.

Pozivanjem metode „*login()*“ šalje se zahtjev na poslužitelj, a za rukovanje odgovorom potrebno je implementirati metode „*onResponse()*“ i „*onFailure()*“. Ukoliko postoji odgovor s poslužitelja automatski se pokreće „*onResponse()*“ metoda, a ako se ne može uspostaviti veza s poslužiteljem pokreće se „*onFailure()*“ metoda. Za prijavu korisnika tijelo odgovora (eng. *response body*) je prazno, ali je šifra odgovora (eng. *response code*) jednaka „200“ što označava da je zahtjev uspješno izvršen te je iz zaglavlja odgovora potrebno dohvatiti vrijednost kolačića „*Set-cookie*“ koji se postavlja u zaglavlje svih daljnjih zahtjeva na poslužitelj, a ukoliko je šifra odgovora jednaka „401“ tada zahtjev nije prošao i korisnik nije uspješno autentificiran te ga se o tome obavještava. Ako je prijava bila uspješna kreira se novi zahtjev za dohvaćanje korisničkih podataka s poslužitelja.

```
private void login(String username, String password) {
    Call<ResponseBody> call = RetrofitImpl.getUserService().login(username, password);

    call.enqueue(new Callback<ResponseBody>() {
        @Override
        public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {
            Log.d( tag: "LOGIN RESPONSE", String.valueOf(response.code()));
            if (response.code() == 200) {
                cookie = response.headers().get("Set-Cookie");
                sharedPrefsHelper.setCookie(cookie);
                fetchUserInfo(cookie);
            } else if (response.code() == 401) {
                Log.e( tag: "LOGIN ERROR",  msg: "Unauthorized");
                Toast.makeText(getApplicationContext(), LOGIN_FAIL, Toast.LENGTH_SHORT).show();
            }
        }

        @Override
        public void onFailure(Call<ResponseBody> call, Throwable t) {
            Log.e( tag: "ERROR", t.toString());
            showDefaultError();
        }
    });
}
```

Sl. 3.20. *„login()“ metoda unutar aktivnosti „LoginActivity“*

3.2.4. *„RegistrationActivity“*

Na slici 3.18. se na zaslonu mogu vidjeti dva dodatna dugmeta ispod naslova „Registracija“ – dugme „UČENIK/CA“ i dugme „UČITELJ/ICA“. Pritiskom na bilo koji od njih pokreće se „*RegistrationActivity*“, ali ovisno o odabranom tipu korisnika (učenik/ica ili učitelj/ica) stvara se korisničko sučelje za registraciju korisnika. Zajednički dio korisničkog sučelja su *TextInputLayout* kontrole za unos imena, prezimena, korisničkog imena, lozinke i elektroničke pošte te kontrola za pretraživanje postojećih škola. Prilikom pokretanja aktivnosti se na temelju tipa korisnika određuje slika pozadine zaslona te ako se radi o učenicima uz kontrolu za pretraživanje škola se dodaje kontrola za upis razreda kojeg učenik pohađa, a ako se radi o učiteljima uz kontrolu za pretraživanje škola se postavlja *Spinner* kontrola za odabir predmeta kojeg učitelj predaje u odabranoj školi.

Sl. 3.21.*Korisničko sučelje aktivnosti „RegistrationActivity“*

Pritiskom na dugme „Registracija“ izvršava se metoda „*registerUser(User user)*“ unutar koje se šalje zahtjev za registraciju korisnika na poslužitelj. Parametar „*user*“ je objekt izveden iz vrijednosti unesenih u kontrole na korisničkom sučelju. Za validaciju unesenih podataka brinu se dvije metode koje se izvršavaju prije slanja zahtjeva za registraciju:

1. „*getUserFromFields()*“ – metoda koja dohvaća sve unesene vrijednosti iz kontrola s korisničkog sučelja i sprema ih u varijable te provjerava je li korisnik dodao barem jednu školu koju pohađa ili u kojoj predaje. Ukoliko je lista škola prazna, obavještava korisnika da je obavezno unijeti barem jednu školu. Ako se uspostavi da su uneseni podaci prošli validaciju, kreira se i vraća novi objekt klase „*User*“, a ako podaci ne prođu validaciju metoda vraća *null* vrijednost.
2. „*checkData()*“ - metoda koja obavlja validaciju unesenih vrijednosti koje je primila kao parametre. Za svaku varijablu provjerava ima li vrijednost *null*, ako ima postavlja grešku

s porukom na odgovarajuću *TextInputLayout* kontrolu, a ukoliko nema provjerava se format unesene vrijednosti:

a. Ime i prezime - može se sastojati samo od velikih i malih slova abecede
b. Korisničko ime – može se sastojati od malih i velikih slova abecede, brojeva i znakova minus (-), povlaka (_) i točka (.) s time da ta tri znaka ne smiju biti na kraju ili početku korisničkog imena
c. Adresa elektroničke pošte – za dio elektroničke pošte prije znaka @ vrijedi isto pravilo kao i za korisničko ime, kao i za dio elektroničke pošte između znaka @ i domene koja počinje s točkom (.), a može sadržavati dva, tri ili četiri slova ili broja

Ukoliko se dogodi da podaci nisu dobrog formata, odgovarajuća *TextInputLayout* kontrola ispisuje grešku. Naposljetku se provjerava jesu li unesene lozinke jednake te imaju li više od pet znakova, ako je to slučaj te su sve druge vrijednosti prošle validaciju, metoda „*checkData()*“ će vratiti vrijednost „*true*“, u suprotnom vraća vrijednost „*false*“

U metodi „*registerUser()*“ provjerava se jesu li uneseni podaci prošli validaciju, tj. provjerava se je li iz unesenih vrijednosti stvoren objekt klase „*User*“. Ako je objekt stvoren poziva se metoda *„registration()*“ iz „*UserService*“ sučelja za stvaranje zahtjeva. Ako je šifra odgovora „200“ registracija je uspješno izvršena, o tome se obavještava korisnik te se ponovno pokreće aktivnost „*LoginActivity*“. Ukoliko šifra odgovora nije „200“, registracija korisnika nije uspjela zbog iznimke na poslužitelju te se o tome obavještava korisnik.

```
private void registerUser(User userFromFields) {
    if (userFromFields != null) {
        Call<User> call = RetrofitImpl.getUserService().registration(userFromFields);

        call.enqueue(new Callback<User>() {
            @Override
            public void onResponse(Call<User> call, Response<User> response) {
                if (response.isSuccessful()) {
                    Toast.makeText(getApplicationContext(), text: "Uspješna registracija",
                            Toast.LENGTH_SHORT).show();
                    Intent intent = new Intent(getApplicationContext(), LoginActivity.class);
                    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK)
                    startActivity(intent);
                } else {
                    Log.e( tag: "ERROR", response.toString());
                    Toast.makeText(getApplicationContext(), text: "Registracija nije uspjela",
                            Toast.LENGTH_SHORT). show();
                }
            }

            @Override
            public void onFailure(Call<User> call, Throwable t) {
                Log.e( tag: "ERROR", t.toString());
                showDefaultError();
            }
        });
    }
}
```

Sl. 3.22. *Metoda „registerUser()“ u aktivnosti „RegistrationActivity“*

3.2.5. „DashboardActivity“

Glavna aktivnost ove aplikacije je kartična aktivnost „*DashboardActivity*“ koja pomoću tri fragmenta manipulira podacima dobivenim s poslužitelja. Kartična aktivnost je aktivnost koja omogućuje navigaciju između kartica gestama prstiju (povlačenjem prsta lijevo ili desno). Navigaciju između kartica omogućuje *ViewPager* kontrola koja svaki pogled promatra kao posebnu stranicu odnosno karticu. Za umetanje pogleda koji će predstavljati pojedinu stranicu potrebno je *ViewPager* spojiti s instancom *FragmentPagerAdapter* klase unutar koje se stvara lista fragmenata čiji će pogledi predstavljati stranice. [7]

```
public class SectionsPagerAdapter extends FragmentPagerAdapter {
    private final List<Fragment> fragments = new ArrayList<>();

    SectionsPagerAdapter(FragmentManager fm) { super(fm); }

    @Override
    public Fragment getItem(int position) { return this.fragments.get(position); }

    @Override
    public int getCount() { return this.fragments.size(); }

    void addFragment(Fragment fragment) { this.fragments.add(fragment); }

    @Override
    public CharSequence getPageTitle(int position) { return null; }
}
```

Sl. 3.23. *„SectionPageAdapter“ klasa za postavljanje ViewPager kontrole*

Prilikom pokretanja aktivnosti stvara se instanca „*SectionPageAdapter*“ te se stvaraju novi fragmenti koji će predstavljati kartice unutar *ViewPager*-a i dodaju se u listu fragmenata unutar „*SectionPageAdapter*“ instance. Nakon toga se metodom „*setAdapter()*“ klase *ViewPager*, instanca „*SectionPageAdapter*“ klase predaje u instancu *ViewPager*-a koji se koristi u ovoj aktivnosti.

```
private void setUpViewPager() {
    Fragment rankingFragment = RankingFragment.newInstance(mAuthdUser, mCookie);
    Fragment quizFragment = QuizFragment.newInstance(mAuthdUser, mCookie);
    Fragment settingsFragment = SettingsFragment.newInstance(mAuthdUser, mCookie);
    mSectionsPagerAdapter.addFragment(rankingFragment);
    mSectionsPagerAdapter.addFragment(quizFragment);
    mSectionsPagerAdapter.addFragment(settingsFragment);
    mViewPager.setAdapter(mSectionsPagerAdapter);
}
```

Sl. 3.24. *Stvaranje fragmenata, dodavanje u instancu „SectionPageAdapter“ klase i postavljanje adaptera u ViewPager*

3.2.5.1. „*RankingFragment*“

Prvu karticu u aktivnosti „*DashboardActivity*“ predstavlja fragment „*RankingFragment*“. Ovaj fragment (kao i svi ostali) prilikom kreiranja nove instance prima podatke o prijavljenom korisniku te kolačić sesije. Kreira se korisničko sučelje koje se sastoji od *TextView*-a za prikaz trenutne pozicije prijavljenog učenika, skupa *Spinner* kontrola za filtriranje rang liste i *RecyclerView* kontrole za prikaz dohvaćene liste s poslužitelja. Rang listu je moguće filtrirati

prema školi, predmetu, gradu i županiji. Pritiskom na dugme za pretraživanje pokreće se metoda „*fetchRankingList()*“ koja kreira zahtjev za dohvaćanje rang liste na temelju odabranih filtera. Ukoliko je zahtjev uspješan, poslužitelj će vratiti listu „*Rank*“ objekata koji sadrže podatke o korisničkom imenu korisnika, njegovom ukupnom broju skupljenih bodova te o školi i razredu koji pohađa. Nakon spremanja dobivenog odgovora potrebno je obavijestiti „*RankAdapter*“ o promjeni podataka kako bi se *RecyclerView* osvježio.

```
private void fetchRankingList(Integer school, String city, String state, Subject subject) {
    Call<List<Rank>> call = RetrofitImpl.getRankService().listRanks(mCookie, school, city, state, subject);

    call.enqueue(new Callback<List<Rank>>() {
        @Override
        public void onResponse(Call<List<Rank>> call, Response<List<Rank>> response) {
            if (response.isSuccessful()) {
                mRanks.clear();
                mRanks.addAll(response.body());
                mRankAdapter.notifyDataSetChanged();
            } else {
                Log.e( tag: "ERROR", response.toString());
            }
        }

        @Override
        public void onFailure(Call<List<Rank>> call, Throwable t) {
            Log.e( tag: "ERROR", t.toString());
            Toast.makeText(getContext(), DEFAULT_ERROR, Toast.LENGTH_SHORT).show();
        }
    });
}
```

Sl. 3.25. *Metode „fetchRankingList()“ i „updateCurrentPositionText()“*

3.2.5.2. „*QuizFragment*“

Drugu karticu u aktivnosti „*DashboardActivity*“ predstavlja fragment „*QuizFragment*“ koji na temelju filtera prikazuje listu kvizova. Ukoliko je prijavljeni korisnik učenik, za dohvaćanje liste kvizova će se automatski dodati razred koji učenik pohađa i tako će biti prikazani svi omogućeni (eng. *enabled*) kvizovi koji su napravljeni za isti razred koji učenik pohađa. Dakle ako je korisnik učenik 3. razreda osnovne škole biti će prikazani svi omogućeni kvizovi za učenike 3. razreda osnovne škole. Ako je prijavljeni korisnik učitelj prikazat će se svi omogućeni i onemogućeni kvizovi te će se primijeniti filter za predmet koji korisnik predaje, odnosno učitelji mogu dohvatiti kvizove samo za predmete koji oni predaju. Zahtjev za dohvaćanje filtrirane liste kvizova stvara se unutar metode „*fetchQuizList()*“. Za uspješno dohvaćanje liste kvizova obavještava se adapter „*QuizAdapter*“ koji osvježava podatke u *RecyclerView* kontroli. Učiteljima je na korisničkom sučelju prikazana *FloatingActionButton*

kontrola čijim se odabirom pokreće nova aktivnost „*QuizInsertActivity*“ unutar koje se stvara novi kviz. Učenicima je omogućen odabir kviza s popisa čime se pokreće aktivnost „*QuizSubmitActivity*“ koja omogućuje rješavanje odabranog kviza.

```
private void fetchQuizList(Integer difficulty, Integer classNum, Subject subject) {
    boolean enabled = mAuthUser.getRole().equals(Role.ROLE_STUDENT);
    Call<List<Quiz>> call = RetrofitImpl.getQuizService().listQuizzes(
            mCookie, difficulty, enabled, classNum, SchoolType.ELEMENTARY_SCHOOL, subject);

    call.enqueue(new Callback<List<Quiz>>() {
        @Override
        public void onResponse(Call<List<Quiz>> call, Response<List<Quiz>> response) {
            if (response.isSuccessful()) {
                mQuizzes.clear();
                mQuizzes.addAll(response.body());
                mQuizAdapter.notifyDataSetChanged();
            }
        }

        @Override
        public void onFailure(Call<List<Quiz>> call, Throwable t) {
            Log.e( tag: "ERROR", t.toString());
            Toast.makeText(getContext(), DEFAULT_ERROR, Toast.LENGTH_SHORT).show();
        }
    });
}
```

Sl. 3.26. *Metoda „fetchQuizList()“ za dohvaćanje liste kvizova u fragmentu „QuizFragment“*

3.2.5.3. „*SettingsFragment*“

Treću karticu u aktivnosti „*DashboardActivity*“ predstavlja fragment „*SettingsFragment*“ koji služi za prikazivanje te ažuriranje profila prijavljenog korisnika. Prilikom stvaranja fragmenta na korisničkom sučelju su prikazana dva dugmeta: „Odjava“ i „Uredi“. Pritiskom na dugme „Odjava“ brišu se podaci o korisniku iz zajedničkih postavki uređaja, šalje se zahtjev za odjavu na poslužitelj kako bi se uklonila korisnikova sesija te se pokreće aktivnost „*LoginActivity*“. Prvobitno su na korisničkom sučelju prikazane onemogućene *TextInputLayout* kontrole koje prikazuju korisničke podatke (ime, prezime, korisničko ime i adresu elektroničke pošte) i u *RecyclerView* kontroli se nalazi lista škola koje je korisnik dodao pri registraciji. Pritiskom na dugme „Uredi“ pojavljuju se nove *TextInputLayout* kontrole koje zajedno s već postojećima omogućuju korisniku unos podataka. Nove kontrole se odnose na unos stare i nove lozinke te se pojavljuje i kontrola za pretraživanje škola, a ovisno o tipu korisnika pojavljuje se kontrola za upis razreda ili *Spinner* kontrola za odabir predmeta. Pojavljuje se novo dugme „Uredi“ i dugme „Odustani“, pritiskom na „Odustani“ korisničko sučelje se vraća u prvobitno

stanje i prikazuje podatke koji su bili prikazani prije uređivanja, a pritiskom na dugme „Spremi“ pokreće se metoda „*updateUser()*“. Ova metoda radi na istom principu kao i metoda za registraciju korisnika u „*RegistrationActivity*“, uzima vrijednosti iz polja i sprema ih u varijable te vrši validaciju nad upisanim podacima. Ukoliko polja uspješno prođu validaciju poziva se metoda „*updateUserCall()*“ koja kao parametre prima objekt klase „*User*“ odnosno korisnika s novim podacima i *boolean* vrijednost koja označava da se nakon uspješno obrađenog zahtjeva osvježi korisničko sučelje i postave novi podaci u kontrole ili ne. Hoće li se korisničko sučelje osvježiti ovisi o tome želi li korisnik promijeniti lozinku. Ako korisnik unese staru i novu lozinku stvara se drugi zahtjev za ažuriranje lozinke metodom „*updatePassword()*“. „*updateUserCall()*“ kreira zahtjev za ažuriranje podataka o korisniku (svih podataka osim lozinke) te za uspješnu obradu zahtjeva poslužitelj vraća ažurirani objekt klase „*User*“. Zbog korištenja ovog objekta u svim fragmentima, potrebno je o njegovoj promjeni obavijestiti aktivnost (ukoliko se lozinka ne mijenja). Kako bi fragment proslijedio podatke aktivnosti, unutar fragmenta je potrebno definirati sučelje s metodom koja kao parametar prima željeni podatak. Implementacijom sučelja od strane aktivnosti omogućeno je nadjačavanje tj. implementacija definirane metode. Dovoljno je unutar fragmenta pozvati metodu, ona će se izvršiti u aktivnosti.

```
private void updateUserCall(User userFromFields, boolean resetLayout) {
    Call<User> call = RetrofitImpl.getUserService().updateUser(
            mCookie, mAuthUser.getUserId(), userFromFields);

    call.enqueue(new Callback<User>() {
        @Override
        public void onResponse(Call<User> call, Response<User> response) {
            if (response.isSuccessful()) {
                String password = mAuthUser.getPassword();
                mAuthUser = response.body();
                mAuthUser.setPassword(password);

                mSchoolsAdapter.notifyDataSetChanged();
                if (resetLayout) {
                    resetLayoutValues();
                    mUserUpdateListener.onUserUpdateListener(mAuthUser);
                }
            } else {
                Log.e( tag: "ERROR", response.toString());
            }
        }

        @Override
        public void onFailure(Call<User> call, Throwable t) {
            Log.e( tag: "ERROR", t.toString());
            Toast.makeText(getActivity(), DEFAULT_ERROR, Toast.LENGTH_SHORT).show();
        }
    });
}
```

Sl 3.27. *Metoda „updateUserCall()“*

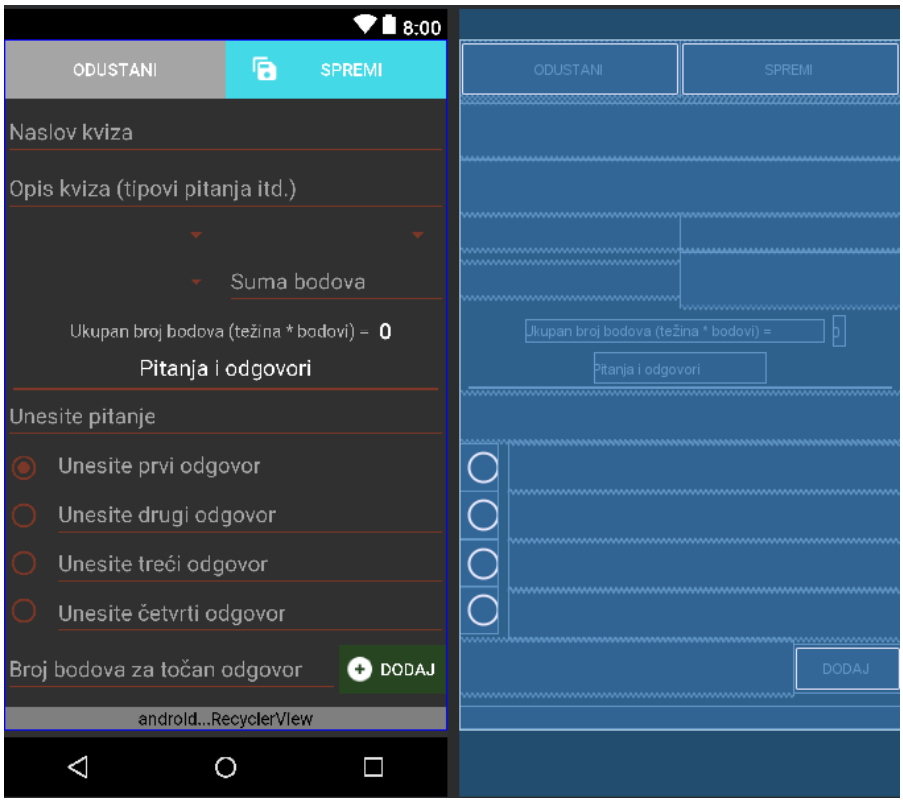
Ako je korisnik zatražio i promjenu lozinke, metodom „*updatePassword()*“ šalje se zahtjev za promjenu lozinke. U slučaju uspješne promjene lozinke poslužitelj vraća *boolean* vrijednost *true*, u suprotnom vraća *false*. Ako je poslužitelj vratio *true* korisnik dobiva poruku o uspješnom ažuriranju profila, osvježava se korisničko sučelje s novim podacima i obavještava se aktivnost o ažuriranju korisnika.

```
private void updatePassword(String password, String newPass) {
    Call<Boolean> call = RetrofitImpl.getUserService().updateUserPassword(
            mCookie, mAuthUser.getUserId(), password, newPass);

    call.enqueue(new Callback<Boolean>() {
        @Override
        public void onResponse(Call<Boolean> call, Response<Boolean> response) {
            if (response.isSuccessful()) {
                Toast.makeText(getActivity(), text: "Profil je uspješno ažuriran", Toast.LENGTH_SHORT).show();
                if (response.body()) {
                    mAuthUser.setPassword(newPass);
                    mSchoolsAdapter.notifyDataSetChanged();
                    resetLayoutValues();
                    mUserUpdateListener.onUserUpdateListener(mAuthUser);
                }
            } else {
                Log.e( tag: "ERROR", response.toString());
            }
        }

        @Override
        public void onFailure(Call<Boolean> call, Throwable t) {
            Toast.makeText(getActivity(), DEFAULT_ERROR, Toast.LENGTH_SHORT).show();
            Log.e( tag: "ERROR", t.toString());
        }
    });
}
```

Sl. 3.28. *Metoda „updatePassword()“*

3.2.6. „*QuizInsertActivity*“

Aktivnost za stvaranje novog kviza naziva se „*QuizInsertActivitiy*“ koju mogu pokrenuti samo učitelji pritiskom na *FloatingActionButton* kontrolu u fragmentu „*QuizFragment*“. Na korisničkom sučelju ove aktivnosti nalaze se *TextInputLayout*, *EditText*, *RadioButton*, *Button* i *Spinner* kontrola. *TextInputLayout* kontrole služe za unos naziva, opisa kviza te sume bodova, *EditText* kontrole se koriste za unos pitanja, odgovora i broja bodova za točan odgovor, *RadioButton* kontrole služe za odabir točnog odgovora između četiri unesena odgovora, a *Spinner* kontrole za odabir predmeta na kojeg se odnose pitanja, težinu kviza te razred za koji je kviz primjeren.



Sl. 3.29. *Korisničko sučelje aktivnosti „QuizInsertActivity“*

Na dnu korisničkog sučelja postoji *RecyclerView* unutar kojeg se prikazuju spremljena pitanja i odgovori. Polja za unos pitanja i odgovora su obavezna, kao i polje za unos bodova za točan odgovor. Dakle svaki kviz se sastoji od pitanja, a svako pitanje mora imati četiri ponuđena odgovora od kojih je samo jedan točan. Unosom pitanja i odgovora te odabirom točnog odgovora pritiskom na odgovarajuću *RadioButton* kontrolu moguće je dodati pitanje s odgovorima u *RecyclerView,* odnosno u listu pitanja koja će se spremiti u kviz, pritiskom na dugme „Dodaj“. Unosom ostalih podataka (naziva kviza, opisa kviza, odabira predmeta, razreda i težine) i pritiskom na dugme „Spremi“ izvršava se metoda „*saveQuizToServer()*“. Ova metoda kreira zahtjev za dodavanje novog kviza u bazu podataka, ako je zahtjev uspješan završava se trenutna aktivnost s rezultatom „OK“ te se time obavještava aktivnost „*DashboardActivitiy*“ da je kviz uspješno dodan.

```
private void saveQuizToServer(Quiz newQuiz) {
    Call<Quiz> call = RetrofitImpl.getQuizService().insertQuiz(mCookie, newQuiz);

    call.enqueue(new Callback<Quiz>() {
        @Override
        public void onResponse(Call<Quiz> call, Response<Quiz> response) {
            if (response.isSuccessful()) {
                Intent intent = new Intent();
                setResult(RESULT_OK, intent);
                finish();
            } else {
                Log.e( tag: "ERROR", response.toString());
            }
        }

        @Override
        public void onFailure(Call<Quiz> call, Throwable t) {
            Toast.makeText(getApplicationContext(), DEFAULT_ERROR, Toast.LENGTH_SHORT).show();
            Log.e( tag: "ERROR", t.toString());
        }
    });
}
```

Sl. 3.30. *Metoda „saveQuizToServer()“*

3.2.7. „*QuizSubmitActivity*“

Učenici pritiskom na jedan od ponuđenih kvizova unutar liste prikazane u fragmentu „*QuizFragment*“ pokreću aktivnost „*QuizSubmitActivity*“ čime korisnik automatski započinje rješavanje kviza. Prilikom pokretanja aktivnosti na korisničkom sučelju se prikažu pojedinosti kviza, pitanja s ponuđenim odgovorima te preostalo vrijeme za rješavanje kviza. Vrijeme za rješavanje kviza se izračunava na temelju sljedeće formule:

$$preostalo\_vrijeme = broj\_pitanja \cdot 10000 \cdot težina\_kviza$$

Pretpostavka je da za težinu kviza „1“ korisnik treba 10 sekundi za odabir točnog odgovora pa se ta vrijednost množi s brojem pitanja čime se dobiva ukupno vrijeme u sekundama. Za odbrojavanje se koristi klasa „*CountDownTimer*“ koja vrijeme promatra u milisekundama i zbog toga se još vrijeme u sekundama mora pomnožiti s 1000 čime se dobiva ukupno vrijeme trajanja kviza u milisekundama. „*CountDownTimer*“ ima dva događaja:

1. „*onTick()*“ – događaj koji se u ovom slučaju javlja nakon svake sekunde kako bi se postavio tekst unutar *TextView* kontrole koja prikazuje preostalo vrijeme. Provjerava se je li vrijednost preostalog vremena veća od 60 sekundi, ako je - preostalo vrijeme će biti prikazano u obliku „mm:ss“ gdje „mm“ označava minute, a „ss“ sekunde. Ako je do kraja kviza ostalo manje od minute vrijeme će biti prikazano u sekundama.

2. „*onFinish()*“ – događaj koji se javlja nakon isteka definiranog vremena.

```
private void setUpTimer() {
    long time = mQuiz.getQuestions().size() * 10 * mQuiz.getDifficulty() * 1000;
    timer = new CountDownTimer(time,  countDownInterval: 1000) {

        @SuppressLint("DefaultLocale")
        public void onTick(long millisUntilFinished) {
            fullTime += 1;
            if (millisUntilFinished > (60 * 1000)) {
                tvTimeLeft.setText(
                        String.format("%d:%d",
                                ((millisUntilFinished / (1000 * 60)) % 60),
                                (millisUntilFinished / 1000) % 60
                        )
                );
            } else {
                tvTimeLeft.setTextColor(getResources().getColor(R.color.red));
                tvTimeLeft.setText(String.format("%d sekundi", millisUntilFinished / 1000));
            }
        }

        @SuppressLint("SetTextI18n")
        public void onFinish() {
            tvTimeLeft.setText("Vrijeme je isteklo!");
            submitAnswersForQuizToServer();
        }
    }.start();
}
```

Sl. 3.31. *Metoda „setUpTimer()“ za postavljanje odbrojavanja*

Na korisničkom sučelju se u *RecyclerView* kontroli prikazuju pitanja i odgovori, odgovor se odabire pritiskom na željenu *RadioButton* kontrolu te se on sprema u listu objekata klase „*QuizSolution*“. Metoda koja kreira zahtjev za predaju rješenja kviza naziva se „*submitAnswersForQuizToServer()*“ koja se poziva ili nakon isteka vremena preostalog za rješavanje kviza ili nakon što korisnik pritisne dugme „Predaj“. Ova metoda kao parametre za zahtjev šalje ID kviza, listu objekata klase „*QuizSolution*“ te vrijeme koje je bilo potrebno za rješavanje kviza. Nakon obrade zahtjeva poslužitelj vraća objekt klase „*QuizReport*“. Obavještava se *RecyclerView* o novim podacima pa se na popisu pitanja i odgovora označuju odgovori koji su točni i broj ostvarenih bodova za svako pitanje, uz to se na mjestu gdje je pisalo vrijeme do završetka kviza prikazuje ukupan broj ostvarenih bodova. Ukoliko je kviz već rješavan od strane korisnika poslužitelj će vratiti šifru odgovara „500“ te se odgovori neće spremiti na poslužitelj, već će se korisniku prikazati poruka da je kviz već riješen.

```
private void submitAnswersForQuizToServer() {
    Call<QuizReport> call = RetrofitImpl.getQuizService().submitAnswersForQuiz(
            mCookie, mQuiz.getQuizId(), mAdapter.getmSolutions(), fullTime
    );

    call.enqueue(new Callback<QuizReport>() {
        @Override
        public void onResponse(Call<QuizReport> call, Response<QuizReport> response) {
            if (response.isSuccessful()) {
                QuizReport quizReport = response.body();
                mAdapter.setmSolutions(quizReport.getSolutions());
                tvTimeLeft.setText(String.valueOf(quizReport.getPoints()));
                btnSubmit.setVisibility(View.GONE);
                tvTimePoints.setText("Ostvareni bodovi:");
            } else if(response.code() == 500){
                Log.e( tag: "ERROR", response.toString());
                Toast.makeText(getApplicationContext(), text: "Ovaj test ste već rješavali!",
                        Toast.LENGTH_SHORT).show();
                finish();
            }else{
                finish();
            }
        }

        @Override
        public void onFailure(Call<QuizReport> call, Throwable t) {
            Log.e( tag: "ERROR", t.toString());
        }
    });
}
```

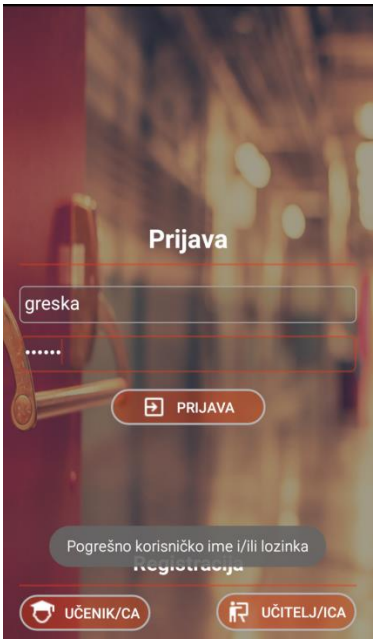
Sl. 3.32. *Metoda „submitAnswersForQuizServer()“*

3.3. Testiranje i način korištenja Android aplikacije

Testiranje se svodi na testiranje Android aplikacije na dva Android uređaja (Samsung Galaxy J3 (verzija Anroida 5.1.1., API 22) i Samsung Galaxy S6 Edge (verzija Androida 7.0., API 24) te na ugrađenom emulatoru (Nexus 5, API 28). Za uspješan rad aplikacije mora postojati veza s poslužiteljem, ukoliko ne postoji većinu podataka neće biti moguće dohvatiti i prikazati na korisničkom sučelju. Prilikom pokretanja aplikacije pojavljuje se početni zaslon (eng. *splash screen*) s animacijom *loga* aplikacije.

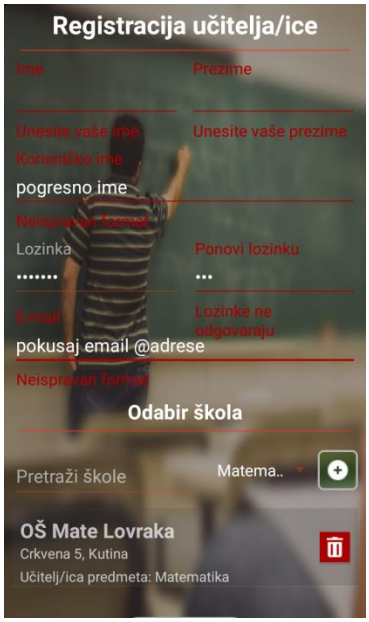
Sl. 3.33. *Početni zaslon aplikacije*

Tijekom animacije koja traje pet sekundi provjerava se postoji li spremljeni korisnik u zajedničkim postavkama uređaja, ako postoji prikazuje se glavni zaslon, a ako ne postoji prikazuje se zaslon za prijavu korisnika. Na zaslonu za prijavu korisnika unosi se korisničko ime i lozinka te se pritiskom na dugme „Prijava“ šalje zahtjev za autentifikaciju korisnika. Ukoliko je zahtjev uspješno obrađen pokreće se glavni zaslon, a ako nije prikazuje se poruka korisniku kao na slici 3.34.



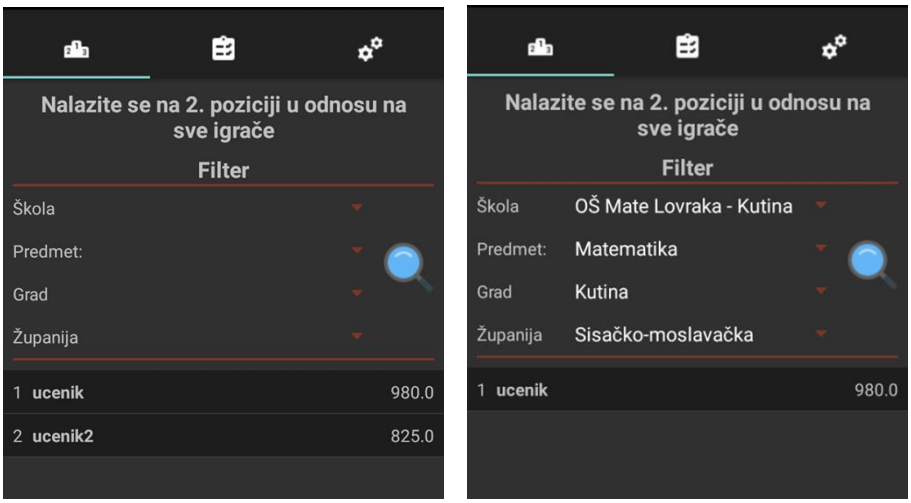
Sl. 3.34. *Neuspjela prijava*

Pritiskom na dugme „UČENIK/CA“ ili „UČITELJ/ICA“ pojavljuje se zaslon za registraciju korisnika. Pritiskom na dugme „Registracija“ pokreće se provjera validacije polja te se na odgovarajućim *TextInputLayout* kontrolama prikazuju greške kao što je prikazano na slici 3.35.



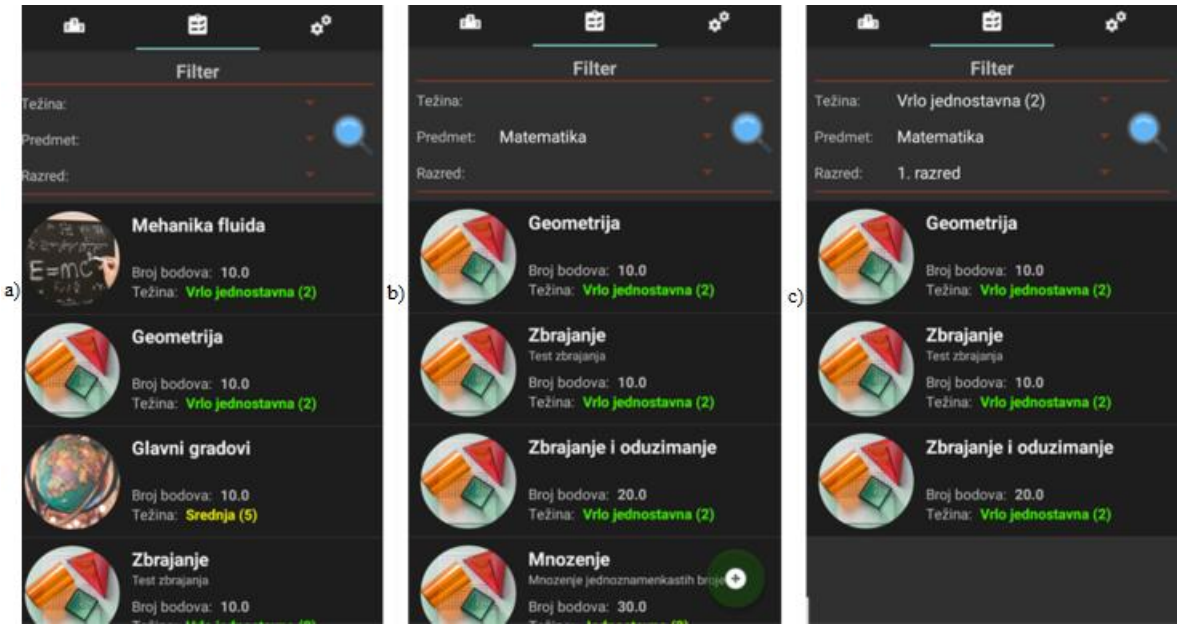
Sl. 3.35. *Greška pri pokušaju registracije korisnika*

Nakon uspješne prijave korisnika pokreće se glavni zaslon koji se sastoji od tri stranice (kartice). Na prvoj kartici nalazi se prikaz rang liste korisnika na temelju odabranog filtera. Razlika na korisničkom sučelju učenika i učitelja je u tome da za učenike postoji informacija o tome na kojoj poziciji se nalaze u odnosu na sve igrače. Rang lista sadrži informacije o poziciji korisnika s njihovim korisničkim imenom i ukupnim brojem ostvarenih bodova. Primjenom filtera poslužitelj treba dohvatiti samo one korisnike za koje vrijede postavljeni filteri (primjenom filtera dohvaćaju se korisnici koji pohađaju odabranu školu te se nalaze u odabranom gradu i županiji, a odabirom predmeta računa se broj bodova ostvarenih samo za taj predmet).



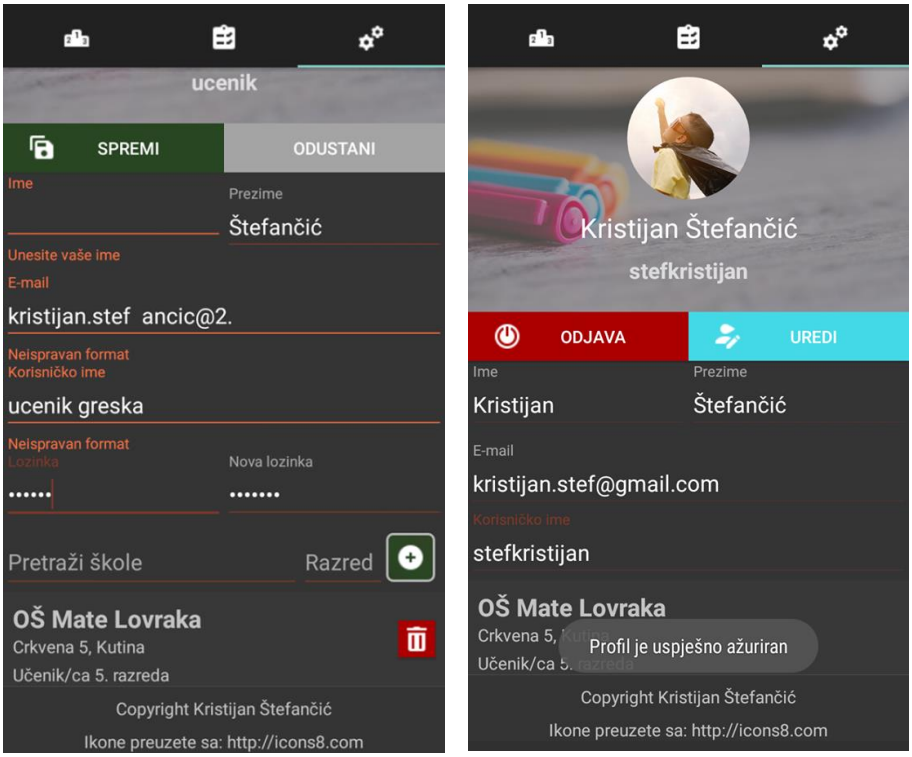
Sl. 3.36. *Rang lista s primijenjenim filterom (lijevo) i bez filtera (desno)*

Na drugoj kartici glavne aktivnosti nalazi se lista dostupnih kvizova. Postoji jedna razlika između korisničkih sučelja učenika i učitelja, a to je kontrola *FloatingActionButton* koja učiteljima omogućuje dodavanje novog kviza za predmet koji predaju. Prilikom prvog stvaranja korisničkog sučelja aplikacija s poslužitelja dohvaća sve omogućene kvizove koji su primjereni za razred koji učenik pohađa ili sve kvizove koji odgovaraju predmetu kojeg učitelj predaje u školi. Odabirom različitih filtera i slanjem zahtjeva, poslužitelj bi trebao dohvatiti listu kvizova na temelju primljenih vrijednosti filtera. U toj listi bi trebali biti kvizovi koji imaju odabranu težinu, koji su namijenjeni samo za odabrani razred i predmet).



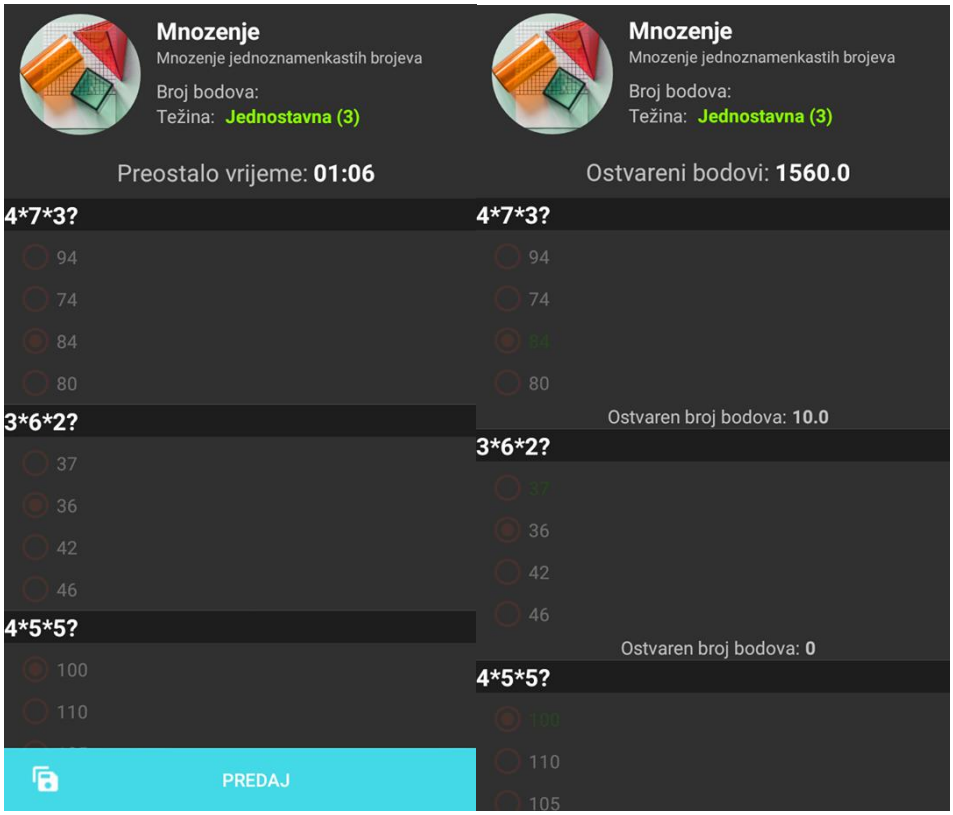
Sl. 3.37. *Lista kvizova za učenike bez filtera a), s filterom c) i za učitelje b)*

Na trećoj kratici glavnog zaslona prikazan je korisnički profil te dugme za odjavu i dugme za promjenu korisničkih podataka. Pritiskom na dugme „Odjava" korisnik se odjavljuje iz sustava te se prikazuje zaslon za prijavu korisnika, a pritiskom na dugme „Uredi" omogućuju se kontrole za upis korisničkih podataka i pojavljuje se dugme „Spremi" i dugme „Odustani". Pritiskom na dugme „Spremi" radi se validacija unesenih podataka te ukoliko postoje greške, prikaže se poruka na odgovarajućoj kontroli (kao i pri registraciji korisnika). Ako su podaci ispravni šalje se zahtjev na poslužitelj za ažuriranje korisničkih podataka nakon čega se o ishodu ažuriranja obavještava korisnik.



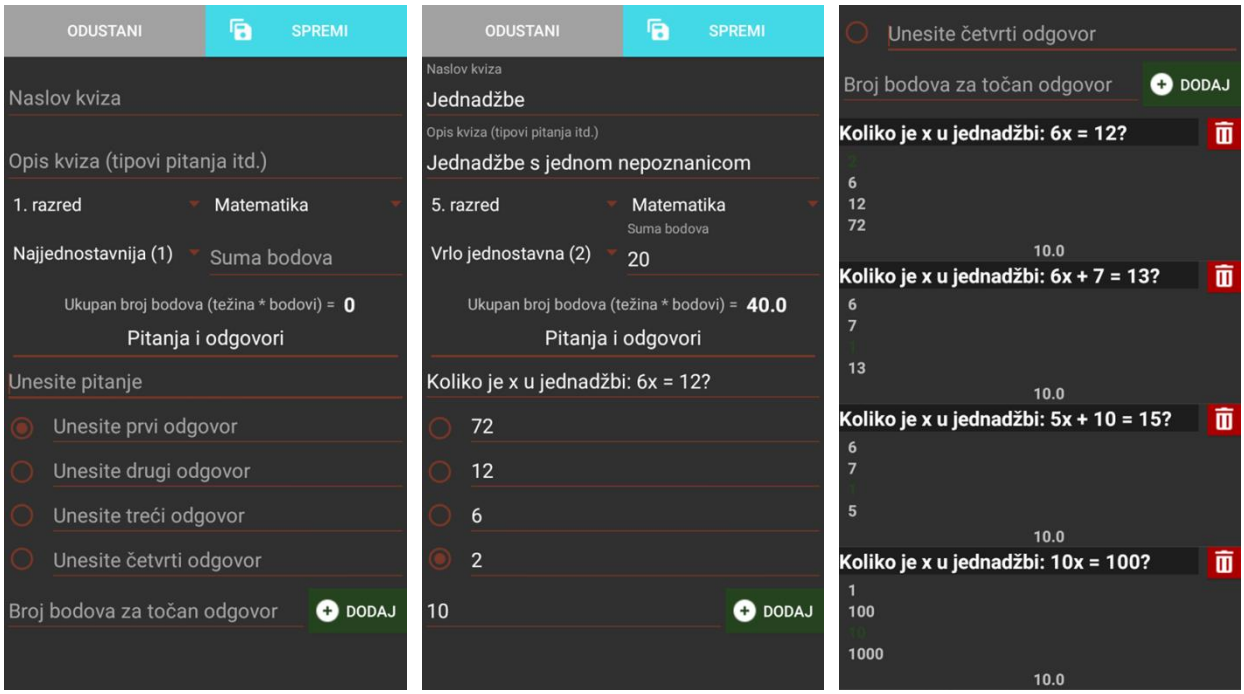
Sl. 3.38. *Spremanje promjena s greškama (lijevo) i s valjanim podacima (desno)*

Ukoliko je prijavljeni korisnik učenik omogućeno mu je pokretanje odabranog kviza iz liste kvizova na drugoj kartici glavnog zaslona. Prelaskom na zaslon za rješavanje kviza, automatski se pokreće odbrojavanje, prikazuju se pitanja zajedno s odgovorima od kojih je potrebno odabrati samo jedan (i samo jednom, ne može se mijenjati odabir odgovora naknadno). Nakon isteka vremena ili pritiskom na dugme „Predaj" spremljeni odgovori se šalju na poslužitelj koji obrađuje dobivene podatke te vraća rezultat zajedno s točnim odgovorima kako bi korisnik mogao vidjeti gdje je pogriješio.



Sl. 3.39. *Zaslon za rješavanje kviza s odabranim odgovorima prije predaje (lijevo) i ostvarenim rezultatom (desno)*

Ako je prijavljeni korisnik učitelj tada se na kartici za prikaz liste kvizova nalazi *FloatingActionButton* kontrola i pritiskom na nju pojavljuje se zaslon za dodavanje novog kviza. Na tom zaslonu prikazane su kontrole za unos naziva i opisa kviza, težine, predmeta i razreda za koji je kviz namijenjen i polja za unos pitanja i odgovora. Nakon unosa jednog pitanja i njegovih odgovora od kojih je potrebno označiti koji je točan, potrebno je pritisnuti dugme „Dodaj“ kako bi se pitanje dodalo na listu pitanja za kviz te će ta pitanja biti prikazana ispod kontrola za unos podataka. Pritiskom na dugme „Spremi“ novi kviz se sprema na poslužitelja i o tome se obavještava korisnik.



Sl. 3.40. *Primjer dodavanja novog kviza*

4. ZAKLJUČAK

Moderno doba donijelo nam je puno korisnih stvari (prijenosna računala, pametni telefoni, tableti itd.) koje nam olakšavaju svakodnevni život. Uz sve prednosti koje nam pružaju navedeni uređaji, dolazi do jednog vrlo bitnog problema. Djeca i mladi sve više koriste pametne telefone za zabavu (koriste različite društvene mreže, igraju igrice...) te zbog toga imaju manje vremena za školske i fakultetske obveze ili ih čak i u potpunosti zapostavljaju. Profesori i roditelji učestalo ističu prethodno navedene probleme te pokušavaju pronaći učinkovito rješenje kako bi educirali mlade na zanimljiv i zabavan način.

Android aplikacija za zadavanje zadataka osmišljena je upravo za tu namjenu te je glavna ideja aplikacije bila spojiti ugodno s korisnim odnosno spojiti upotrebu pametnih telefona s edukacijom. Korištenjem poslužitelja omogućena je povezanost korisnika diljem zemlje, a uz to poslužitelj predstavlja centralizirani spremnik podataka kojem svi korisnici mogu pristupati. Android aplikacija pruža razumljivo i jednostavno korisničko sučelje za prikaz i obradu podataka s poslužitelja. Rang liste su osmišljene kako bi učenik mogao procijeniti svoje znanje iz različitih predmeta u odnosu na druge vršnjake. Listu kvizova je lako filtrirati na različite načine te je na taj način korisniku lakše odabrati kviz kojeg bi želio rješavati. Rješavanje kviza se svodi na odabiru jednog od ponuđenih točnih odgovora za svako pitanje u zadanom vremenu, a rezultat rješavanja kviza korisniku daje uvid u točne odgovore čime može nadograditi svoje znanje pa čak i naučiti nešto novo. Sučelje za stvaranje novih kvizova je vrlo jednostavno, razumljivo i omogućuje brzo dodavanje pitanja i odgovora kao i uredan pregled svih postavljenih pitanja te uređivanje liste pitanja jednim pritiskom na ekran.

Za izradu ovog sustava korišteno je IntelliJ razvojno okruženje za izradu poslužiteljske aplikacije i Android Studio za razvoj Android aplikacije. Programski jezik korišten za razvoj poslužiteljske i klijentske aplikacije je Java. Moguće nadogradnje koje bi Android aplikaciju za zadavanje zadataka učinile zabavnijom za učenike i ugodnijom za korištenje su:

- bolje dizajnirano korisničko sučelje s različitim interaktivnim animacijama
- mogućnost osvajanja raznih virtualnih nagrada (poput trofeja i medalja)
- uz unos pitanja dodati mogućnost postavljanja slike zadatka te mogućnost stvaranja različitih vrsta zadataka

LITERATURA

[1] Spring Framework – Wikipedia
https://en.wikipedia.org/wiki/Spring_Framework (Stranica posjećena: 17.9.2018.)

[2] Spring Boot
https://spring.io/projects/spring-boot (Stranica posjećena: 20.9.2018.)

[3] Spring Data
https://spring.io/projects/spring-data (Stranica posjećena: 20.9.2018.)

[4] Spring Security
https://spring.io/projects/spring-security (Stranica posjećena: 20.9.2018.)

[5] Android Architecture Patterns Part 1: Model-View-Controller
https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6 (Stranica posjećena: 20.9.2018.)

[6] Consuming APIs with Retrofit
https://github.com/codepath/android_guides/wiki/Consuming-APIs-with-Retrofit (Stranica posjećena 19.9.2018.)

[7] Creating swipe views with tabs | Android Developers
https://developer.android.com/training/implementing-navigation/lateral (Stranica posjećena 19.9.2018.)

[8] The Android Arsenal – Splash Screens – Awesome Splash
https://android-arsenal.com/details/1/2605 (Stranica posjećena: 14.9.2018.)

[9] Stack Overflow
https://stackoverflow.com/ (Stranica posjećena: 15.9.2018.)

SAŽETAK

Android aplikacija za zadavanje zadataka

Glavni zadatak diplomskog rada je omogućiti učenicima interaktivno učenje, rješavanjem zadataka zadanih od raznih nastavnika, na pametnim telefonima. Izrađena je poslužiteljska aplikacija koja omogućuje manipulaciju podacima i služi kao centralizirani spremnik podataka te Android aplikacija koja učiteljima omogućuje kreiranje zadataka koje nakon toga učenici mogu rješavati na svojim pametnim telefonima. Zadatci koje učitelji kreiraju se spremaju u obliku kvizova koji se sastoje od nekoliko pitanja, a svako pitanje ima četiri odgovora od kojih je samo jedan točan. Učenici rješavanjem kvizova skupljaju bodove i na temelju tih bodova se kreira rang lista koju mogu vidjeti učenici i učitelji.

Ključne riječi: učenje, zadavanje zadataka, rješavanje zadataka, kviz, Android, *Spring Framework*, *Spring Boot*

ABSTRACT

Android application for task assignments

The main task of this graduate thesis is to provide interactive learning to students with the usage of their smartphones by solving tasks set by various teachers. A server application was made in order of data manipulation and to serve as a centralized data storage. Along with the server application there is an Android application that allows teachers to create tasks that students can solve using their smartphones. The tasks that teachers create are stored in the form of quizzes which are consisted of a number of questions that have exactly four answers and only one of them is correct. Students are collecting points by solving quizzes made by teachers and based on that points a ranking list is created that is visible to teachers and students.

Keywords: learning, task assignments, task solving, quiz, Android, Spring Framework, Spring Boot

ŽIVOTOPIS

Kristijan Štefančić rođen je 30. kolovoza 1994. godine u Zagrebu. Stanuje u Kutini, na adresi Hrvatskih branitelja 74. Nakon završetka osnovnoškolskog obrazovanja u OŠ Mate Lovrak Kutina, u rujnu 2009. godine je upisao Tehničku školu Kutina, smjer računalstvo gdje se pokazao kao jedan od najboljih učenika. 2004. godine osvojio je drugo mjesto na županijskom natjecanju Sisačko-moslavačke županije iz informatike. Maturirao je u lipnju 2013. godine s odličnim uspjehom te ostvario kvalifikaciju: tehničar za računalstvo. Polaganjem državne mature 2013. godine je upisao preddiplomski studij računarstva na Elektrotehničkom fakultetu u Osijeku. 2014. godine dobio je stipendiju za izvrsnost Sisačko-moslavačke županije. 2015. godine je sudjelovao na natjecanju *Case Study* za tvrtku Ožujsko, a 2016. godine se natjecao na Elektrijadi iz predmeta „Objektno orijentirano programiranje“ gdje je ostvario dobar rezultat. U istoj godini završava preddiplomski studij računarstva i time postiže titulu sveučilišni prvostupnih računarstva te iste godine upisuje diplomski studij računarstva – smjer programsko inženjerstvo. Godine 2017. pohađa i završava praksu na mjestu Android i back-end softver developera u tvrtki EM2 d.o.o. gdje je naposlijetku ostao raditi kao Spring Framework back-end developer. U svome obrazovanju stekao je određeno znanje iz elektrotehnike i računarstva, posjeduje odlično znanje rada na računalu, Microsoft Office-u (Word, Excel, PowerPoint, Visio) te programskim jezicima Java, C, C++, C#, SQL. Zna kreirati Android aplikacije te poslužiteljske aplikacije temeljene na Spring Frameworku. Posjeduje znanje o internetskim tehnologijama (MVC, HTML, CSS, JavaScript, TypeScript...). Posjeduje vrlo dobro znanje engleskog jezika te ima vozačku dozvolu B kategorije.

Potpis:

_____

PRILOZI

DVD

- Android Studio projekt
- Spring Boot (Maven) projekt
- Rad u .docx i .pdf formatu